Chapter 2

SW Architecture and Integration

2.1 Introduction

It has become accepted that there is a clear need for an 'architectural view' of systems [198]. EAs that satisfies nowadays enterprises' requirements should be presented. Software architecture is one class of EA that should address organizational requirements like systems interoperability, integration, agility, and other requirements [51, 172]. The architectural view of systems (both business and IT systems) is defined in ANSI/IEEE standard 1471-2000 as "the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution". Enterprises can be thought of as the combination of business needs, and IT capabilities. Agile organizations are those where IT serves business needs, not limiting it [221].

2.1.1 Enterprise Architecture Dimensions

Different dimensions of the enterprise need to be defined in order to generate the EA. From enterprise point of view, architectures are classified into four categories [51]: Business Architecture, Information Technology (IT) Architecture, Information Architecture, Application (software) Architecture as depicted in figure 2.1.1 presented in 24. EA tends to define the enterprise from the four dimensions in order to connect between them and present a complete view for the enterprise.

2.1.2 Business Architecture

A business or business process architecture defines the business strategy, governance, organization, and key business processes within an enterprise. The fields of Business Process Reengineering (BPR) and BPM focus on the analysis and design of business processes, not necessarily represented in an IT system [51]. Business Architecture defines the business roadmap usually via defining the business processes [29].

IT Architecture

The IT architecture defines the hardware and software building blocks that make up the overall IS of the organization [51]. IT architecture includes hardware and soft-

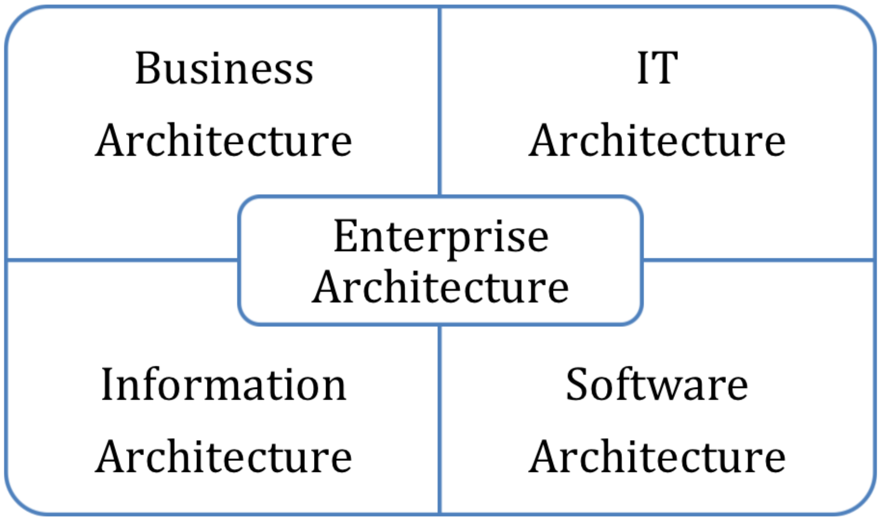

Figure 2.1: EA Dimensions

ware infrastructure including DB and middleware technologies. The IT architecture should enable achievement of the business goals using a software infrastructure that supports the procurement, development, and deployment of core mission-critical business applications. The purpose of the IT architecture is to enable a company to manage its IT investment in a way that meets its business needs by providing a foundation upon which data and application architectures can be built.

Information Architecture

The data architecture of an organization includes logical and physical data assets and data management resources. Information is becoming one of the most important assets a company has in achieving its objectives, and the IT architecture must support it. Information Architecture spans Business and IT Architectures, brings them together, keeps them together, and provides the necessary rich contextual environment to solve the ubiquitous data-quality problem [44].

Software Architecture

Application architecture serves as the blueprint for individual application systems, their interactions, and their relationships to the business processes of the organization. A software application is a computer program or set of programs that uses existing technologies to solve some end-user problem such as the automation of an existing business process. Software architecture can be defined as “the sum of the nontrivial modules, processes, and data of the system, their structure and exact relationships to each other, how they can be and are expected to be extended and modified, and on which technologies they depend, from which one can deduce the exact capabilities and flexibility of the system, and from which one can form a plan for the implementation or modification of the system” [156]. Application Architecture defines the form and function of the applications that will be developed to

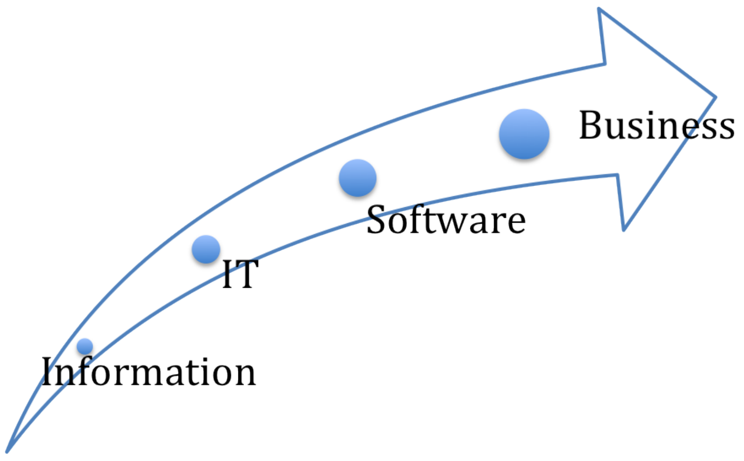

Figure 2.2: Enterprise Architecture Classes Utilization

deliver the required functionality of the system [198]. EA classes utilize each other, and build over each other. Distinctions between classes are blurred because they all serve each other, and serve the enterprise. EA classes' utilization can be thought of as depicted in figure 2.1.2 presented in page 25. Common software architectures are presented in order to address the architecture that might solve most well known software architecture limitations; like lack of scalability, integration, and interoperability with different IS. From this point on, Software Architecture and Architecture will be used interchangeably to refer to Software Architecture.

This chapter goes as follows: Section Two presents the relation between Software Architecture and Systems' Non-Functional Requirements. Section Three discusses the importance of Software Architecture. Section Four lists the most common Software Architecture Patterns while in Section Five authors present "Integration" as the selected criteria for comparing between different patterns. Section Six maps the different integration techniques and different Software Architecture Patterns; highlighting the driving and restraining forces for utilizing each one. Summary is presented in Section Seven with a proposed map between different integration techniques and software architecture patterns.

2.2 SW Architecture and System Design

There are aspects that should be designed within any system. Those aspects are: Architecture, Functionality, and Presentation as depicted in figure 2.2 presented in page 26. System design include: Architecturally defining subsystems and their functions, defining interface(s) for each subsystem and determining how those subsystems will interact, and allocating those subsystems to different components, as defined in [114].

Each of System Design's steps includes one or more functionality. Functionality can be mapped to system design steps as follows:

1. Architectural Design: includes identifying and documenting sub-systems mak-

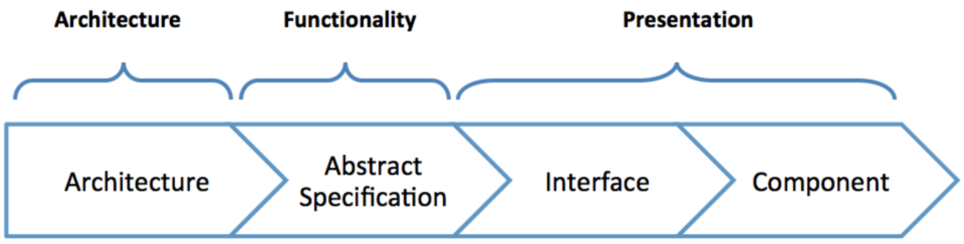

Figure 2.3: Architecture Design is one aspect of System Design

ing the system and their relationships.

2. Abstract Specification: includes producing an abstract specification of each sub-systems services and the constraints under which it must operate.

3. Interface Design: includes defining each subsystems interface with other subsystems is designed and documented.

4. Component Design: includes allocating services to different components and designing interfaces of these components.

Architecture is one aspect of a system to be designed, as well as the presentation of information and the functionality of the system. System requirements are either functional or non-functional [156]. Functional requirements are statements of services that systems should provide. Non-functional requirements present requirements that arose as a result of functional requirements. Architecture design is one step of systems design that shall satisfy non-functional requirements as it satisfies functional requirements [51]. Some of the non-functional requirements that shall be satisfied by architectural design are presented in figure 2.4 presented in page 27.

2.3 Importance of SW Architecture

Software (SW) architecture matters because a good one is a key element to success [156]. Though the fact that Systems have architectures but architectures are not systems [51] does not decrease architectures effects on systems. Some of the ways that software architecture influences success are [156]:

- Longevity: It is important to take time defining a good architecture because Architectures live longer than teams created them expects.
- Stability: Architectural stability helps ensure a minimum of fundamental rework as the system's functionality is extended over multiple release cycles, thus reducing total implementation costs.
- Degree and Nature of Change: Architecture determines the nature of change within the system. Some changes are perceived as applicable; others are perceived as not applicable.

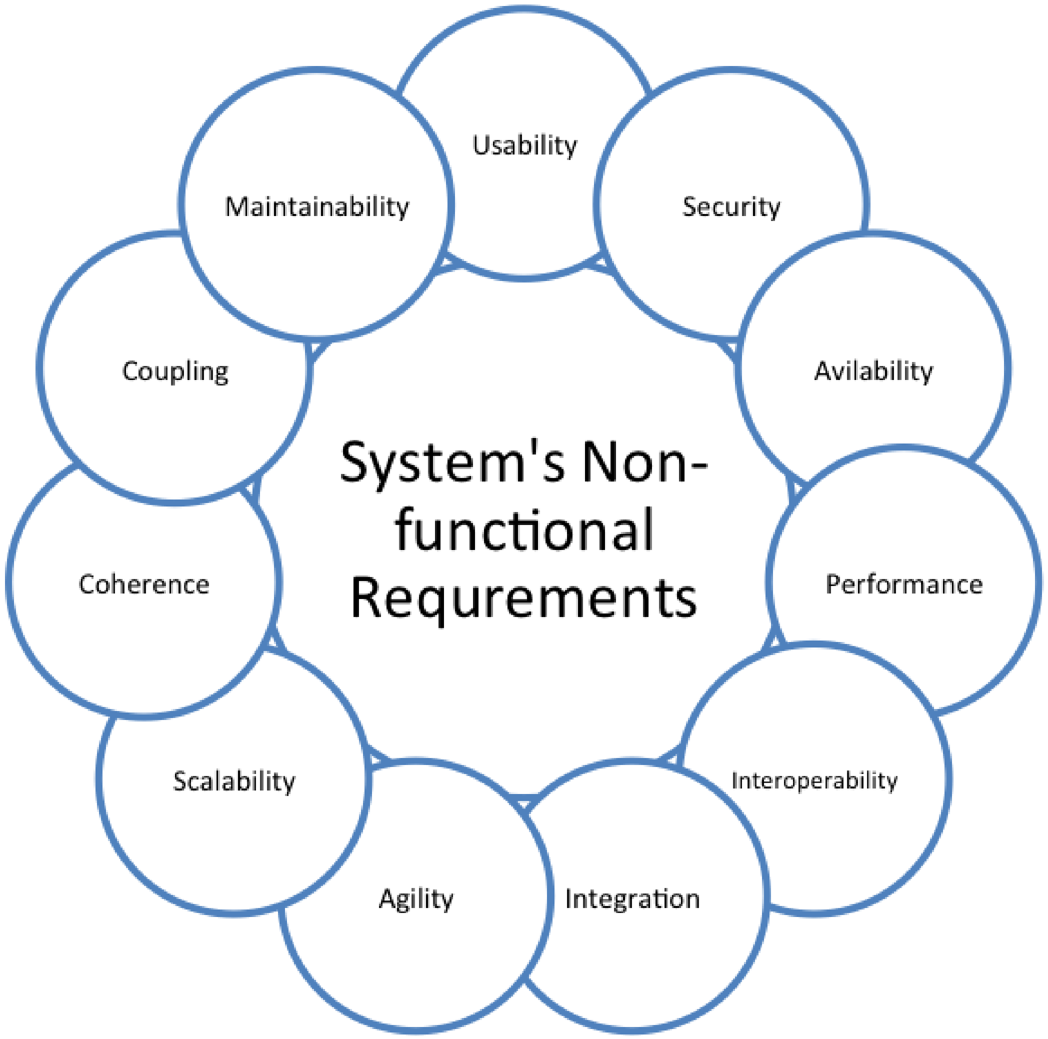

Figure 2.4: Systems non-functional requirements

- Profitability: Profitable architecture is the sustainable architecture within an acceptable cost.
- Social Structure: Architectures helps strengthening and weakening the social structure of the team. Teams can make social benefits form architectures during project time.
- Boundaries Defined: During the architectural design process the team makes numerous decisions about what is "in" or "out of" the system. These boundaries, and the manner in which they are created and managed, are vital to the architecture's ultimate success.

2.3.1 Distributed Approach Advantages

Advantages of using a distributed approach to systems development [246]:

1. Resource sharing A distributed system allows the sharing of hardware and software resources—such as disks, printers, files, and compilers—that are associated with computers on a network.
2. Openness Distributed systems are normally open systems, which means that they are designed around standard protocols that allow equipment and software from different vendors to be combined.
3. Concurrency In a distributed system, several processes may operate at the same time on separate computers on the network. These processes may (but need not) communicate with each other during their normal operation.
4. Scalability In principle at least, distributed systems are scalable in that the capabilities of the system can be increased by adding new resources to cope with new demands on the system. In practice, the network linking the individual computers in the system may limit the system scalability.
5. Fault tolerance The availability of several computers and the potential for replicating information means that distributed systems can be tolerant of some hardware and software failures. In most distributed systems, a degraded service can be provided when failures occur; complete loss of service only occurs when there is a network failure.

2.4 Common SW Architecture Patterns

As we seek to handle complexity in building software systems, we encounter various architectural styles better suited to solving a certain class of problems. For example, pipes and filters, blackboard architectures, layered architectures, and so on all have been used to solve specific problems in specific domains. However, other architectural styles provide a greater degree of general utility. In other words, the use of that architectural style is not necessarily limited to solving specific problems within specific domains. Instead, foundation elements of the solutions provided by that architectural style find their way into the foundations of software architecture, rather than being confined to alleviating problems in a specific domain. When developing software systems, service- oriented thinking pushes the boundaries of traditional software architecture and provides new insights into handling complexity, thus deriving commonality and increasing agility. An Architecture Pattern expresses a fundamental structural organization or schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them [1]. Software architecture patterns can be divided into two categories: Data flow and Control flow [156, 64]. Data Flow category focuses on functional modules and data transfers, while Control Flow category describes the way control is passed from one part of the system to the other. Data flow software architecture patterns include:

- Model-View-Controller
- Presentation-Abstraction-Control
- Pipe-And-Filter
- Layered Systems
- Microkernel
- Client-Server
- N-Tier
- Repository
- Blackboard
- Finite State Machine
- Process Control
- Multi Agent System
- SOA
- Master-Slave
- Interpreter
- Message Broker
- Message Bus
- Structural Model
- Peer-to-peer

On the other hand, control flow software architecture patterns include:

- Call And Return a.k.a. Main program And Subroutines
- Implicit Invocation a.k.a. Event Based
- Manager Model
- Emulated Parallel

2.5 Non-Functional Requirements

Shortages, limitations, and deficiencies of IS that include lack of agility support and limitations of integration and interoperability are two of the non-functional requirements category. Integration is the main challenge with institutions, and it will be tested against the mentioned software architectures as an indicator for the architecture satisfaction of non-functional requirements.

2.5.1 Design Issues of Distributed Systems

Some of the most important design issues that have to be considered in distributed systems engineering are [246]:

1. **Transparency** To what extent should the distributed system appear to the user as a single system? When it is useful for users to understand that the system is distributed?
2. **Openness** Should a system be designed using standard protocols that support interoperability or should more specialized protocols be used that restrict the freedom of the designer?
3. **Scalability** How can the system be constructed so that it is scaleable? That is, how can the overall system be designed so that its capacity can be increased in response to increasing demands made on the system?
4. **Security** How can usable security policies be defined and implemented that apply across a set of independently managed systems?
5. Quality of service How should the quality of service that is delivered to system users be specified and how should the system be implemented to deliver an acceptable quality of service to all users?
6. Failure management How can system failures be detected, contained (so that they have minimal effects on other components in the system), and repaired?

2.5.2 Integration

Application integration is a strategic approach to binding many IS together [192]. The need to integrate applications has been a requirement since business process automation was presented [121]. Application integration can be one of the forms:

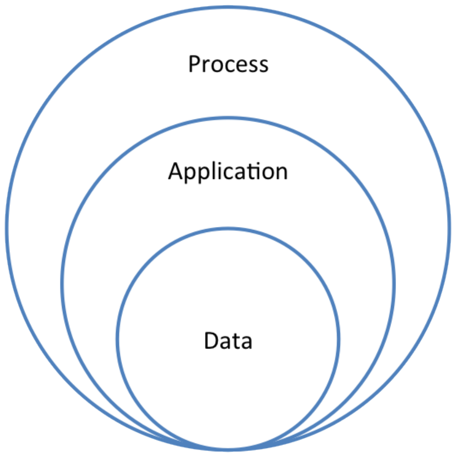

Figure 2.5: Integration Levels

Internal Application Integration, External Application Integration, or EAI [231]. Internal application integration techniques integrate organizational applications with each other, while external application integration techniques consider integrating organizational applications with applications outside organizational boundaries. EAI platforms centrally integrate heterogeneous system landscapes on process, method and data level [65]. Internal and External Application Integration present traditional integration levels, where EAI presents recent integration levels. Figure 2.5 presented in page 31 depicts different integration levels as presented in [121].

Data level integration allows data to be shared among applications without sharing its application logic. Application level integration integrates applications by getting both connected applications logic aware and enablers of exchanging data. Process level integration presents a new level of integrating applications where integration here facilitates or presents a new business processes.

2.6 SW Architectures and Integration Techniques

Integration is one of the addressed problems that requires picking suitable software architecture as a solution. The agile organization is more extensively integrated than previous organizational forms [15], and we need to have agile institutions. Considering how architecture works, advantages and shortages of the architecture, there are architectures that can overcome integration obstacles, and others cannot. There are many integration options that include File Transfer, Shared DB, and Messaging [157]. Considering integration techniques defined in [61], there are three common system integration techniques:

1. Enterprise Wide Standards
   - Standard Data Element Definition
   - Standard Enterprise Wide software

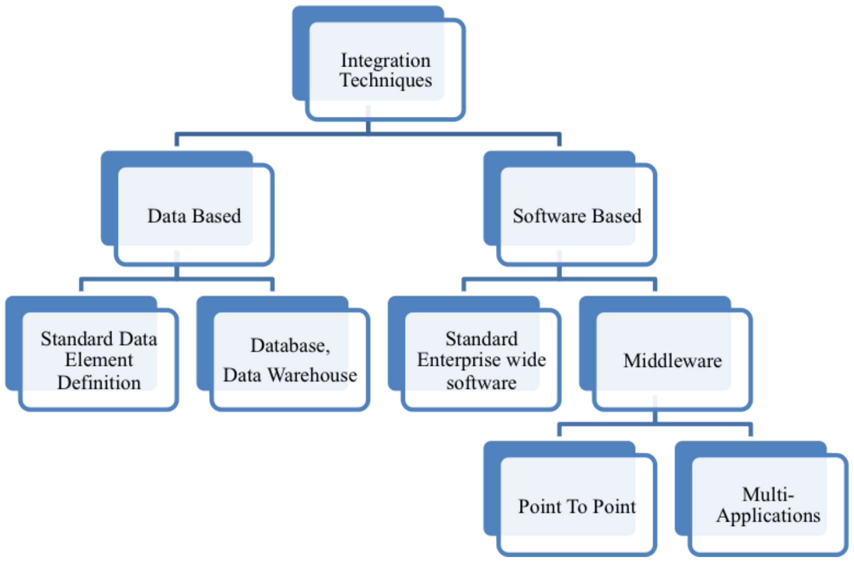

Figure 2.6: Integration Techniques

2. Middleware
   - TP, Remote Procedure Call (RPC), MOM, CORBA, DCOM,.. .
   - Web services
3. Additional Components
   - DW
   - Application Routers

Figure [44] presented in page 32 depicts a reorder of those integration techniques to be categorized in either one of two categories: Data or Software. Each category satisfies an integration level. Data oriented integration techniques satisfy Data level integration, and Software integration techniques satisfy Application level integration. Each integration technique has driving and restraining forces. Driving forces are forces that push software architects towards using the mentioned technique and supporting software architectures. Restraining forces are forces that push software architects not to use the mentioned technique and supporting architectures.

2.6.1 Data Based Integration Techniques

Those are techniques that relies on databases to present integration either by presenting unified standard data element definition, so all applications within the organization can share same data, or by presenting databases, and data warehouses as the organizational repository, and all running applications share same data repository. By using either technique method, applications are integrated; unfortunately; only on DB level. Several restraining forces against this technique are presented in the coming two sections.

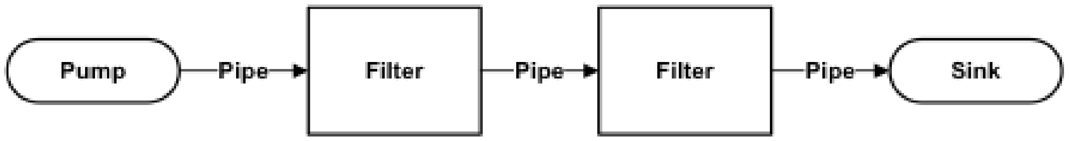

Figure 2.7: Pipe-And-Filter Architecture

Adopting Standard Data Element Definition

Custom software uses the same data element definitions as all other software within the system. Thus, uniqueness in syntax and semantics are presented. One of the Software Architectures that fits this technique is: Pipe-And-Filter software architecture.

- **Pipe and Filter Architecture:** A very simple, yet powerful architecture, that is also very robust. It consists of any number of components (filters) that transform or filter data, before passing it on via connectors (pipes) to other components. The filters are all working at the same time. The architecture is often used as a simple sequence, but it may also be used for very complex structures. Figure 2.7 presented in page 33 depicts mechanism of action of this software architecture [114, 64].

Components of this architecture are:

- Pump: or producer is the data source. It can be a static text file, or a keyboard input device, continuously creating new data, or data of another application.

- Pipe: is the connector that passes data from one filter to the next. It is a directional stream of data that is usually implemented by a data buffer to store all data, until the next filter has time to process it.

- Filter: transforms or filters the data it receives via the pipes with which it is connected. A filter can have any number of input pipes and any number of output pipes.

- Sink: or consumer is the data target. It can be another file, a DB, computer screen, or another application.

Applications share same standard data element definition, so, they can pipe data easily among them. By doing so, applications are integrated on data basis. Table 2.6.1 presented in page 34 shows Driving and Restraining forces of Adopting Standard Data Element Definition as an integration technique, which unfortunately makes adopting databases and data warehouses not the perfect solution for process level integration. Figure 2.8 presented in page 34 presents another view of the driving and restraining forces.

Table 2.1: Adopting Standard Data Element Definition Pitfalls

Driving Forces	Restraining Forces
Easier exchange o f data	Costs to develop standards definitions
Reduced development time	Costs to change existing systems
Reduced maintenance costs	Existing data definitions are different
	Some definitions need to be different
	Products use different data definitions
	Lack of industry standard definitions
	Mergers and acquisitions

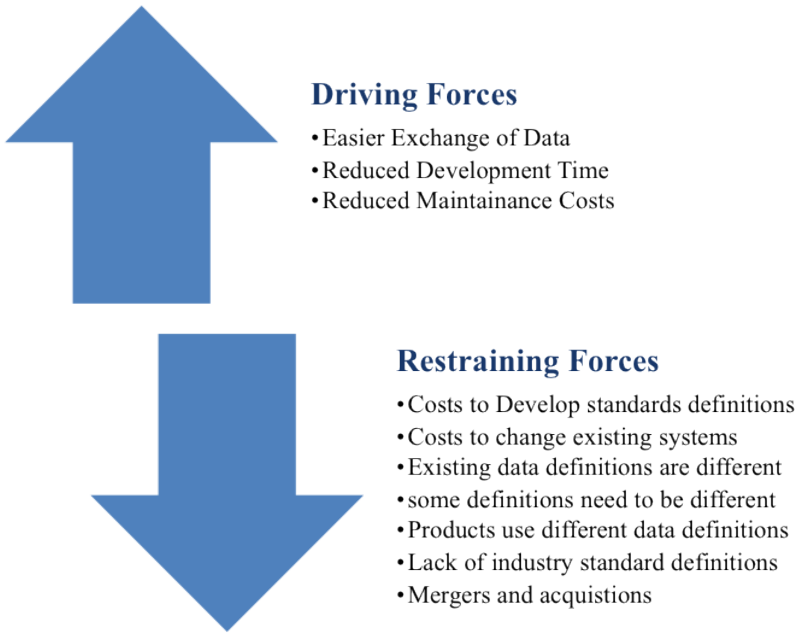
Driving Forces
•Easier Exchange of Data
•Reduced Development Time
•Reduced Maintainance Costs
Restraining Forces
•Costs to Develop standards definitions
•Costs to change existing systems
•Existing data definitions are different
•some definitions need to be different
•Products use different data definitions
•Lack of industry standard definitions
•Mergers and acquistions

Figure 2.8: Adopting Standard Data Element Definition Pitfalls

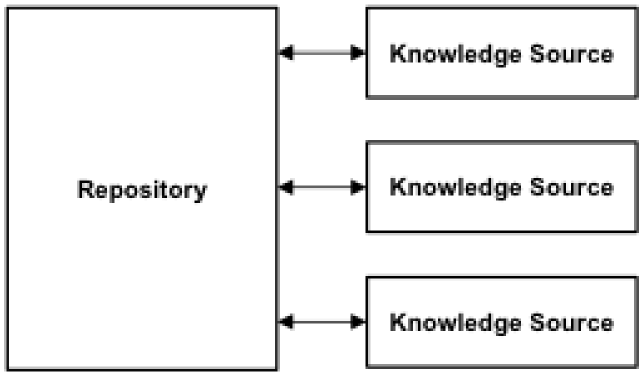

Figure 2.9: Repository Architecture

Adopting DB and DW

DB is a collection of persistent data [1] that is used by the application systems of some given enterprise [2]. Databases have too many advantages and they play an important role in today organizations. Benefits of data approach include sharing of data, and reducing redundancy [96]. Databases allow data to be shared, that means existing and new applications can share the data in the DB. Databases reduce data redundancy by eliminating, or at least controlling redundancy. DW is the subject-oriented, integrated, nonvolatile [3], time variant data store in support of management's decisions [165, 167]. DW can be thought of as a collection of data from more than one data source (data sources include databases) for providing a single, quiet large repository for the entire organization. Databases and data warehouses have been widely used to integrate applications within organizations. Typical DW can be found in [93]. One of the Software Architectures that fits this technique is: Repository software architecture.

Repository Software Architecture The repository contains a single data structure, the Repository, and a number of modules called Knowledge Sources, that modify this data structure. These are the only characteristics of repository architecture [114, 64]. Figure 2.9 presented in page 35 depicts how repository architecture behaves.

Table 2.6.1 presented in page 36 depicts the driving and restraining forces of adopting databases and data warehouses as an integration solution, which unfortunately makes adopting databases and data warehouses not the perfect solution for process level integration. Unfortunately, there are restraining forces that make adopting databases and data warehouses not the perfect solution for internal and external organizational applications integration. Figure 2.10 presented in page 37

[1]Persistent Data are data that once accepted by Database Management System (DBMS) for entry, can subsequently be removed only by some explicit request to DBMS.

[2]Enterprise is simply a convenient generic term for any reasonably self-contained commercial, scientific, technical or other organization.

[3]Nonvolatile means that, once inserted, data cannot be changed, though it might be deleted.

Table 2.2: Adopting Databases and Data warehouses Pitfalls

Driving Forces	Restraining Forces
Easier access to enterprise wide data	Costs of development
Reduced development time	Different semantics in data sources
Reduced maintenance costs	Semantic translation
Minimal effect on operational system	Lack of industry standard definitions
Usage of BI software	Deciding what data to warehouse
	Delays in getting data to warehouse
	Redundancy of data
	Data quality issues
	Brittleness of fixed record exchanges
	Performance tuning

depicts another view of adopting databases and data warehouses pitfalls.

2.6.2 SW Based Integration Techniques

Another integration technique presented is the one based on using software as the orientation of integration solution. Either using standard enterprise wide software or presenting middleware is a software oriented solutions.

Standard Enterprise Wide Software

This means that whole organization uses the same software; this means that the whole organization is using the same data definitions, semantics, and formats for exchanging data. ERP presents this technique as a solution to whole enterprise integration solution. ERP integrates SCM System, Enterprise Management System, CRM, e-commerce [89, 151, 226]. ERP providers list include Oracle [37], and SAP [39]. To provide a unified solution to the enterprise, layered architecture is required to enable integration of applications in different aspects. Software Architectures that fit this technique include: Layered Systems, Client-Server, and N-tier architecture.

- **Layered Systems:** Layered Systems use layers to separate different units of functionality. Each layer only communicates with the layer above and the layer below. Each layer uses the layer below to perform its function. Communication happens through predefined, fixed interfaces. A Layer is a design construct. It is implemented by any number of classes or modules that behave like they are all in the same layer. That means that they only communicate with classes in layers immediately above or below their layer and with themselves. Figure 2.11 presented in page 38 depicts layered systems architecture [64]. Each layer offers its own kind of functionality. A higher layer uses its lower layer to perform its function. It requires its lower layer. It is possible to define multiple layers at the same level. The user calls a function on an object in the upper layer. This object calls functions in the layer below. These functions in turn approach the layer below and the layer above. The primary disadvantage of layered systems

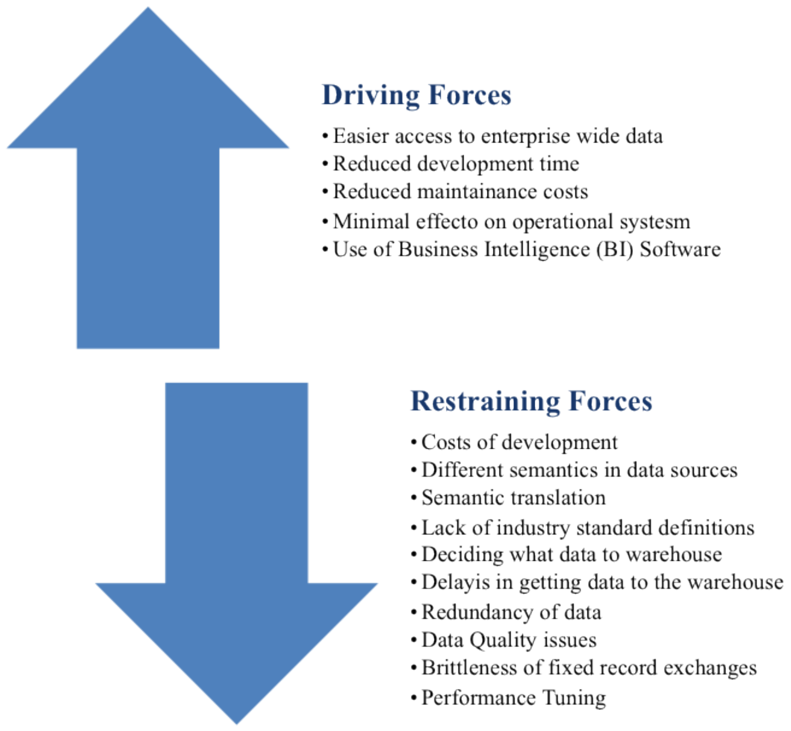
Driving Forces
• Easier access to enterprise wide data
• Reduced development time
• Reduced maintainance costs
• Minimal effecto on operational systesm
• Use of Business Intelligence (BI) Software
Restraining Forces
• Costs of development
• Different semantics in data sources
• Semantic translation
• Lack of industry standard definitions
• Deciding what data to warehouse
• Delayis in getting data to the warehouse
• Redundancy of data
• Data Quality issues
• Brittleness of fixed record exchanges
• Performance Tuning

Figure 2.10: Adopting Databases and Data warehouses Pitfalls

is that they add overhead and latency to the processing of data, reducing user-perceived performance [91]. Another software architecture that ERPs heavily rely on integrating enterprise applications is the Client-Server and N-Tier architecture. A detailed SAP architecture based on N-Tier architecture is presented in [54]. There are five architectural patterns for layered systems:

1. Master-slave architecture, which is used in real-time systems in which guaranteed interaction response times are required.
2. Two-tier client–server architecture, which is used for simple client–server systems, and in situations where it is important to centralize the system for security reasons. In such cases, communication between the client and server is normally encrypted.
3. Multitier client–server architecture, which is used when there is a high volume of transactions to be processed by the server.
4. Distributed component architecture, which is used when resources from different systems and databases need to be combined, or as an implementation model for multi-tier client–server systems.
5. Peer-to-peer architecture, which is used when clients exchange locally stored information and the role of the server is to introduce clients to each other. It may also be used when a large number of independent computations may have to be made.

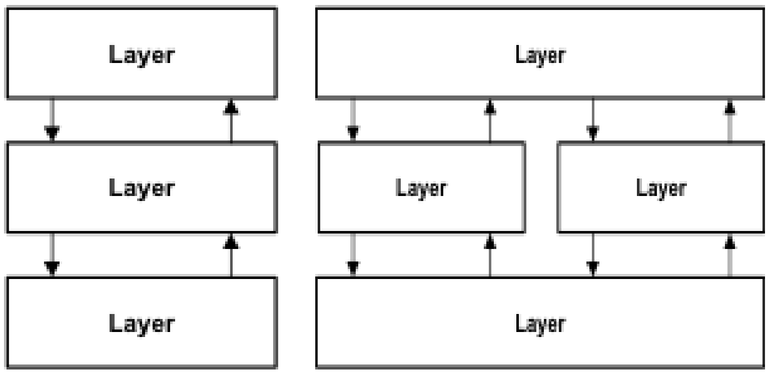

Figure 2.11: Layered Architecture Different Implementations

- **Client – Server Architecture:** Distributed systems that are accessed over the Internet are normally organized as client–server systems. In a client–server system, the user interacts with a program running on their local computer (e.g., a web browser or phone-based application). This interacts with another program running on a remote computer (e.g., a web server). The remote computer provides services, such as access to web pages, which are available to external clients. This client–server model, is a very general architectural model of an application. It is not restricted to applications distributed across several machines. You can also use it as a logical interaction model where the client and the server run on the same computer. In a client–server architecture, an application is modeled as a set of services that are provided by servers. Clients may access these services and present results to end users. Clients need to be aware of the servers that are available but do not know of the existence of other clients. Clients and servers are separate processes. The client-server style is the most frequently encountered of the architectural styles for network-based applications [7]. Client - Server components are: Client triggers process, Server reactive process [56]. Clients make requests that trigger reactions from servers, as presented in figure 2.12 presented in page 39. Variety of Client – Server systems are surveyed in [241, 261]. Client – Server architecture mainly consists of two layers: Single Server and many clients. Figure 2.13 presented in page 40 depicts Client – Server Architecture different implementations. A Client-Server system is one in which the server performs some kind of service that is used by many clients [64]. The clients take the lead in the communication. The basic form of client-server does not constrain how application state is partitioned between client and server components, it is often referred to by the mechanisms used for the connector implementation, such as remote procedure call or message-oriented middleware [7].

- **N-Tier Software Architecture:** N-Tier architecture is a Client-Server architecture combined with the Layered architecture where N equals three or higher [64]. Three-Tier architecture is an example of N-Tier architecture. Figure 2.14 presented in page 41 depicts Three-Tier Architecture. Three-Tier

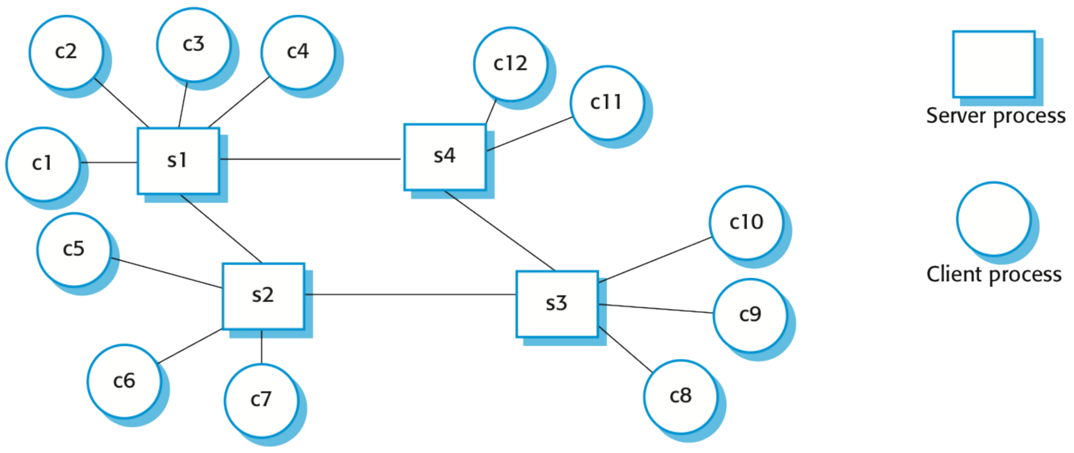

Figure 2.12: Layered Architectural Model of Client-Server [246]

architecture consists of three layers: Presentation, Application, and DB.

- **Presentation Layer**: deals with user interactions. Thin clients interface do not contain business logic, just code required to process user input, send requests to the server, and show the results of these requests.
- **Application Layer**: processes client requests. It is the actual web application that performs all functionality specific to the web application. However, it does not store the persistent data itself. Whenever it needs data of any importance, it contacts the DB server.
- **DB Layer**: contains DB and DBMS (DBMS).

Other applications can be divided into four layers [246]. Figure 2.15 presented in page 41 shows that four Layers applications can be:

1. A presentation layer that is concerned with presenting information to the user and managing all user interaction;

2. A data management layer that manages the data that is passed to and from the client. This layer may implement checks on the data, generate web pages, etc.;

3. An application processing layer that is concerned with implementing the logic of the application and so providing the required functionality to end users;

4. A DB layer that stores the data and provides transaction management services, etc.

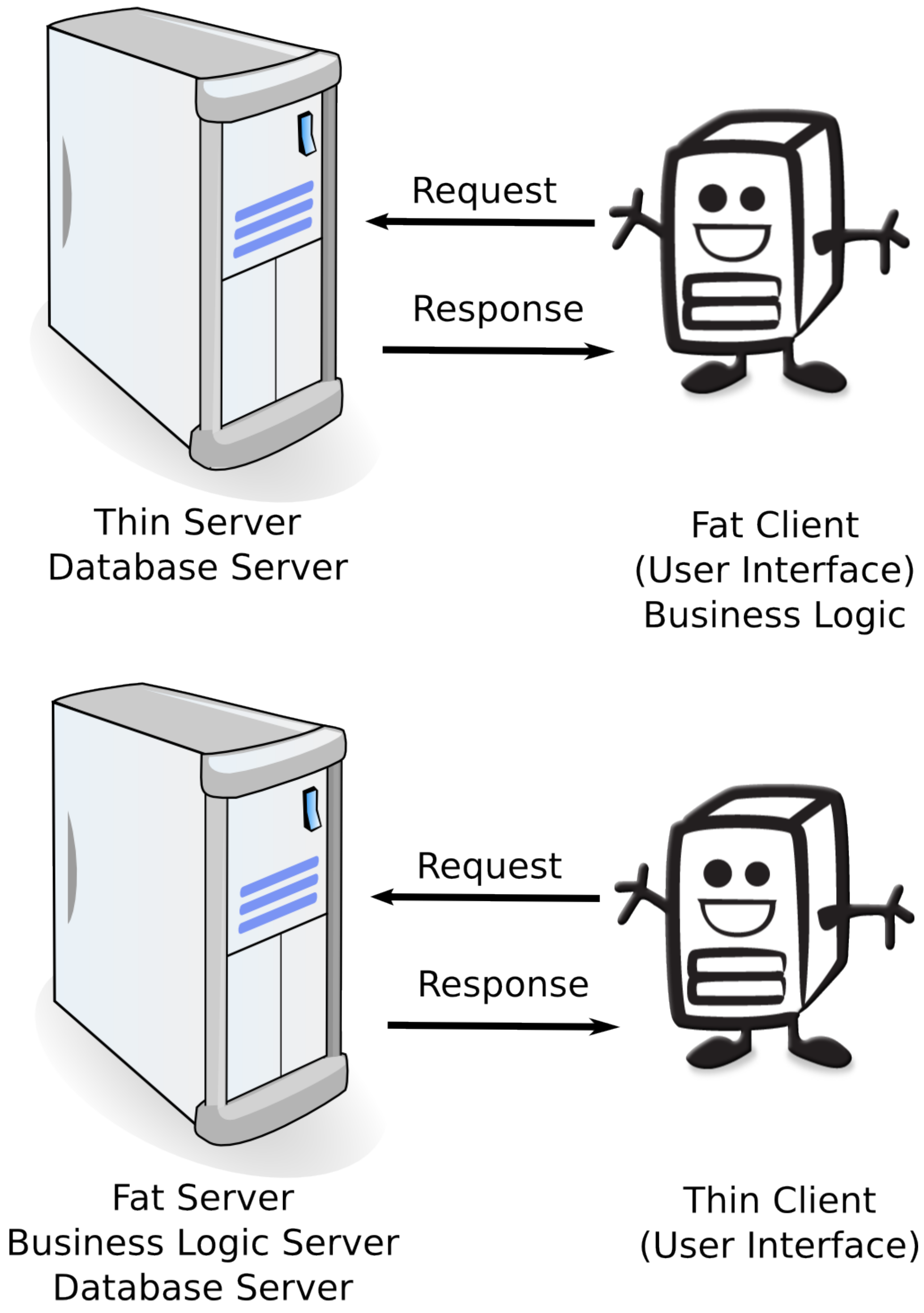

Figure 2.13: Client - Server Architecture Different Implementations

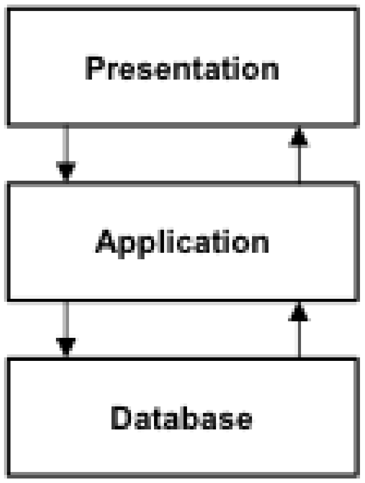

Figure 2.14: Three Tier Architecture

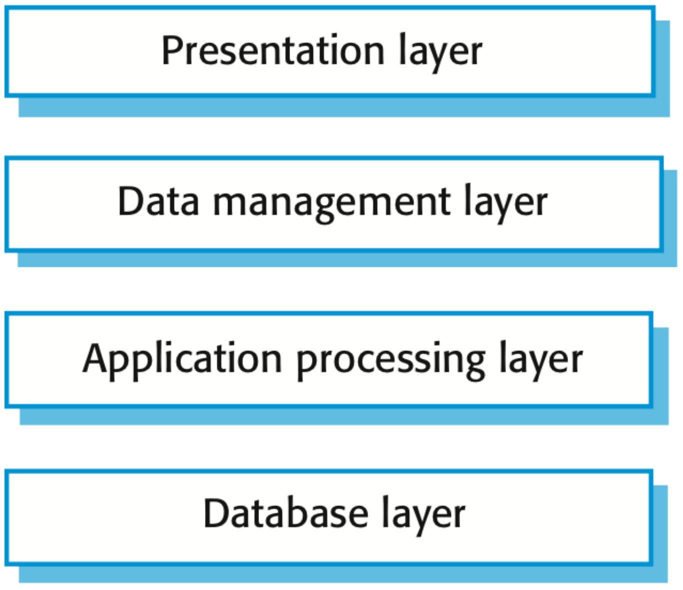

Figure 2.15: Layered architectural model for client–server application [246]

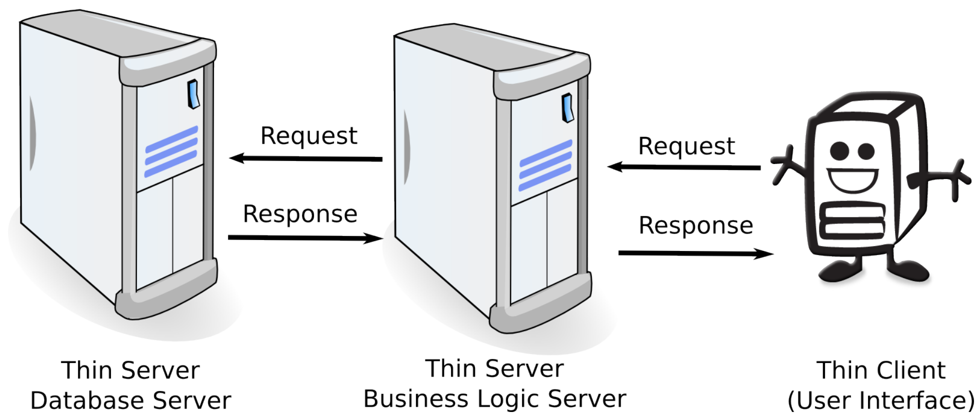

Figure 2.16: Three Tier Architecture Implementation

In the second and third tier there can be multiple instances, because of scalability, load-balancing and redundancy [10]. N-tier architecture (with N more than 3) is really 3 tier architectures in which the middle or bottom tier is split up into new tiers. Figures 2.16, 2.17, 2.18, and 2.19 present different implementations of N-Tier architecture. Figure 2.16 presented in page 42 shows the simplest implementation of Three-Tier architecture, where the most left layer that includes Smart Phone, Cell Phone, and PC presents required logic to display information and validate user inputs. Business logic ‘presented in applications’ required to provide web applications to users are present on the Business Logic server. Required data to support those applications are present in the DB server.

Figure 2.17 presented in page 43 illustrates the repetition of data layer by presenting more than one DB server. DB servers can be connected together or separated. Business logic server can connect to both DB servers, or connect to one DB server incase DB servers are connected together. Servers’ connections is an architectural decision that software architect shall consider while designing software architecture.

Figure 2.18 presented in page 44 presents another N-Tier implementation which replicates business logic servers. Scalability can force such a solution. Business logic servers can be connected to each other or not based on application requirements.

Figure 2.19 presented in page 45 presents the third N-Tier implementation where business logic and DB servers are replicated. Large scaled institutions require this type of architectures to face increasing requirements. N-Tier Architecture serves integration by forcing enterprises to implement standard software at each tier. Even if different software is implemented, by connecting each tier to the other, enterprises overcome integration problems. N-Tier has been widely accepted in today’s organizations as the standard architecture. Unfortunately, N-Tier architecture does not define what tier should include, how tiers should connect to each other, thus, architectural decisions still needs to be taken by software architect. Table 2.6.2 shows driving and restraining forces on adopting wide standard software as an integration solution [104, 33, 34]. Unfortunately, this integration technique is not well suited for process level integration. Figure 2.20 presented in page 45 presents another view of enterprise software pitfalls.

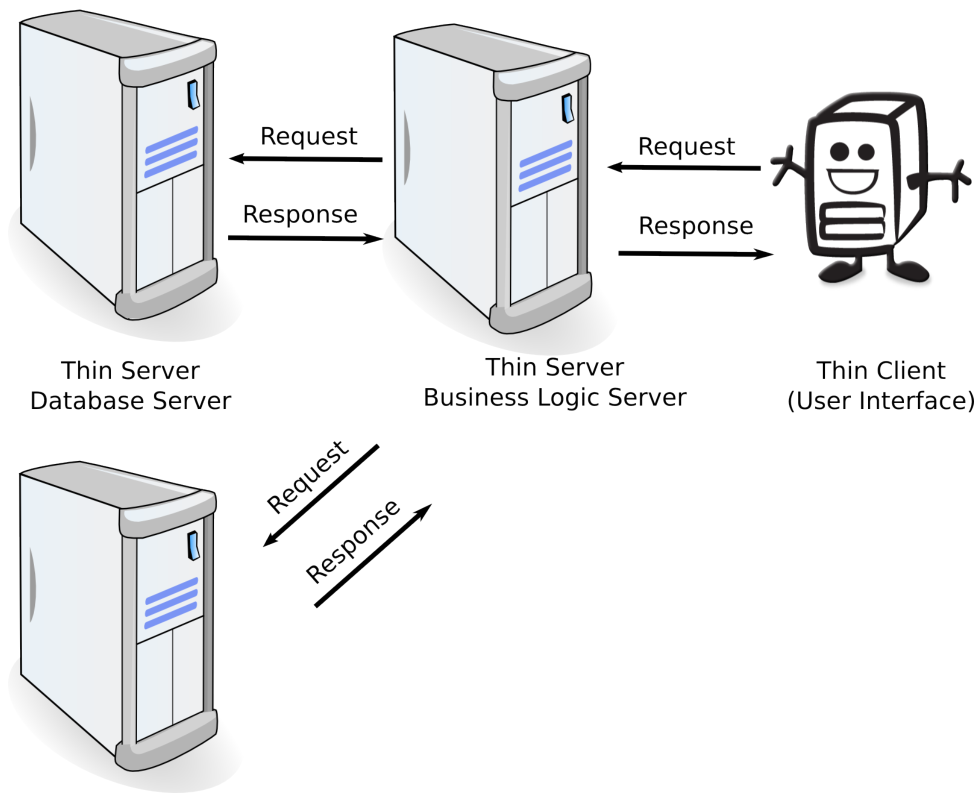

Figure 2.17: Three Tier Architecture First Implementation

Table 2.3: Adopting Standard Enterprise Software Pitfalls

Driving Forces	Restraining Forces
Easier access to enterprise wide data	Mergers and acquisitions
Reduced development time	Departments have different needs
Reduced maintenance costs	Dependence on different software
	Conversion to new software

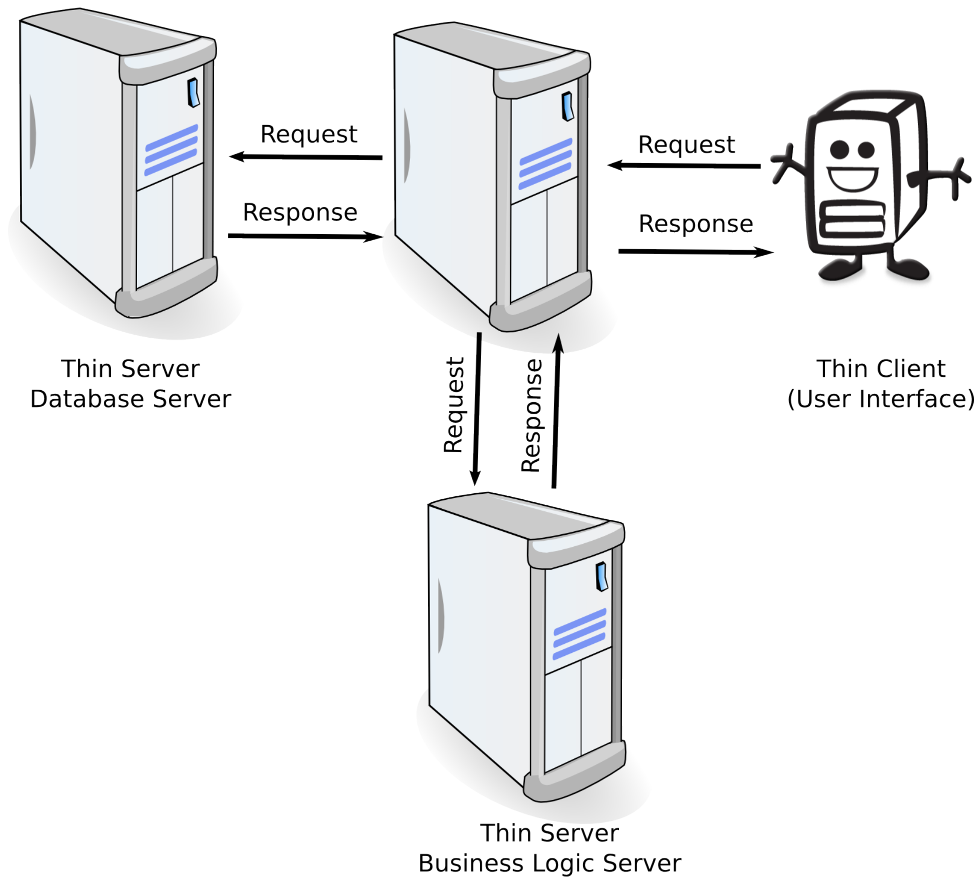

Figure 2.18: Three Tier Architecture Second Implementation

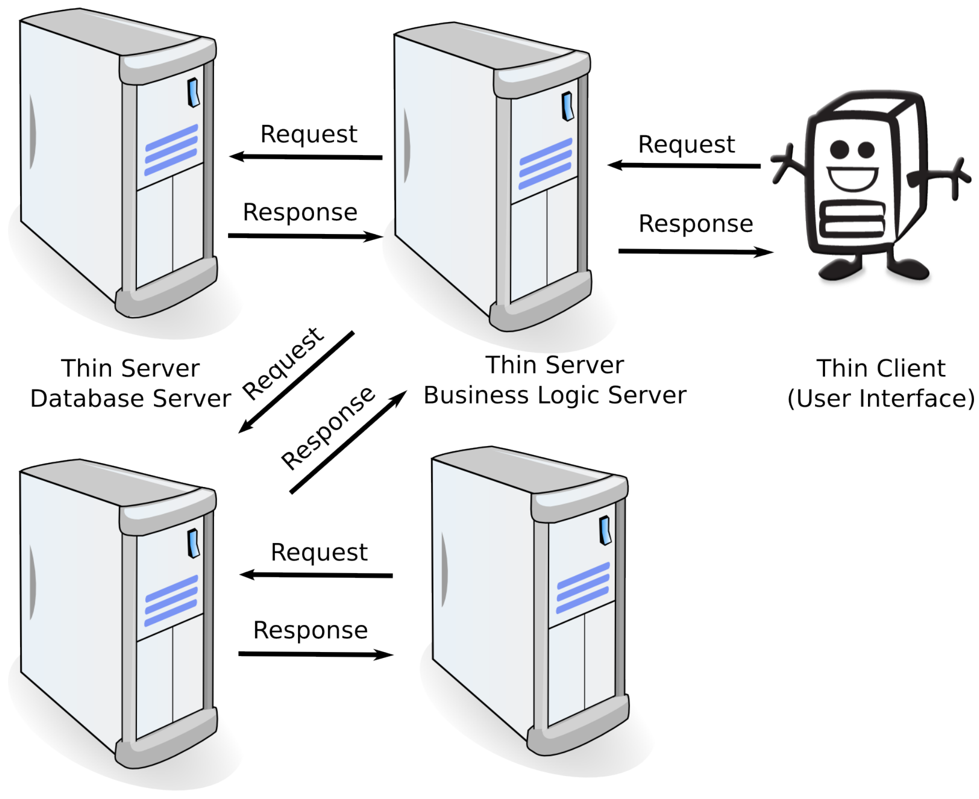

Figure 2.19: Three Tier Architecture Third Implementation

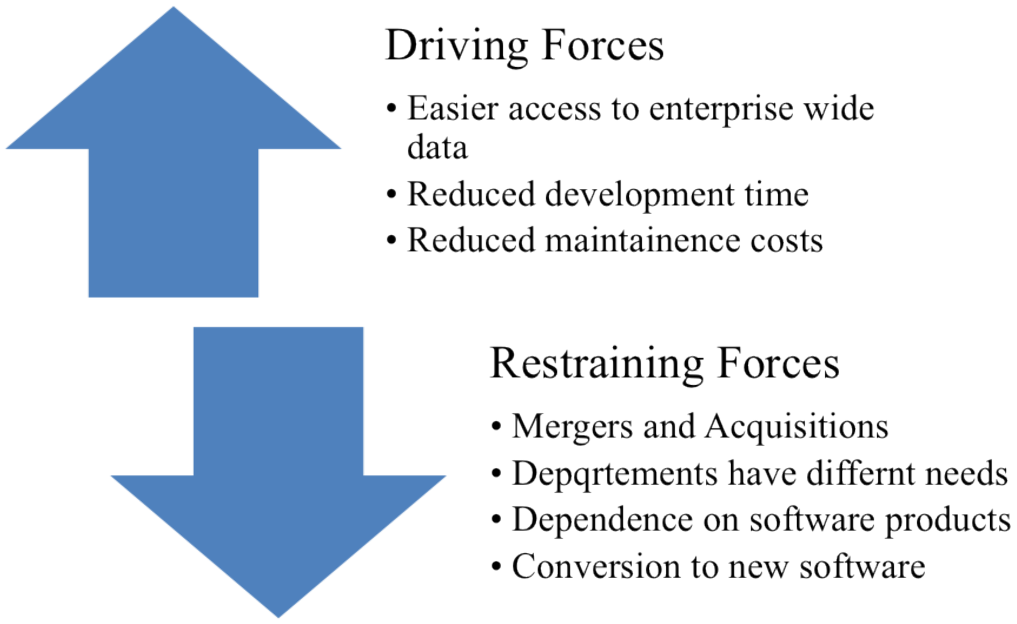
Driving Forces
• Easier access to enterprise wide data
• Reduced development time
• Reduced maintainence costs
Restraining Forces
• Mergers and Acquisitions
• Depqrtements have differnt needs
• Dependence on software products
• Conversion to new software

Figure 2.20: Adopting Standard Enterprise Software Pitfalls

Middleware Integration Technique

Middleware integration facilitates communication between applications in order to exchange data and business logic. This communication is either: Point-To-Point, or Multi-Applications.

- **Point To Point Middleware:** Point To Point refers to direct connection and communication between two applications. This direct communication can be presented by two methods: Adapters, and Remote Procedure Calls (RPCs).
  - Application Adapters: In this case, a special purpose adapter is presented to integrate two applications and provide communication and exchange of data among them. Examples of adapters are presented in [51]. This adapter is capable of transforming data syntax and semantics, and managing communication among connected application. Special adapters are presented for each application.
  - RPC: an RPC allows execution of program logic on a remote system by calling local routine. This calling is done via special message between client and server [4].
- **Multi-Applications Middleware**: The components in a distributed system may be implemented in different programming languages and may execute on completely different types of processor. Models of data, information representation, and protocols for communication may all be different. A distributed system therefore requires software that can manage these diverse parts, and ensure that they can communicate and exchange data. The term 'middleware' is used to refer to this software—it sits in the middle between the distributed components of the system. Middleware is normally implemented as a set of libraries, which are installed on each distributed computer, plus a run-time system to manage communications. Middleware is general-purpose software that is usually bought off the shelf rather than written specially by application developers. Examples of middleware include software for managing communications with databases, transaction managers, data converters, and communication controllers. Multi-Applications middleware relies heavily on messages in order to hide complexity of communication between systems. It is not acceptable to provide adapters for the whole system; instead, a middleware is required to integrate applications within the system [61]. Exchanging messages enabled integration on software level. Table 2.6.2 presented in page 47 depicts driving and restraining forces of adopting message routing integration technique. Software architectures supported different implementations for integrating applications via exchanging messages. Those architectures include: Message Broker, Message Bus, and Broker. Figure 2.21 presented in page 47 depicts another view of adopting message routing pitfalls.
  - Message Bus: Message bus architecture connects all applications through the logical component known as Message Bus [28]. Message bus is respon-

[4]In this case, machine that initiated RPC is considered Client, while the machine that holds code to be invoked and executed by reception of RPC is Server.

Table 2.4: Adopting Message Routing Pitfalls

Driving Forces	Restraining Forces
Consistent enterprise wide data	Costs of development
Reduced development time	Different semantics in data sources
Reduced maintenance costs	Semantic translation
Minimal effect on systems	Lack of industry standard definitions
	Deciding what data to route
	Delays getting data updates distributed
	Data quality issues
	Brittleness of fixed record exchange

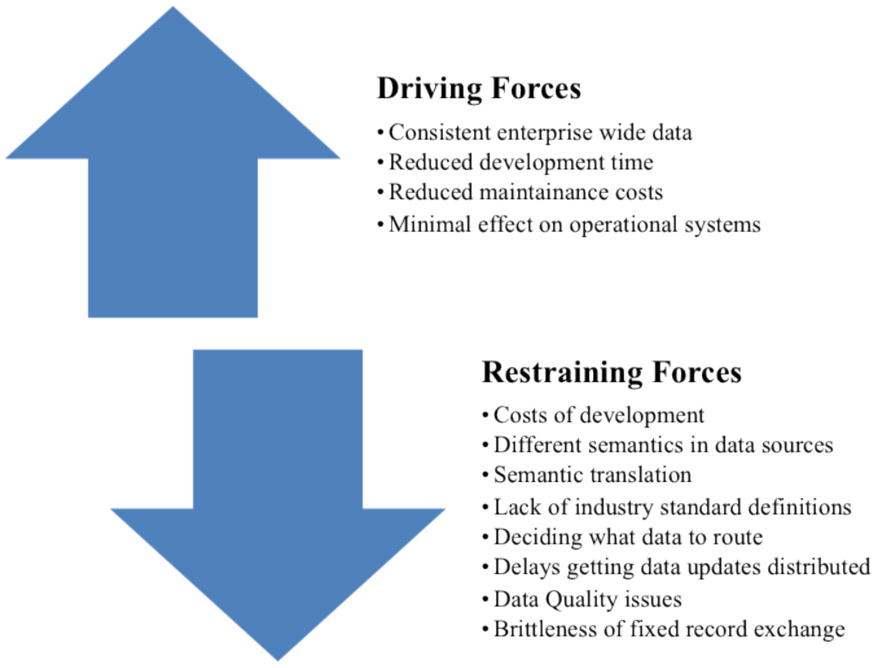
Driving Forces
• Consistent enterprise wide data
• Reduced development time
• Reduced maintainance costs
• Minimal effect on operational systems
Restraining Forces
• Costs of development
• Different semantics in data sources
• Semantic translation
• Lack of industry standard definitions
• Deciding what data to route
• Delays getting data updates distributed
• Data Quality issues
• Brittleness of fixed record exchange

Figure 2.21: Adopting Message Routing Pitfalls

sible for delivering messages to destined applications. Sending application has no responsibility of direct connection with destined application. Sending application needs to prepare the message in the message bus agreed format, and delivers it to the message bus. Figure 2.22 presented in page 49 depicts how message bus architecture works. There are three options for subscribing the application to the Message Bus [38]:

* List-Based Publish/Subscribe: There are managed lists for subscribers. All subscribers receive certain messages when they are sent to the bus.
* Broadcast-Based Publish/Subscribe: Messages are sent to the all connected applications.
* Content-Based Publish/Subscribe: it enables Message Bus to investigate message contents to determine recipient of the message.

Limitations include that message bus can expose only one interface. In order for an application to be part of the architecture, it should implement the message bus interface, use the message bus messages type, and registers itself to the message bus. Performance issues are also a potential problem; poor application partitioning can create excessively high volumes of messages, and some use cases can be impacted through high network latency [155]. One of middleware integration techniques implementation that utilizes message bus is CORBA. CORBA: is the acronym for Common Object Request Broker Architecture, OMG's open, vendor-independent architecture and infrastructure that computer applications use to work together over networks [9]. CORBA applications are composed of objects. CORBA creates a distributed objects infrastructure which makes activating and accessing remote objects transparent [71]. Legacy application is wrapped in code with CORBA interfaces and opened up to clients on the network. Systems wrapped with CORBA can be integrated under CORBA architecture. Table 2.6.2 presented in page 49 depicts driving and restraining forces of adopting CORBA as an integration technique implementation. Figure 2.21 presented in page Figure 2.23 presented in page 49 depicts another view of adopting message CORBA pitfalls.

– Message Broker: The message broker can expose different interfaces to the collaborating applications, and it can translate messages between these interfaces [33]. Message broker does not enforce a common interface on the applications, thus handles shortages of Message Bus. Figure 2.24 presented in page 50 depicts Message Broker architecture. Subscribing options are the same as mentioned in Message Bus architecture. Commercially, Message Broker is referred to as Message Oriented Middleware (MOM). One commercial implementation of MOM is Microsoft BizTalk Server [3]. Software Architectures that support Message Broker includes: Broker 'SOA'.

– SOA: Servers can register and deregister with the broker. If a server fails, it will be automatically (after a timeout) unregistered by the bro-

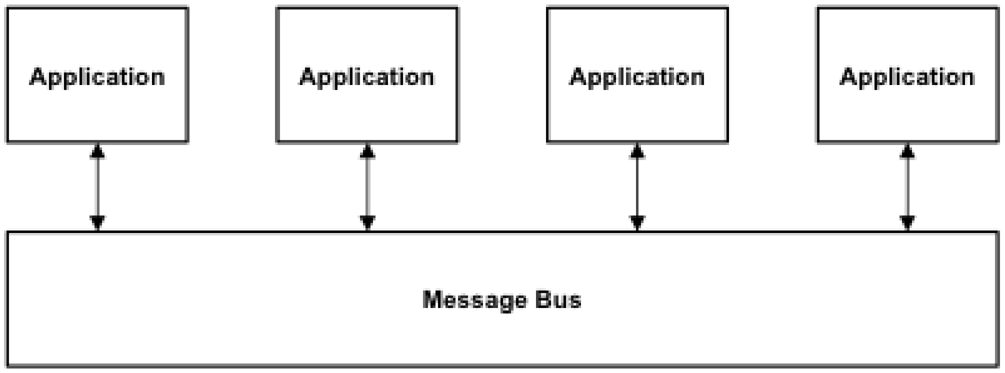

Figure 2.22: Message Bus Architecture

Table 2.5: Adopting Message CORBA Pitfalls

Driving Forces	Restraining Forces
Interoperable networked applications	Different semantics in data sources
Reduced development time	Semantic translations
Reduced maintenance costs	Lack of industry standard definitions
	Lack of CORBA/DCOM product support
	Perceived CORBA/DCOM complexity
	Effect on operational systems of data requests
	Merges and acquisitions

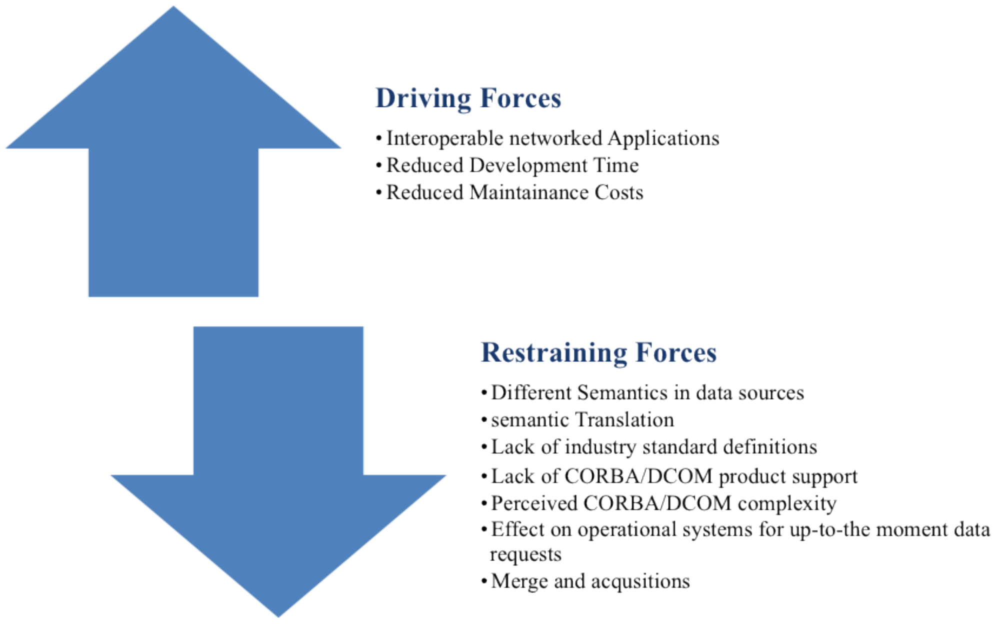
Driving Forces
• Interoperable networked Applications
• Reduced Development Time
• Reduced Maintainance Costs
Restraining Forces
• Different Semantics in data sources
• semantic Translation
• Lack of industry standard definitions
• Lack of CORBA/DCOM product support
• Perceived CORBA/DCOM complexity
• Effect on operational systems for up-to-the moment data requests
• Merge and acqusitions

Figure 2.23: Adopting Message CORBA Pitfalls

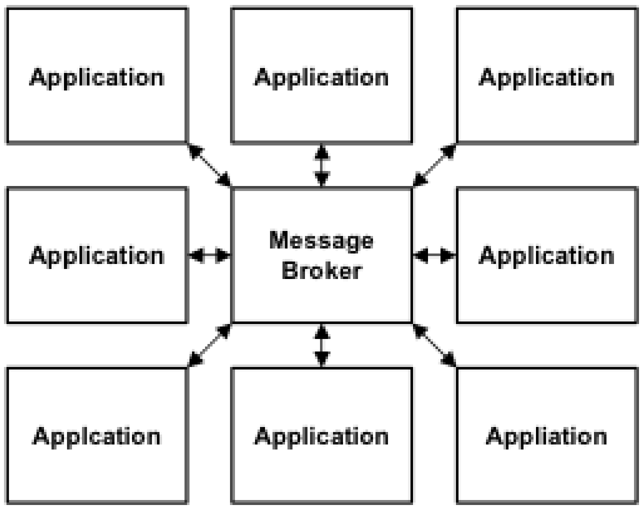

Figure 2.24: Message Broker Architecture

ker. Broker handles user requests by providing user capability to find different service providers, pick the most suitable one, and utilizes its service [64]. The client requests a specific service. It formats its request in a specific format and sends it to its broker. The broker then selects the most suitable server to process the request. When the link between the client and the server is set up, they may start communicating directly, freeing the broker. Figure 2.25 presented in page 51 depicts broker architecture mechanism of action. CORBA was attempted to satisfy Broker architecture requirements, unfortunately CORBA itself has shortages that inhibited it from being the standard integration solution. SOA is the most suitable architecture to integrate organizations on process level. SOA integration is the strongest integration provided to organizations [65]. With this importance for SOA, CORBA limitations were addressed among software vendors and industry standards creators. Web services were presented as the industry solution to overcome CORBA limitations. Table 2.6.2 presented in page 51 depicts driving and restraining forces of adopting Web services as an integration solution. Figure 2.26 presented in page 52 depicts another view of adopting Web services pitfalls. With those driving forces mentioned for adopting Web services, and with the fact that SOA being the only software architecture capable on providing a process level integration, combining both Web services and SOA is almost a must. Most software capabilities will be provided and consumed as services. When the service is abstracted from the implementation it is possible to consider various alternative options for delivery and collaboration models [96].

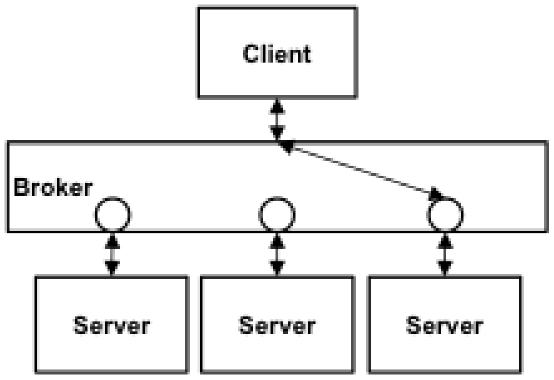

Figure 2.25: Broker Architecture

Table 2.6: Adopting Web services Pitfalls

Driving Forces	Restraining Forces
Interoperable networked applications	Different semantics in data sources
Emerging industry wide standards	Effect on operational systems and data requests
Support of Web services in products	
Easier exchange of data	
Reduced brittleness using tags	
Reduced development time	
Availability of external services	
Availability of training and tools	
Mergers and acquisitions	
Semantic translation	

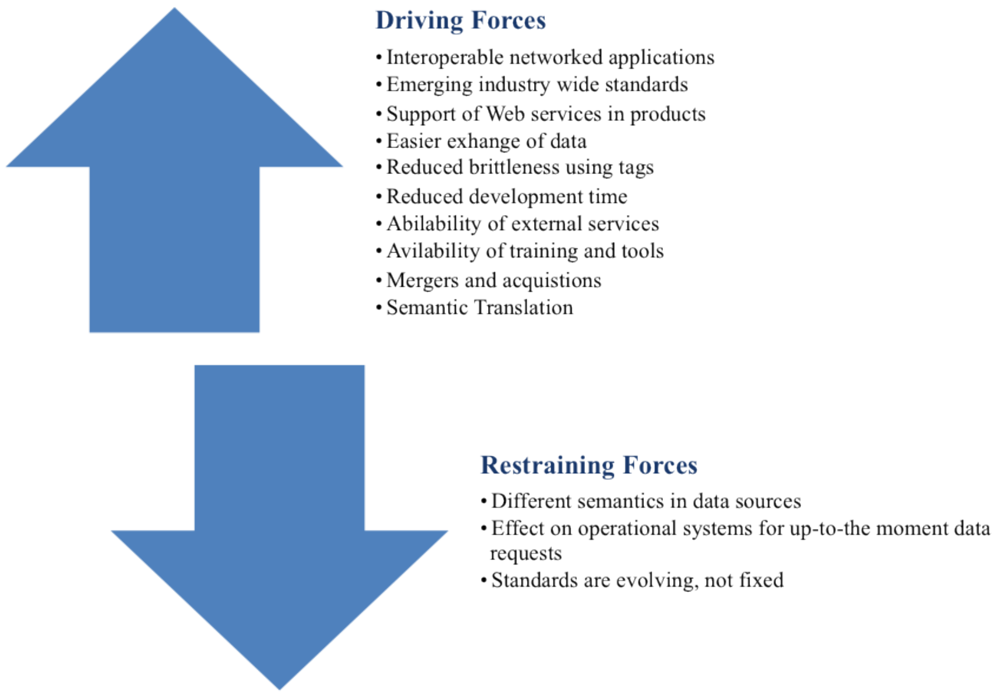
Driving Forces
• Interoperable networked applications
• Emerging industry wide standards
• Support of Web services in products
• Easier exhange of data
• Reduced brittleness using tags
• Reduced development time
• Abilability of external services
• Avilability of training and tools
• Mergers and acquistions
• Semantic Translation
Restraining Forces
• Different semantics in data sources
• Effect on operational systems for up-to-the moment data requests
• Standards are evolving, not fixed

Figure 2.26: Adopting Web services Pitfalls

2.7 SOA Selected Topics

2.7.1 Relationship Between EA and SOA

SOA can be applied to different levels of scope: from the enterprise level to individual project level. SOA is integral to successful EA, and for some organizations it renews the value of EA. A service portfolio is one of the primary linkages of SOA to EA as enterprise architects play a major role in advancing the use and awareness across the enterprise of the service portfolio. The service portfolio is an enterprise asset, and enterprise architects must play a governance role in the use of services, vertically and horizontally in the enterprise. An architectural review board that includes enterprise architects will govern the service portfolio as an enterprise asset. EA is an architectural discipline that merges strategic business and IT objectives with opportunities for change and governs the resulting change initiatives. SOA represents a strategic business and IT objective that EA helps govern across multiple project instances. EA uses SOA principles and assets (e.g., service portfolio and SOA reference models) to integrate with business architecture, information architecture, application architectures, and infrastructure architectures. EA has an enterprise-wide focus and addresses both SOA and non-SOA aspects of the enterprise.

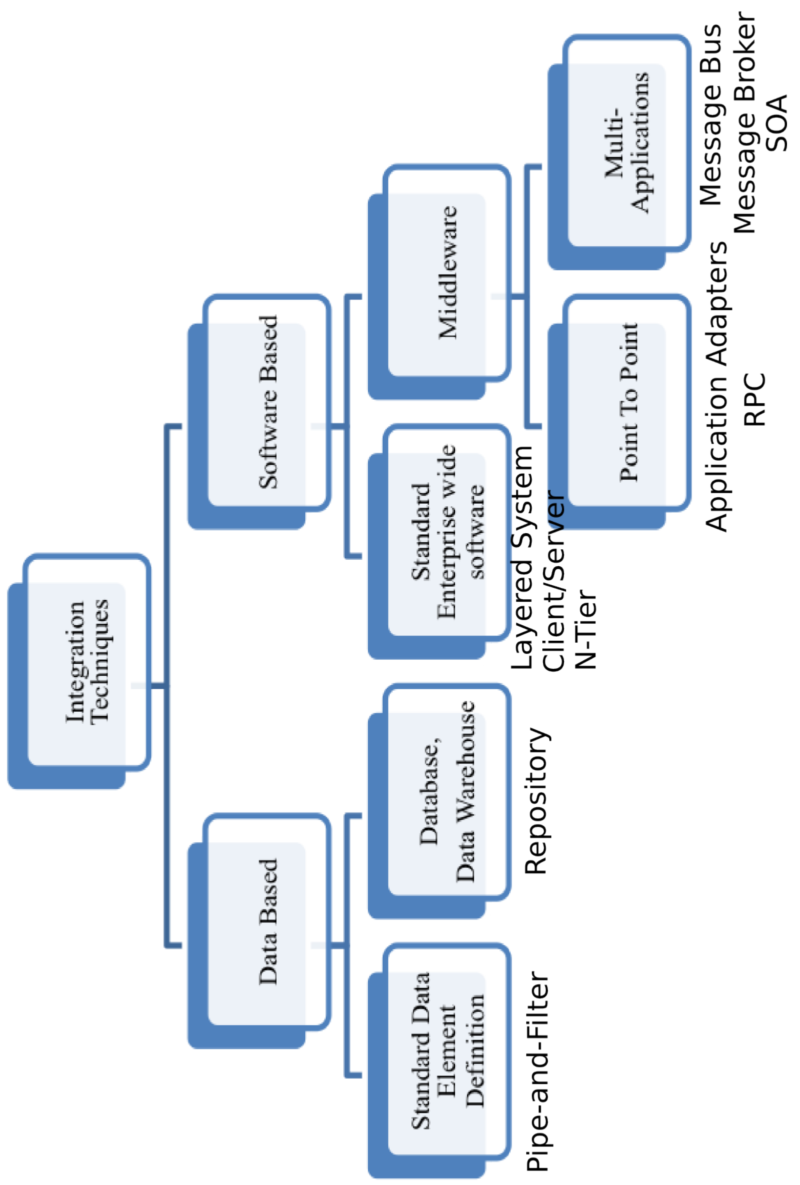

Figure 2.27: Integration Techniques and Software Architectures

2.7.2 Services Identification Methods

Services are optimally identified using three complementary techniques that provide a balance between tactical imperatives and strategic vision:

- Goal service modeling looks at business opportunities, strategy, and business goals to both confirm and validate that candidate services have been identified, which fulfill goals and enable the business strategy.
- Domain decomposition focuses on business process modeling, rules, information, and potential variability of services.
- Asset analysis addresses the reality that businesses have accumulated legacy systems and applications that must be integrated, enhanced, or leveraged. This bottom-up approach looks at the existing application portfolio and other assets that can be used in identifying candidates for service exposure. In contrast, goal service modeling combines the top-down (domain decomposition) and bottom-up (asset analysis) approaches and pulls them together into alignment.

2.7.3 CBA

Component Based Architecture (CBA) is a software architecture that is based on components. A component is a software object, meant to interact with other components, encapsulating certain functionality or a set of functionality. A component has a clearly defined interface and conforms to a prescribed behavior common to all components within an architecture [220]. A component model is probably used for the developing and executing of components. This model defines a framework, which defines structural requirements for connection- and composition options as well as behavior-oriented requirements for collaboration options to the components. Beyond that a component model provides an infrastructure which implements frequently used mechanism like persistence, message-exchange, security and versioning. The idea is to build exchangeable software units through clearly defined interfaces. Different manufactures offer platforms like DCOM, JavaBeans, Enterprise JavaBeans, and CORBA. There is no clear dividing line between SOA and Component Based Architecture. In principle SOA is the enhancement of Components: The individual services are single components, which can be linked to gain new business logic, new services or a new component. The big difference is the connection between and the possibilities of offering single services for third parties. For example, EJBs (especially Session Beans) can be designed to offer its business methods like services in a context free way. These services of this EJB can be used by other EJBs or clients. In a big company (or a coalition of parties gaining access to each others EJB's) single departments could offer their services (in the shape of the business methods of the EJBs) for other departments, so that the same effect could be achieved as with services supporting Simple Object Access Protocol (SOAP): The business methods of the EJBs represent the activities one department offers. Other departments could use these services for their belongings and perhaps use them in a way the business method wasn't built for. So, this usage of Enterprise JavaBeans could be seen as service oriented too.

2.7.4 Agile/OOAD and SOA

Many of the current system development methods focus on making IT more effective, cheaper, or faster, whereas SOA methods focus on making the business and IT more effective and faster. SOA methods provide prescriptive software engineering guidance by addressing the following:

- They provide guidance on how to identify and develop reusable business services that can be reconfigured to provide new business capability or repurposed to serve different business processes or market opportunities.
- They focus on how to reduce, using services, the lifetime cost of the application.
- They reduce system development activities, allowing for an accelerated system development process using services.
- They allow engineer applications to be built for change.

To identify and build reusable, reconfigurable, and flexible services as business assets, you must change existing methods to accommodate the identification, specification, and realization of five primary constructs:

- Business processes
- Services
- Components
- Information
- Rules/policies (and their flows)

A business service catalog will be the result, which will grow over time, project by project. Business services should be reused across applications and channels supporting vertical and horizontal business processes. SOA methods provide guidance on how to identify, specify, and realize reusable business services.

2.8 Summary

EA is the framework of business, IT, information, and software architectures. Software architecture design is an important step within system design because software architectures should satisfy non-functional as it satisfies functional system requirements. Non-functional requirements like integration, interoperability, and agility can be satisfied by architectural design, if considered. Software architecture satisfaction to integration; as one of the non-functional requirements, is used in this chapter as an indicator to the software architecture of non-functional requirements satisfaction.

The most common Software architectures are mapped to presented integration techniques; highlighting architectures driving and restraining forces, and its satisfaction level to system non-functional requirements. Figure 2.27 presented in page 53 summarizes the integration techniques and different software architectures presented as an enabler of each technique. Each one of the presented software architecture has its driving and restraining forces which presents strengths and weaknesses of each one. Web services based SOA achieved the maximum driving forces and satisfied almost all non-functional requirements addressed in the comparative study.

2.9 Review Questions

1. What are Enterprise Architecture Dimensions?
2. Discuss the importance of SW Architecture.
3. What are the advantages of Distributed SW Approaches?
4. List, and describe in brief the most common SW architecture patterns.
5. What are the main SW architectures integration techniques?
6. What are the advantages and disadvantages of each of the following techniques in SW integration:
   (a) Standard Data Element Definition
   (b) Middleware Integration Technique
   (c) Standard Enterprise Wide Software
   (d) Middleware
7. What is the relationship between EA and SOA?
8. What is CBA?

2.10 Exercises and Labs

In Labs, you shall be getting familiar with Java programming language, as it is the programming language we will be using to build, deploy, and consume different online Webservices and SOA systems presented in this book.