SOA UNIT 3

I. INTRODUCTION

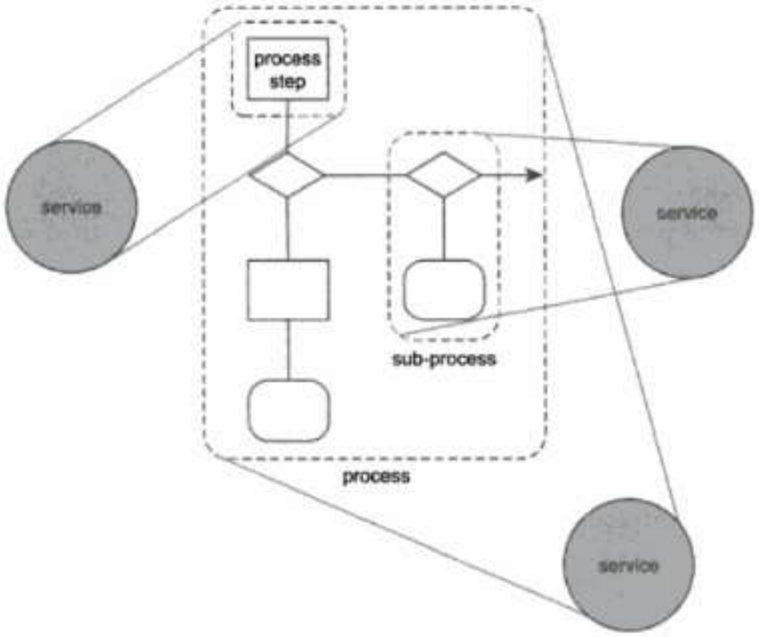
- ❖ SOA is an architectural style that supports service-orientation. Service-orientation means separating things into independent and logical units.
- ❖ SOA is a style of design that guides all aspects of creating and using business services throughout their life cycle.
- ❖ It allows different applications to exchange data and participate in business processes.

SOA represents a model in which automation logic is decomposed into smaller, units of logic. These units comprise a larger piece of business automation logic.
SOA encourages individual units of logic to exist autonomously.
Units of logic are required to conform to a set of principles that allow them to evolve independently, while maintaining a sufficient amount of commonality and standardization. Within SOA, these units of logic are known as *services*.

A)How services encapsulate logic

Service logic can include the logic provided by other services.
As shown in Figure 1, when building an automation solution consisting of services, each service can encapsulate a task performed by an individual step or a sub-process comprised of a set of steps. A service can encapsulate the entire process logic.

Figure 1. Services can encapsulate varying amounts of logic

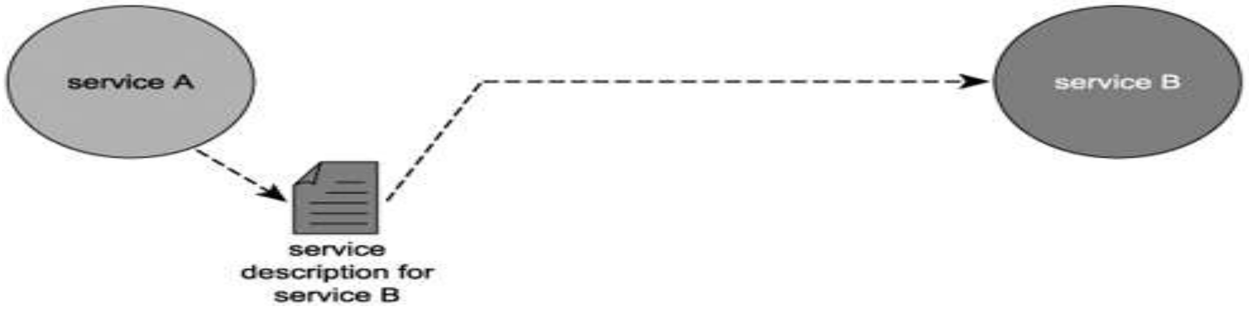


For services to use the logic they encapsulate, they can participate in the execution of business activities. They must form relationships with those that want to use them.

B)How services relate

Within SOA, services can be used by other services or other programs.
The relationship between services is based on an understanding for services to interact, they must be aware of each other. This awareness is achieved through the use of *service descriptions*.
A service description establishes the name of the service and the data expected and returned by the service.
The manner in which services use service descriptions results in a relationship classified as *loosely coupled*. For example, Figure.2 illustrates that service A is aware of service B because service A is in possession of service B's service description.

Figure 2. Because service A has access to service B's service description, it has all of the information it needs to communicate with service B.

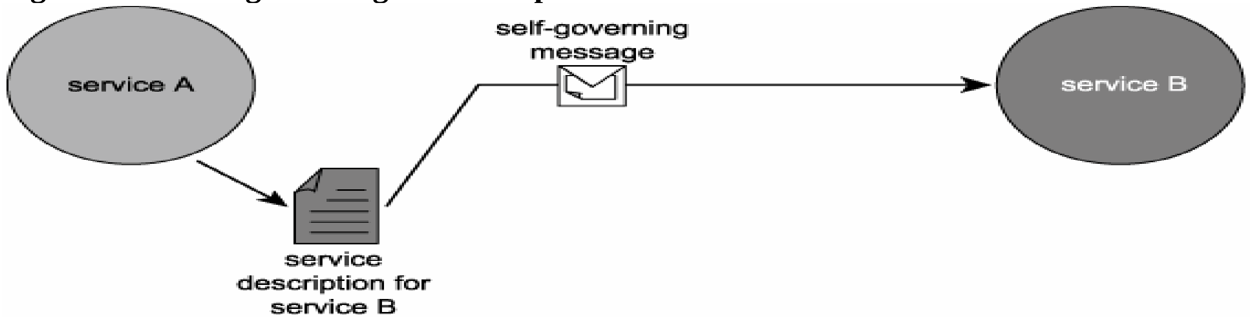


For services to interact and accomplish something meaningful, they must exchange information. A communications framework capable of preserving their loosely coupled relationship is required. One such framework is *messaging*.

C)How services communicate

After a service sends a message on its way, it loses control of what happens to the message thereafter. So messages are required to exist as "independent units of communication". The messages, like services, should be autonomous. The messages can be outfitted with intelligence to self-govern their parts of the processing logic (Figure.3).

Figure.3. A message existing as an independent unit of communication

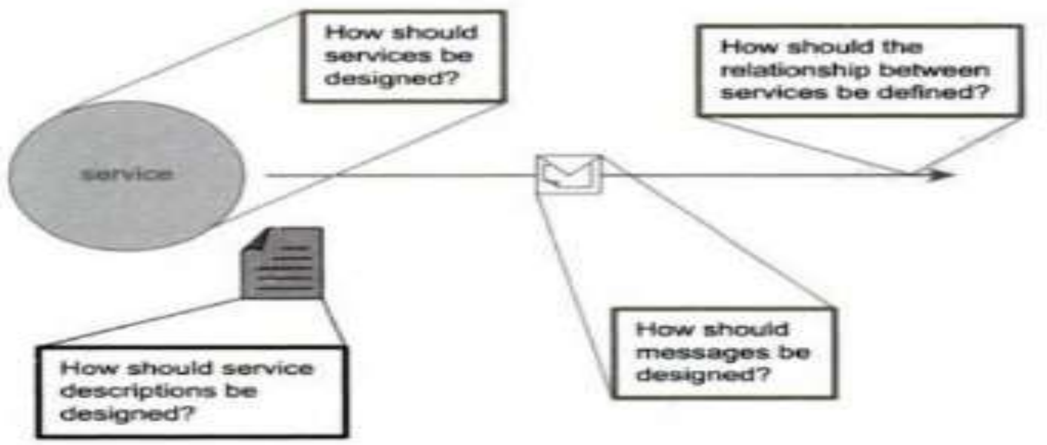


Services that provide service descriptions and communicate via messages form a basic architecture.

D)How services are designed

Like object-orientation, service-orientation has a design approach which introduces accepted principles that govern the positioning and design of architectural components (Figure.4).

Figure 4. Service-orientation principles address design issues


How should services be designed?
How should the relationship between services be defined?
service
How should messages be designed?
How should service descriptions be designed?

The application of service-orientation principles to processing logic results in standardized service-oriented processing logic.

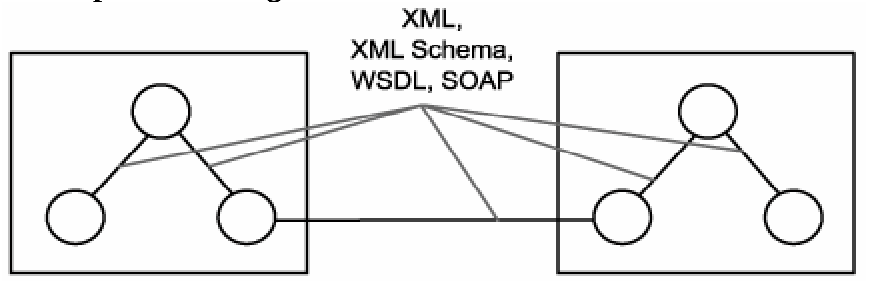
When a solution is comprised of units of service-oriented processing logic, it refer to as a *service-oriented solution*.
The key aspects of these principles:
Loose coupling: Services maintain a relationship that minimizes dependencies and requires that they retain an awareness of each other.
Service contract: Services adhere to a communications agreement, as defined collectively by one or service descriptions and related documents.
Autonomy: Services have control over the logic they encapsulate.
Abstraction: Services hide logic from the outside world.
Reusability: Logic is divided into services with the intention of promoting reuse.
Composability: Collections of services can be coordinated and assembled to form composite services.
Statelessness: Services minimize retaining information specific to an activity.
Discoverability: Services are designed to be outwardly descriptive so that they can be found and assessed via available discovery mechanisms.
E)How services are built
SOA models existed before the arrival of Web services. No one technology advancement has been suitable in manifesting SOA than Web services.
All vendor platforms support the creation of service-oriented solutions, and SOA support provided is based on the use of Web services.

II. THE CHARACTERISTICS OF SOA

Major software vendors are continually conceiving new Web services specifications and building increasingly powerful XML and Web services support into current technology platforms.
The result is an extended variation of service-oriented architecture which is referred to as *contemporary SOA*. The following are the characteristics:

1. Contemporary SOA is at the core of the service-oriented computing platform.
2. Contemporary SOA increases Quality of Service (QoS).
3. Contemporary SOA is fundamentally autonomous.
4. Contemporary SOA is based on open standards.
5. Contemporary SOA supports vendor diversity.
6. Contemporary SOA fosters inherent interoperability.
7. Contemporary SOA promotes discovery.
8. Contemporary SOA promotes federation.
9. Contemporary SOA promotes architectural composability.
10. Contemporary SOA fosters inherent reusability.
11. Contemporary SOA emphasizes extensibility.
12. Contemporary SOA supports a service-oriented business modeling paradigm.
13. Contemporary SOA implements layers of abstraction.
14. Contemporary SOA promotes loose coupling throughout the enterprise.
15. Contemporary SOA promotes organizational agility.
16. Contemporary SOA is a building block.
17. Contemporary SOA is an evolution.
18. Contemporary SOA is maturing.
19. Contemporary SOA is an achievable ideal.

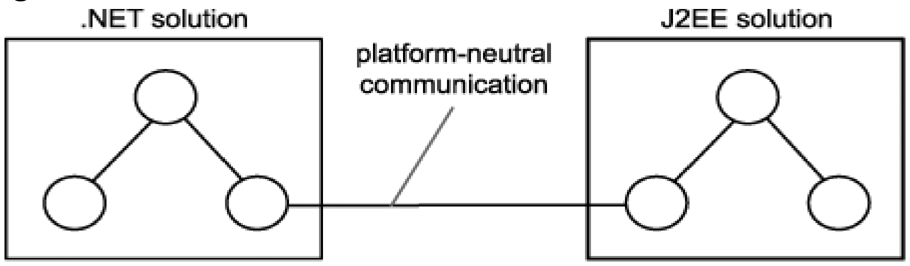
The traditional architectural qualities are "secure," "transactional" and "reliable". These have been grouped into the "Contemporary SOA increases QoS" characteristic.
1. Contemporary SOA is at the core of the service-oriented computing platform

The terms "client-server" or "n-tier," for example, can be used to classify a tool, an administration infrastructure, or application architecture.
Because we positioned contemporary SOA as building upon and extending the primitive SOA model, *Contemporary SOA represents an architecture that promotes service-orientation through the use of Web services*.
2. Contemporary SOA increases QoS
SOA can implement enterprise-level functionality as safely and reliably. This relates to QoS requirements:
The ability for tasks to be carried out in a secure manner, protecting the contents of a message and access to individual services.
Allowing tasks to be carried out reliably so that message delivery or notification of failed delivery can be guaranteed.
Performance requirements ensure that the overhead imposed by Simple Object Access Protocol (SOAP) message and XML content processing does not prevent the execution of a task.
Transactional capabilities protect the integrity of business tasks with a guarantee that should the task fail, exception logic is executed.
Contemporary SOA is determined to fill the QoS gaps of the primitive SOA model.
3. Contemporary SOA is fundamentally autonomous
The service-orientation principle of autonomy requires that individual services be as independent and self-contained.
This is realized through message-level autonomy where messages passed between services are intelligence-heavy that they can control the processing by recipient services.
SOA expands this principle by promoting the concept of autonomy throughout solution environments and the enterprise.
Applications comprised of autonomous services, for example, composite and self-reliant services exercise their self-governance within service-oriented integration environments.
4. Contemporary SOA is based on open standards
The characteristic of Web services is that data exchange is governed by open standards.
After a message is sent from one Web service to another, it travels via a set of protocols that is globally standardized and accepted.
The message is standardized in format and in how it represents its payload.
The use of SOAP, Web Services Definition Language (WSDL), XML, and XML Schema allow for messages to be fully self-contained and support the agreement to communicate, services requires knowledge of each other's service descriptions.
The use of an open, standardized messaging model eliminates the need for service logic to share type systems and supports the loosely coupled paradigm.
Contemporary SOAs fully influence and reinforce open, vendor-neutral communications framework (Figure.1).
Figure 1. Standard open technologies are used within and outside of solution boundaries.


5. Contemporary SOA supports vendor diversity
The open communications framework has implications for bridging the heterogeneity within corporations.

For example, as long as it supports the creation of standard Web services, it can be used to create a non-proprietary service interface layer, opening up interoperability opportunities with other, service-capable applications (Figure.2).
This has changed the integration architectures, which can encapsulate legacy logic through service adapters, and leverage middleware advancements based on Web services.
Organizations can certainly continue building solutions with existing development tools and server products.
This option is made possible by the open technology provided by the Web services framework and is made attainable through the standardization and principles introduced by SOA.

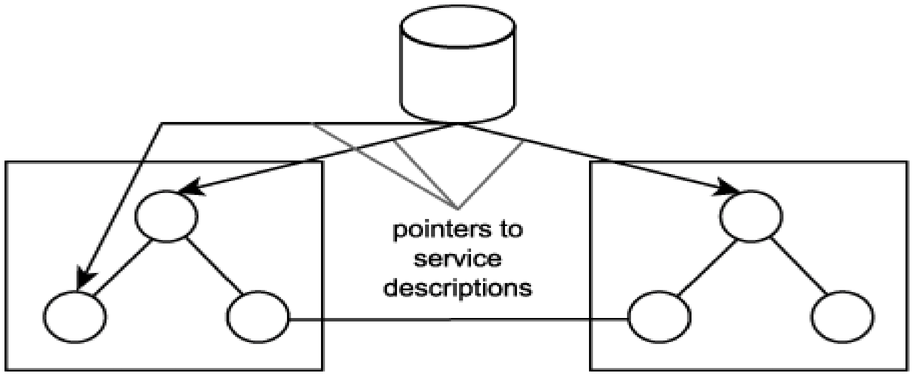
Figure 2. Disparate technology platforms do not prevent service-oriented solutions from interoperating.



6. Contemporary SOA promotes discovery

Even though the first generation of Web services standards included Universal Description Discovery and Integration (UDDI), early implementations used service registries as part of their environments.
When utilized within distributed architectures, Web services were employed to facilitate point-to-point solutions.
SOA supports and encourages the advertisement and discovery of services throughout the enterprise and beyond.
A SOA will rely on form of service registry or directory to manage service descriptions (Fig. 3).

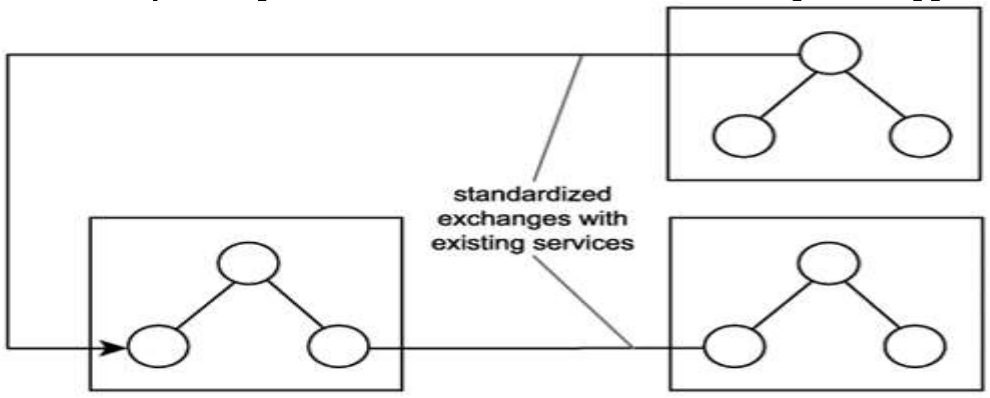
Figure 3. Registries enable a mechanism for the discovery of services



7. Contemporary SOA fosters intrinsic interoperability

Leveraging and supporting the required usage of open standards, a vendor diverse environment, and the availability of a discovery mechanism, is the concept of intrinsic interoperability. Regardless of whether an application has immediate integration requirements, design principles can be applied to outfit services with characteristics that promote interoperability.
When building an SOA application from the ground up, services with intrinsic interoperability become integration endpoints (Figure 4).
When properly standardized, this leads to service-oriented integration architectures wherein solutions achieve a level of intrinsic interoperability.
Fostering this characteristic can alleviate the cost and effort of fulfilling future cross-application integration requirements.

Figure 4. Intrinsically interoperable services enable unforeseen integration opportunities.



8. Contemporary SOA promotes federation

Establishing SOA within an enterprise does not require that you replace what you already have.

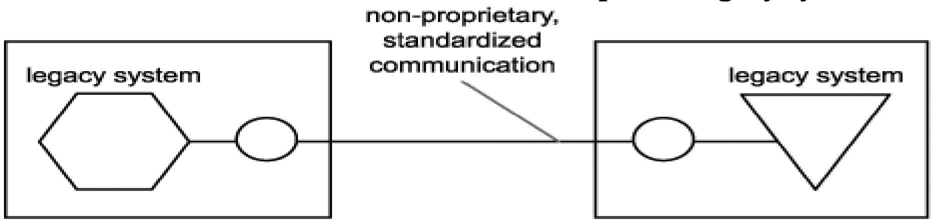
The attractive aspect of this architecture is its ability to introduce unity across previously non-federated environments.

While Web services enable federation, SOA promotes this cause by establishing and standardizing the ability to encapsulate legacy and non-legacy application logic and by exposing it via a common, open, and standardized communications framework.

The incorporation of SOA with previous platforms can lead to a variety of hybrid solutions.

The key aspect is that the communication channels achieved by this form of service-oriented integration are all uniform and standardized (Figure.5).

Figure 5. Services enable standardized federation of disparate legacy systems



9. Contemporary SOA promotes architectural composability

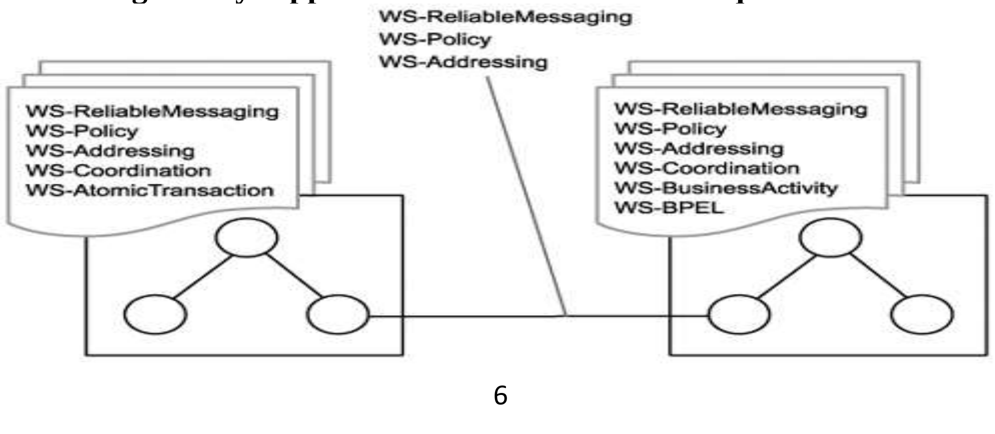
Composability is a deep-rooted characteristic of SOA that can be realized on different levels.

For example, by fostering the development and evolution of composable services, SOA supports the automation of flexible and highly adaptive business processes.

As the offering of WS-* extensions supported by a given vendor platform grows, the flexibility to compose allows you to continue building solutions that implement the features you need (Figure 6).

The WS-* platform allows for the creation of streamlined and optimized service-oriented architectures, applications, services, and messages.

Figure 6. Different solutions can be composed of different extensions and can continue to interoperate as long as they support the common extensions required

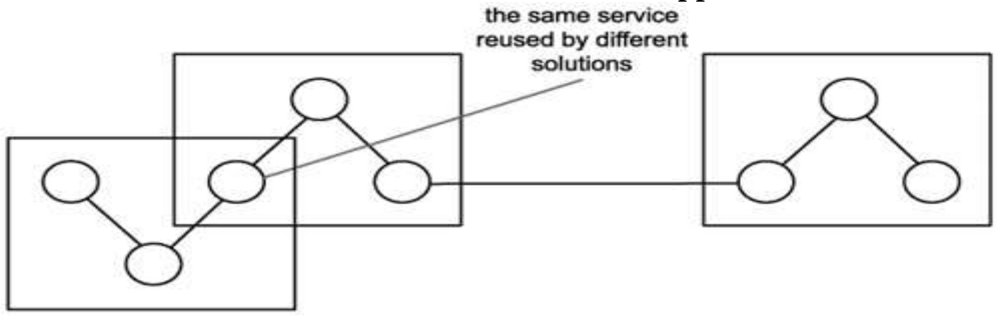


This characteristic represents composable services, as well as the extensions that comprise individual SOA implementations.
Services exist as independent units of logic. A business process can be broken down into a series of services, each responsible for executing a portion of the process.
A broader example of composability is represented by the second-generation Web services framework that is evolving out of the release of the numerous WS-* specifications.
The modular nature of these specifications allows an SOA to be composed of the functional building blocks it requires.
The second-generation Web services specifications are being designed specifically to leverage the SOAP messaging model.
Individual specifications consist of modular extensions that provide specific features.

10. Contemporary SOA fosters inherent reusability

SOA establishes an environment that promotes reuse on many levels. For example, services designed according to service-orientation principles are encouraged to promote reuse, if no immediate reuse requirements exist.
Collections of services that form service compositions can be reused by larger compositions.
The emphasis placed by SOA on the creation of services that are agnostic to the business processes and the automation solutions that utilize them leads to an environment in which reuse is realized as a side benefit to delivering services for a given project.
Thus, inherent reuse can be fostered when building service-oriented solutions (Figure.7).

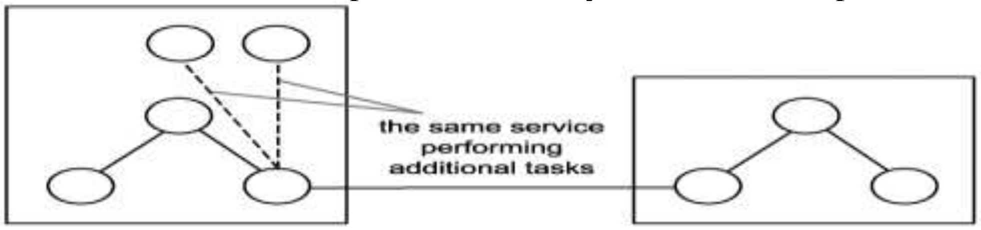
Figure 7. Inherent reuse accommodates unforeseen reuse opportunities



11. Contemporary SOA emphasizes extensibility

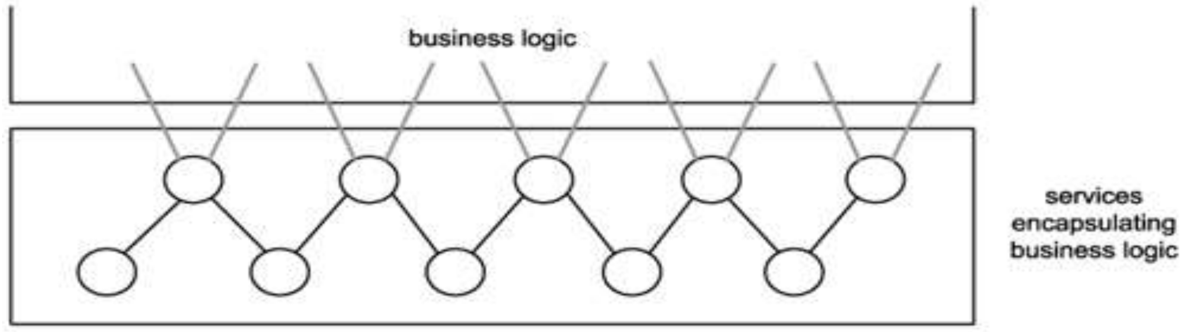
When expressing encapsulated functionality through a service description, SOA encourages you to think beyond immediate, point-to-point communication requirements.
When service logic is properly partitioned via an appropriate level of interface granularity, the scope of functionality offered by a service can be extended without breaking the established interface (Figure.8).

Figure 8. Extensible services can expand functionality with minimal impact

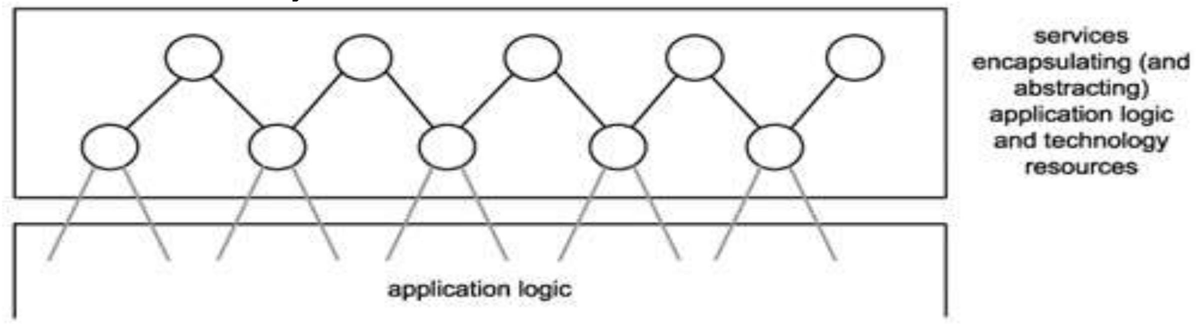


Extending entire solutions can be accomplished by adding services or by merging with other service-oriented applications ("adds services").
Because the loosely coupled relationship fostered among all services minimizes inter-service dependencies, extending logic can be achieved with significantly less impact.
Contemporary SOA represents an ***open, extensible, federated, composable*** *architecture that promotes service-orientation and is comprised of* ***autonomous, QoS-capable, vendor diverse, interoperable, discoverable****, and* ***reusable*** *services, implemented as Web services.*

12. Contemporary SOA supports a service-oriented business modeling paradigm
Business processes can be represented and expressed through services.
Partitioning business logic into services that can be composed has implications as to how business processes can be modeled (Figure.9).
Analysts can leverage these features by incorporating an extent of service-orientation into business processes for implementation through SOAs.
Figure 9. A collection (layer) of services encapsulating business process logic



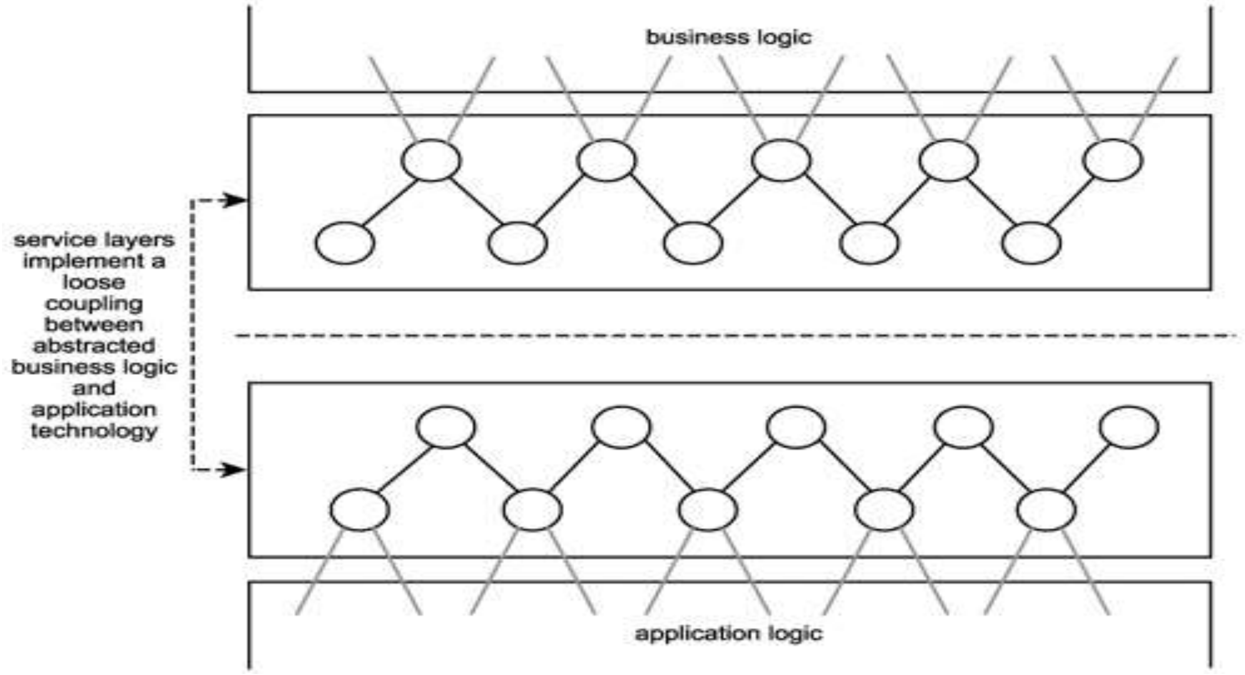
Services can be designed to express business logic. BPM models, entity models, and other forms of business intelligence can be represented through the coordinated composition of business-centric services. This is an area of SOA that is not accepted.
13. Contemporary SOA implements layers of abstraction
One of the characteristics that tend to evolve through the application of service-oriented design principles is that of abstraction.
SOAs can introduce layers of abstraction by positioning services as the access points to a variety of resources and processing logic.
When applied through proper design, abstraction can be targeted at business and application logic.
For example, by establishing a layer of endpoints that represent entire solutions and technology platforms, all of the proprietary details associated with these environments disappear (Fig .10).
Figure 10. Application logic created with proprietary technology can be abstracted through a dedicated service layer


services encapsulating (and abstracting) application logic and technology resources
application logic

It is the mutual abstraction of business and technology that supports the service-oriented business modeling paradigm and establishes the loosely coupled enterprise model.
14. Contemporary SOA promotes loose coupling throughout the enterprise
A core benefit to building a technical architecture with loosely coupled services is the resulting independence of service logic.
Services require an awareness of each other, allowing them to evolve independently.
Within an organization where service-orientation principles are applied to business modeling and technical design, the concept of loose coupling is amplified.
By implementing standardized service abstraction layers, a loosely coupled relationship can be achieved between the business and application technology domains of an enterprise (Fig. 11).
Each end requires an awareness of the other, allowing each domain to evolve independently.

The result is an environment that can better accommodate business and technology-related change a quality known as *organizational agility*.

Figure 11. Through the implementation of service layers that abstract business and application logic, the loose coupling paradigm can be applied to the enterprise as a whole


business logic
service layers implement a loose coupling between abstracted business logic and application technology
application logic

15. Contemporary SOA promotes organizational agility

Whether the result of an internal reorganization, a corporate merger, a change in an organization's business scope, or the replacement of an established technology platform, an organization's ability to accommodate change determines the efficiency with which it can respond to unplanned events.

Change in an organization's business logic can impact the application technology that automates it. Change in an organization's application technology infrastructure can impact the business logic automated by this technology.

By leveraging service business representation, service abstraction, and the loose coupling between business and application logic provided through the use of service layers, SOA offers the to increase organizational agility (Figure.12).

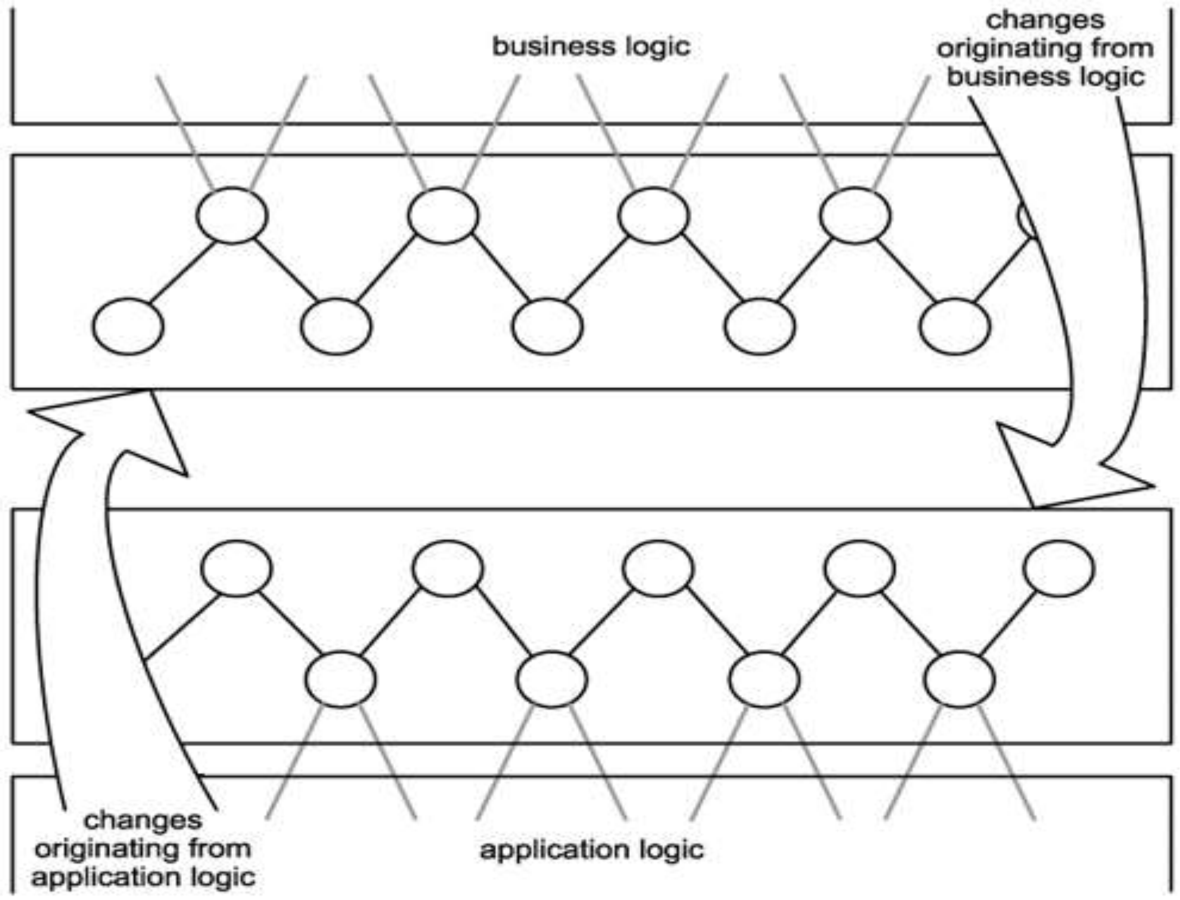
Benefits realized through the standardization of SOA contribute to minimizing dependencies and increasing overall responsiveness to change:

the intrinsic interoperability that can be built into services and the open communications framework established across integration architectures that enable interoperability between disparate platforms.

Change imposed on any of these environments is facilitated for the same reasons a loosely coupled state between services representing either ends of the communication channel.

Organizational agility is a benefit that can be realized with contemporary SOA.

Figure 12. A loosely coupled relationship between business and application technology allows each end to efficiently respond to changes in the other.


business logic
changes originating from business logic
changes originating from application logic
application logic

16. Contemporary SOA is a building block

Service-oriented application architecture will be one of several within an organization committed to SOA as the standard architectural platform.

Organizations standardizing on SOA work toward the Service-Oriented Enterprise (SOE), where all business processes are composed of and exist as services, logically and physically.

When viewed in the context of SOE, the functional boundary of an SOA represents a part of this future-state environment, either as a standalone unit of business automation or as a service encapsulating some or all of the business automation logic.

In responding to business model-level changes, SOAs can be augmented to change the nature of their automation, or they can be pulled into service-oriented integration architectures that require the participation of multiple applications.

An individual service-oriented application can be represented by and modeled as a single service. There are no limits to the scope of service encapsulation. An SOA consists of services within services; a solution based on SOA is one of many services within an SOE.

SOA can establish an abstraction of business logic and technology that introduce changes to business process modeling and technical architecture, resulting in a loose coupling between these models.

These changes foster service-orientation in support of a service-oriented enterprise.

17. Contemporary SOA is an evolution

SOA defines an architecture that is related to but distinct from its predecessors. It differs from traditional client-server and distributed environments in that it is influenced by the concepts and principles associated with service-orientation and Web services.

It is similar to previous platforms in that it preserves the successful characteristics of its predecessors and builds upon them with distinct design patterns and a new technology set.

For example, SOA supports and promotes reuse, the componentization and distribution of application logic.
These and other established design principles that are commonplace in traditional distributed environments are a part of SOA.
18. Contemporary SOA is maturing
Even though SOA is being positioned as the next standard application computing platform, this transition is not complete.
Web services are used to implement application functionality; the support for enterprise-level computing is not fully available.
Standards organizations and major software vendors have produced many specifications to address a variety of supplementary extensions.
The next generation of development tools and application servers promises to support these new technologies.
When SOA platforms and tools reach maturity, the utilization of Web services can be extended to support the creation of enterprise SOA solutions, making the ideal of a service-oriented enterprise attainable.
19. Contemporary SOA is an achievable ideal
A standardized enterprise-wide adoption of SOA is a state to which many organizations would like to fast-forward.
The reality is that the process of transitioning to this state demands an enormous amount of effort, discipline, and, depending on the size of the organization, a good amount of time.
Every technical environment will undergo changes during such a migration, and various parts of SOA will be phased in at different stages and to varying extents.
This will result in hybrid architectures, consisting of distributed environments that are part legacy and part service-oriented.
The evolving state of the technology set that is emerging to realize enterprise-level SOAs.
As companies adopt SOA during this evolution, many will need to retrofit their environments to accommodate changes and innovations as SOA-related specifications, standards, and products continue to mature.
20. Defining SOA
Contemporary SOA represents an open, agile, extensible, federated, composable architecture comprised of autonomous, QoS-capable, vendor diverse, interoperable, discoverable, and reusable services, implemented as Web services.
SOA can establish an abstraction of business logic and technology that may introduce changes to business process modeling and technical architecture, resulting in a loose coupling between these models.
SOA is an evolution of past platforms, preserving successful characteristics of traditional architectures, and bringing with it distinct principles that foster service-orientation in support of a service-oriented enterprise.
SOA is ideally standardized throughout an enterprise, but achieving this state requires a planned transition and the support of a evolving technology set.
SOA is a form of technology architecture that adheres to the principles of service-orientation.
When realized through the Web services technology platform, SOA establishes the to support and promote these principles throughout the business process and automation domains of an enterprise.
21. Separating concrete characteristics
We can split into two groups' characteristics that represent concrete qualities that can be realized as real extensions of SOA and those that can be categorized as commentary or observations.

The set of concrete characteristics of Contemporary SOA is based on open standards, architecturally composable, and capable of improving QoS.
Contemporary SOA supports, fosters, or promotes:
vendor diversity, intrinsic interoperability, discoverability, federation, inherent reusability, extensibility, service-oriented business modeling, layers of abstraction, enterprise-wide loose coupling and organizational agility.
It is these characteristics that, when realized, provide tangible, measurable benefits.

III.COMMON MISPERCEPTIONS ABOUT SOA

We see the common points of confusion about SOA and the use of the term "service-oriented".

1. "An application that uses Web services is service-oriented."

There is a distinction between SOA as an abstract model and SOA based on Web services and service-orientation. Depending on which abstract model you use, any form of distributed architecture can be classified as being service-oriented.
To realize the benefit of the mainstream variation of SOA, you need to standardize how Web services are positioned and designed, according to service-orientation principles.
A traditional distributed architecture can be called service-oriented as long as the benefits associated with primitive and contemporary SOA are not expected.

2. "SOA is a marketing term used to re-brand Web services."

The term "SOA" has been used excessively for marketing purposes. It has become a high-profile brought on by the rise of Web services.
Contemporary SOAs are being implemented using Web services have led to doubt around the validity of the term.
Some believe that "SOA support" is a re-labeling of "Web services support."
SOA is not an invention of the media or some marketing department. SOA is a legitimate and relatively established technical term.
It represents a distinct architecture based on a set of distinct principles.
Contemporary SOA implies the use of a set of technology used to realize fundamental SOA principles. The technology platform is currently Web services and all that comes with it.

3. "SOA is a marketing term used to re-brand distributed computing with Web services."

Many of the migration paths lay out by product vendors blend traditional distributed computing with Web services, accompanied by advertised "SOA support."
The results certainly can be confusing. The validity of promoted SOA support is questionable at best.
SOA is its own entity. It consists of a set of design principles that are related to, but differ from the distributed computing platforms of the past.

4. "SOA simplifies distributed computing."

The principles behind SOA are relatively simple in nature. Applying these principles in real life can prove to be a complex task.
Though SOA offers benefit, this can be realized by investing in thorough analysis and adherence to the design principles of service-orientation.
SOA implementations require up-front research than solutions created under previous platform paradigms.
This is in part due to the broad Web services technology platform imposed by contemporary SOA.
The quality of simplicity surfaces later, once service-orientation is established and standardized within an IT environment.

When integration requirements emerge, when a sufficient number of composable services exist, and when service-orientation principles are well integrated into an organization, that's when SOA can simplify the delivery of automation requirements.

5. "An application with Web services that uses WS-* extensions is service-oriented."

While the entire second generation of Web services specifications are certainly driving SOA into the IT mainstream, making these extensions part of an architecture does not make it service-oriented.

Regardless of the functionality with which Web services are outfitted, what makes them part of a service-oriented architecture is how the architecture is designed.

Solutions that seriously employ the use of WS-* extensions will be service-oriented.

This is partially because the adoption rate of SOA is anticipated to roughly coincide with the availability of WS-*extension support in development and middleware products.

6. "If you understand Web services you won't have a problem building SOA."

A technical and conceptual knowledge of Web services is certainly helpful. Service-orientation requires a change in how business and application logic are viewed, partitioned, and automated. It requires that Web services be utilized and designed according to specific principles.

Web services are incorporated into existing traditional distributed architectures. There they can be centrally positioned and assigned processing responsibilities, or they can be appended as peripheral application endpoints.

The manner in which Web services are utilized in SOA is different. The emphasis placed on business logic encapsulation and the creation of service abstraction layers will require a blend of technology and business analysis expertise.

It is best to assume that realizing contemporary SOA requires a separate skill set that goes beyond knowledge of Web services technology.

7. "Once you go SOA, everything becomes interoperable."

Many assume that by virtue of building service-oriented solutions, their technical environments will transform into a united, federated enterprise.

Though this ultimate goal is attainable, it requires investment, analysis, and, above all, a high degree of standardization.

By leveraging the open Web services communications framework, service-oriented architectures abstract and hide all that is proprietary about a particular solution, its platform, and its technology.

This establishes a predictable communications medium for all applications exposed via a Web service.

It does not automatically standardize the representation of the information that is exchanged via this medium.

As SOAs become common, there will be good and not so good implementations. A quality SOA requires that individual services conform to common design standards for federation, interoperability, reusability, and other benefits to be fully realized.

SUMMARY:

The misperceptions relate to the use of Web services within distributed Internet architectures being mistaken as contemporary SOA.

Some of the dangerous assumptions about SOA are that service-oriented solutions are simple by nature, easy to build, and automatically interoperable.

IV. COMMON TANGIBLE BENEFITS OF SOA

The benefits focus on tangible returns on investment, related to:

how SOA leads to improvements in automated solution construction how the proliferation of service-orientation ends up benefiting the enterprise as a whole SOA will benefit organizations

in different ways, depending on their respective goals and the manner in which SOA and its supporting cast of products and technologies is applied.
This list of common benefits is generalized and certainly not complete. It is an indication of the architectural platform has to offer.

1. Improved integration (and intrinsic interoperability)

SOA can result in the creation of solutions that consist of inherently interoperable services. Utilizing solutions based on interoperable services is part of Service-Oriented Integration (SOI) and results in SOI architecture.
Because of the vendor-neutral communications framework established by Web services-driven SOAs, there are enterprises to implement highly standardized service descriptions and message structures.
The net result is intrinsic interoperability, which turns a cross-application integration project into less of a custom development effort, and of a modeling exercise.
The cost and effort of cross-application integration is lowered when applications being integrated are SOA-compliant.

2. Inherent reuse

Service-orientation promotes the design of services that are inherently reusable. Designing services to support reuse from the get-go opens the door to increased opportunities for leveraging existing automation logic.
Building service-oriented solutions in such a manner that services fulfill immediate application-level requirements while supporting a degree of reuse by future requestors establishes an environment wherein investments into existing systems can be leveraged and re-leveraged as new solutions are built.
Building services to be inherently reusable results in a moderately increased development effort and requires the use of design standards.
Leveraging reuse within services lowers the cost and effort of building service-oriented solutions.

3. Streamlined architectures and solutions

The concept of composition is another fundamental part of SOA. It is not limited to the assembly of service collections into aggregate services.
The WS-* platform is based in its entirety on the principle of composability.
This aspect of service-oriented architecture can lead to highly optimized automation environments, where the technologies required become part of the architecture.
Realizing this benefit requires adherence to design standards that govern allowable extensions within each application environment.
Benefits of streamlined solutions and architectures include the for reduced processing overhead and reduced skill-set requirements.
The reduced performance requirements refer that SOA extensions are composable and allow application-level architecture to contain extensions relevant to its solution requirements.
Message-based communication in SOAs can increase performance requirements when compared to RPC-style communication within traditional distributed architectures.

4. Leveraging the legacy investment

The industry-wide acceptance of the Web services technology set has spawned a large adapter market, enabling many legacy environments to participate in service-oriented integration architectures.
This allows IT departments to work toward a state of federation, where previously isolated environments can interoperate without requiring the development of expensive and fragile point-to-point integration channels.

Though riddled with risks relating mostly to how legacy back-ends must cope with increased usage volumes, the ability to use what you already have with service-oriented solutions that you are building and in the future is extremely attractive.
The cost and effort of integrating legacy and contemporary solutions is lowered. The need for legacy systems to be replaced is lessened.

5. Establishing standardized XML data representation

SOA is built upon and driven by XML. An adoption of SOA leads to the opportunity to fully leverage the XML data representation platform.
A standardized data representation format can reduce the underlying complexity of all affected application environments. Examples include:
XML documents and accompanying XML Schemas (packaged within SOAP messages) passed between applications or application components fully standardize format and typing of all data communicated.
The result is a predictable and extensible and adaptable communications network.
XML's self-descriptive nature enhances the ability for data to be readily interpreted by architects, analysts, and developers.
The result is for data within messages to be maintained, traced, and understood.
The standardization level of data representation lays the groundwork for intrinsic interoperability.
By promoting the use of standardized vocabularies, the need to translate discrepancies between how respective applications have interpreted corporate data models is reduced.
Past efforts to standardize XML technologies have resulted in limited success, as XML was either incorporated in an ad-hoc manner or on an "as required" basis.
These approaches inhibited the benefits XML could introduce to an organization. With contemporary SOA, establishing an XML data representation architecture becomes a necessity, providing organizations the opportunity to achieve a broad level of standardization.
The cost and effort of application development is reduced after a proliferation of standardized XML data representation is achieved.

6. Focused investment on communications infrastructure

Because Web services establish a communications framework, SOA can centralize inter-application and intra-application communication as part of standard IT infrastructure. This allows organizations to evolve enterprise-wide infrastructure by investing in a single technology set responsible for communication.
The cost of scaling communications infrastructure is reduced, as one communications technology is required to support the federated part of the enterprise.

7. "Best-of-breed" alternatives

Some of the harshest criticisms laid against IT departments are related to the restrictions imposed by a given technology platform on its ability to fulfill the automation requirements of an organization's business areas.
This can be due to the expense and effort required to realize the requested automation, or it may be the result of limitations inherent within the technology itself.
IT departments are required to push back and limit or even reject requests to alter or expand upon existing automation solutions.
SOA won't solve these problems entirely, but it is expected to increase empowerment of business and IT communities.
A key feature of service-oriented enterprise environments is the support of "best-of-breed" technology.
Because SOA establishes a vendor-neutral communications framework, it frees IT departments from being chained to a single proprietary development and/or middleware platform.

For any given piece of automation that can expose an adequate service interface, you now have a choice as to how you want to build the service that implements it.
The scope of business requirement fulfillment increases, as does the quality of business automation.

8. Organizational agility

Agility is a quality inherent in just about any aspect of the enterprise. A simple algorithm, a software component, a solution, a platform, a process all of these parts contain a measure of agility related to how they are constructed, positioned, and leveraged.
How building blocks such as these can be realized and maintained within existing financial and cultural constraints ultimately determines the agility of the organization as a whole.
Much of service-orientation is based on the assumption that what you build today will evolve over time.
One of the benefits of a well-designed SOA is to protect organizations from the impact of this evolution.
When accommodating change becomes the norm in distributed solution design, qualities such as reuse and interoperability become commonplace.
The predictability of these qualities within the enterprise leads to a reliable level of organizational agility. All of this is attainable through proper design and standardization.
Change can be disruptive, expensive, and damaging to inflexible IT environments. Building automation solutions and supporting infrastructure with the anticipation of change seems to make a great deal of sense.
A standardized technical environment comprised of loosely coupled, composable, and interoperable and reusable services establishes adaptive automation environment that empowers IT departments to adjust to change.
By abstracting business logic and technology into specialized service layers, SOA can establish a loosely coupled relationship between these two enterprise domains.
This allows each domain to evolve independently and adapt to changes imposed by the other, as required.
Regardless of what parts of service-oriented environments are leveraged, the increased agility with which IT can respond to business process or technology-related changes is significant.
The cost and effort to respond and adapt to business or technology-related change is reduced.

SUMMARY:

When assessing the return on investment for an SOA there are several concrete benefits that can be taken into account.
Many of the benefits promised by SOA do not manifest themselves until the use of service-orientation principles becomes established within an enterprise.

V. COMMON PITFALLS OF ADOPTING SOA

Considering the extent to which organizations need to shift technology and mindset to adopt SOA, some will build bad service-oriented architectures. Following are descriptions of some of the common mistakes.

1. Building service-oriented architectures like traditional distributed architectures

One obstacle organizations face in achieving SOA is building traditional distributed architectures under the pretense that they are building contemporary SOA.
The danger with this scenario is that an organization can go far in terms of integrating the Web services technology set before realizing that they've been heading down the wrong path.
Examples of the problems include:

- Proliferation of RPC-style service descriptions (leading to increased volumes of fine-grained message exchanges).

- Inhibiting the adoption of features provided by WS-* specifications.
- Improper partitioning of functional boundaries within services.
- Creation of non-composable (or semi-composable) services.
- Further entrenchment of synchronous communication patterns.
- Creation of hybrid or non-standardized services.

Understanding the fundamental differences between SOA and architectures is the key to avoiding this situation.

2. Not standardizing SOA

In larger organizations where various IT projects occur concurrently, the need for custom standards is paramount.

If different development projects result in the creation of differently designed applications, future integration efforts will be expensive and fragile.

The ability for SOA to achieve federation across disparate environments has been well promoted.

It does not happen by purchasing the latest upgrades to a vendor's development tools and server software.

SOA, like any other architecture, requires the creation and enforcement of design standards for its benefits to be truly realized.

For example, if one project builds a service-oriented solution in isolation from others, key aspects of its solution will not be in alignment with the neighboring applications it may be required to interoperate with one day.

This can lead to many problems, including:

Incompatible data representation results in disparate schemas representing the same types of information.

Service descriptions with irregular interface characteristics and semantics.

Support for different extensions or extensions being implemented in different ways.

SOA promotes a development environment that abstracts back-end processing so that it can execute and evolve independently within each application.

Standardization is required to ensure consistency in design and interaction of services that encapsulate this back-end logic.

3. Not creating a transition plan

The chances of a successful migration will be diminished without the use of a comprehensive transition plan.

Because the extent to which service endpoints are positioned within an enterprise can lead to a redefinition of an IT environment's infrastructure, the consequences of a poorly executed migration can be significant.

Transition plans allow you to coordinate a controlled phasing in of service-orientation and SOA characteristics so that the migration can be planned on a technological, architectural, and organizational level.

Examples of typical areas covered by a transition plan include:

An impact analysis that predicts the extent of change SOA will impose on existing resources, processes, custom standards, and technology.

The definition of transition architectures, where those environments identified as candidates for a migration to SOA evolve through a series of planned hybrid stages.

A speculative analysis takes into account the future growth of Web services and supporting technologies.

Detailed design changes to centralized logic (such as a new security model).

Creating a transition plan avoids the many problems associated with an ad-hoc adoption of SOA.

Each plan will be unique to an organization's requirements, constraints, and goals.

4. Not starting with an XML foundation architecture

In the world of contemporary SOA, everything begins with Web services. That statement is not entirely true. In the world of contemporary SOA, begins with XML. It is the standard from which multiple supplementary standards have evolved to form data representation architecture.
It is this core set of standards that has fueled the creation of the many Web services specifications that are driving SOA.
So much attention is given to how data is transported between services that the manner in which this same data is structured and validated behind service lines is neglected.
This oversight can lead to an improper implementation of a persistent XML data representation layer within SOAs. The results can affect the quality of data processing.
For example, the same data may be unnecessarily validated multiple times, or redundant data processing can inadvertently be performed before and after a service transmits or receives a message.
Standardizing the manner in which core XML technologies are used to represent, validate, and process corporate data as it travels throughout application environments lays the groundwork for a robust, optimized, and interoperable SOA.

5. Not understanding SOA performance requirements

When starting out small, it is easy to build service-oriented solutions that function and respond as expected. As the scope increases and functionality is added, the volume of message-based communication predictably grows.
This is when unprepared environments can begin experiencing significant processing latency.
Because contemporary SOA introduces layers of data processing, it is subject to the associated performance overhead imposed by these layers.
Contemporary SOA's reliance on Web services deepens its dependence on XML data representation, which can magnify XML processing-related performance challenges.
For example, Web services security measures, such as encryption and digital signing, add new layers of processing to the senders and recipients of messages.
Critical to building a successful service-oriented solution understands the performance requirements of your solution and the performance limitations of your infrastructure ahead of time.
Testing the message processing capabilities of your environments prior to investing in Web services.
Stress-testing the vendor supplied processors (for XML, XSLT, SOAP, etc.) you are planning to use.
Exploring alternative processors, accelerators, or other types of supporting technology.
For example, the XML-binary Optimized Packaging (XOP) and SOAP Message Transmission Optimization Mechanism (MTOM) specifications developed by the W3C.
Performance is one of the reasons coarse-grained service interfaces and asynchronous messaging are emphasized when building Web services. These and other design measures can be implemented to avoid processing bottlenecks.

6. Not understanding Web services security

The extent to which Web services technology grows within a given environment is related to the comfort level developers and architects have with the overall technology framework.
Once it does expand it is easy to simply continue building on simplistic message exchanges, which usually rely on Secure Sockets Layer (SSL) encryption to implement a familiar measure of security.
While SSL can address many immediate security concerns, it is not the technology of choice for SOA.
When services begin to take on greater amounts of processing responsibility, the need for message-level security begins to arise.

The WS-Security framework establishes an accepted security model supported by a family of specifications that end up infiltrating service-oriented application and enterprise architectures on many levels.
One of the design issues you may face when WS-Security hits your world is the centralized security.
With this approach, the architecture abstracts a large portion of security logic and rules into a separate, central layer that is then relied upon by service-oriented applications.
Even if your vendor platform does not yet provide adequate support for WS-Security, and if your current SSL-based implementation is meeting immediate requirements, it is advisable to pay close attention to the changes that are ahead.
Proceeding without taking WS-Security into account will inevitably lead to expensive retrofitting and redevelopment. This impact is amplified if you decide to implement a centralized security model, which would become an extension of IT infrastructure.
Acquiring a sound knowledge of the framework will allow you to adjust your current architecture and application designs to better accommodate upcoming changes.

7. Not keeping in touch with product platforms and standards development

IT professionals used to working within the confines of a single development platform have become accustomed to focusing on industry trends as they apply to the product set they are currently working with.
For example, .NET developers are not too concerned with what's happening in the Java world, and vice versa.
A transition to SOA opens up the arena of products and platforms that IT departments can choose from to build and/or host custom-developed application logic.
While the tendency will be there to continue with what you know best, the option to look elsewhere is ever-present.
This is the result of establishing a vendor-neutral communications framework that allows solutions based on disparate technologies to become fully interoperable.
Another factor that can weigh in when comparing products is how product vendors relate to the WS-* specification development process that is currently underway.
As different vendor alliances continue to produce competing extensions, how your vendors position themselves amidst this landscape will become increasingly important, once you begin to identify the extensions required to implement and execute key parts of your solutions' application logic.
In the meantime, specific aspects to look out for include:
which specifications vendors choose to support
the involvement vendors demonstrate with the standards development process itself
which other organizations vendors choose to ally themselves with roadmaps published by vendors explaining how their product or platform will support upcoming specifications and standards.

SUMMARY:

Many of the pitfalls relate to a limited understanding of what SOA is and what is required to fully incorporate and standardize service-orientation.
A transition plan is the best weapon against the obstacles that tend to face organizations when migrating toward SOA.
Staying in touch with commercial and standards-related developments is an important part of keeping an existing SOA aligned with future developments.

VI. SOAP and web services:

Simple Object Access Protocol (SOAP) specification was designed to unify proprietary RPC communication.
The idea was for parameter data transmitted between components to be serialized into XML, transported, and then deserialized back into its native format.
Corporations and software vendors began to see an increasingly large for advancing the state of eBusiness technology by building upon the proprietary-free Internet communications framework.
This led to the idea of creating a pure, Web-based, distributed technology one that could leverage the concept of a standardized communications framework to bridge the enormous disparity that existed between and within organizations. This concept was called Web services.
The important part of a Web service is its public interface. It is a central piece of information that assigns the service an identity and enables its invocation.
The initiative in support of Web services was the Web Service Description Language (WSDL).
Web services required an Internet-friendly and XML-compliant communications format that could establish a standardized messaging framework.
Although alternatives, such as XML-RPC, were considered, SOAP won out as the industry favorite and remains the fore messaging standard for use with Web services.
Completing the first generation of the Web services standards family was the UDDI specification.
UDDI provides Web services to be registered in a central location, from where they can be discovered by service requestors.
Unlike WSDL and SOAP, UDDI has not attained industry-wide acceptance, and remains an optional extension to SOA.
Custom Web services were developed to accommodate a variety of specialized business requirements, and a third-party marketplace emerged promoting various utility services for sale or lease.
Existing messaging platforms, such as Messaging-Oriented Middleware (MOM) products, incorporated Web services to support SOAP in addition to other message protocols.
Some organizations were able to immediately incorporate Web services to facilitate B2B data exchange requirements as an alternative to Electronic Data Interchange (EDI).

VII. APPLICATION ARCHITECTURE, ENTERPRISE ARCHITECTURE AND SOA

Application architecture: Application architecture is to an application development team what a blueprint is to a team of construction workers.
Different organizations document different levels of application architecture. Some keep it high-level, providing abstract physical and logical representations of the technical blueprint.
Others include detail, such as common data models, communication flow diagrams, application-wide security requirements, and aspects of infrastructure.
For example, an organization that houses .NET and J2EE solutions would have separate application architecture specifications for each.
A key part of any application-level architecture is that it reflects immediate solution requirements, as well as long-term, strategic IT goals.
Enterprise architecture: When numerous, disparate application architectures co-exist and even integrate, the demands on the underlying hosting platforms can be complex.
It is common for a master specification to be created, providing a high-level overview of all forms of heterogeneity that exist within an enterprise.
The relationship between an urban plan and the blueprint of a building are comparable to that of enterprise and application architecture specifications.

Enterprise architectures contain a long-term vision of how the organization plans to evolve its technology and environments. For example, the goal of phasing out an outdated technology platform may be established in this specification.

Service Oriented Architecture

- Spans enterprise and application architecture domains
- Benefits of SOA are realized when applied across multiple solution environments
- An SOA can refer to application architecture or the approach used to standardize technical architecture across the enterprise.
- Because SOA is a composable architecture (individual application-level architectures comprised of different extensions and technologies), organization can have more than one SOA.

VIII. COMPARE SOA TO CLIENT-SERVER ARCHITECTURE

a)SOA:

SOA is a radical departure from client-server architecture. Current SOAs employ technologies used to build client-server applications.

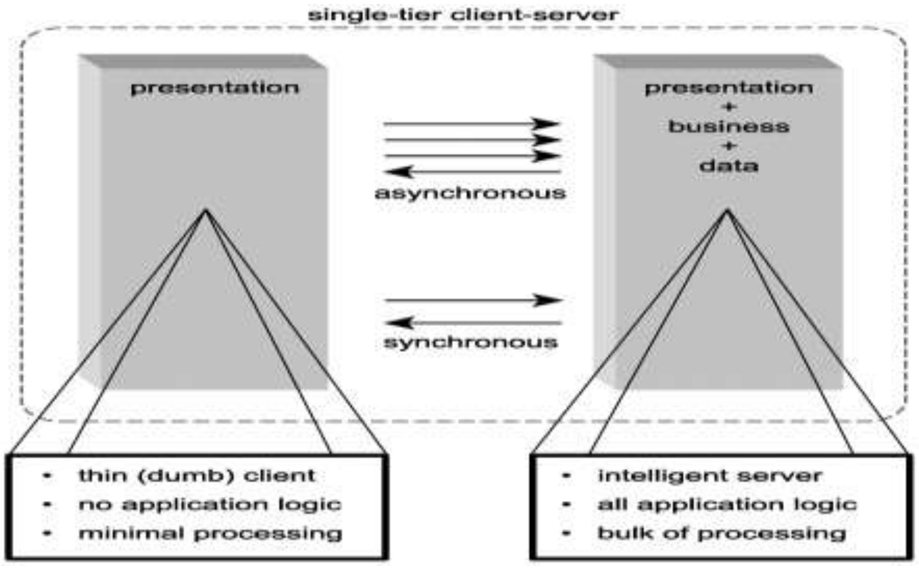
SOAs introduce complexity that sharply contrasts the simplicity of two-tier client-server architecture.

Any environment in which one piece of software requests or receives information from another can be referred to as "client-server".

Every variation of application architecture that ever existed (including SOA) has an element of client-server interaction in it.

Client-server architecture: The environments, in which bulky mainframe back-ends served thin clients, are considered an implementation of the single-tier client-server architecture (Figure 1).

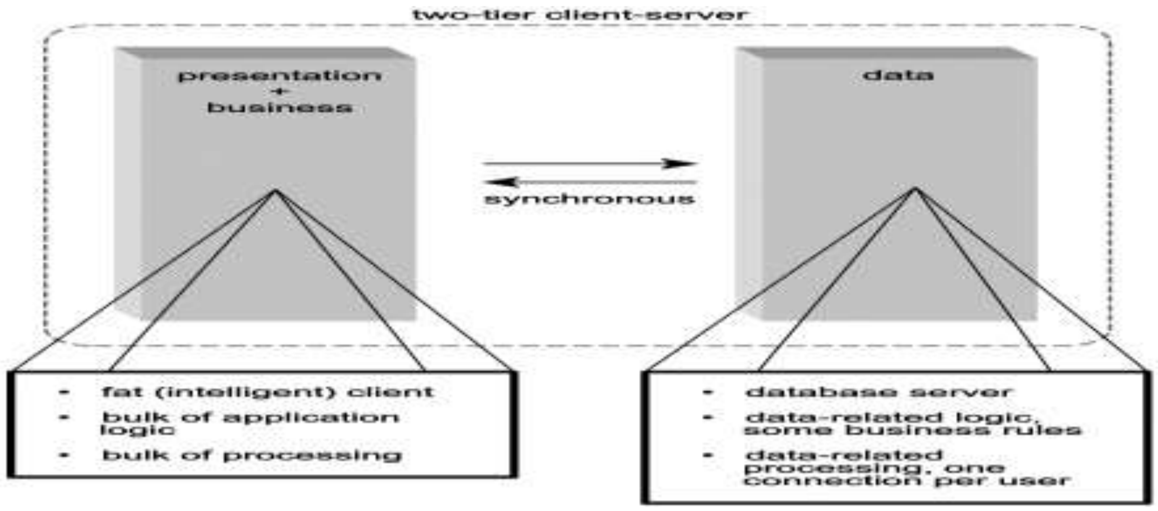
Figure 1. A typical Single-tier Client Server Architecture



- Mainframes supported synchronous and asynchronous communication.
- Client-side software performed the bulk of the processing, including all presentation-related and data access logic Figure.2.
- Asynchronous allow the server to continuously receive characters from the terminal in response to individual key-strokes. Upon certain conditions would the server respond?

The reign of the mainframe as the computing platform began to decline when a two-tier variation of the client-server design emerged in the 80s. This new approach introduced the delegating logic and processing duties onto individual workstations, resulting in the fat client.

Figure 2. A typical Two-tier Client Server Architecture


two-tier client-server
presentation + business
data
synchronous
- fat (intelligent) client
- bulk of application logic
- bulk of processing
- database server
- data-related logic, some business rules
- data-related processing, one connection per user

Supported by the innovation of the GUI, two-tier client-server was considered and dominates the IT world for years during the early 90s.

The configuration of this architecture consisted of multiple fat clients, each with its own connection to a database on a central server.

One or more servers facilitated these clients by hosting scalable RDBMSs.

b)Client Server Application Logic

- Client-server environments place the majority of application logic into the client software. This results in a monolithic executable that controls the user experience, as well as the back-end resources.
- One exception is the distribution of business rules. A trend was to embed and maintain business rules relating to data within stored procedures and triggers on the database.
- This abstracted a set of business logic from the client and simplified data access programming. Overall, though, the client ran the show.

SOA Application Logic

The presentation layer within contemporary SOA can vary. Software capable of exchanging SOAP messages according to required service contracts can be classified as a service requestor. While it is expected for requestors to be services, presentation layer designs are open and specific to a solution's requirements.

Within the server environment, options exist as to where application logic can reside and how it can be distributed.

These options do not preclude the use of database triggers or stored procedures.

Service-oriented design principles dictate the partitioning of processing logic into autonomous units.

This facilitates design qualities, such as service statelessness, interoperability, composability and reusability.

It is common within SOA for these units of processing logic to be solution-agnostic. This supports the goal of promoting reuse and loose coupling across application boundaries.

c)Client-Server Application Processing

Because client-server application logic resides in the client component, the client workstation is responsible for the bulk of the processing.

The 80/20 ratio is used as a rule of thumb, with the database server performing twenty percent of the work.

A two-tier client-server solution with a large user-base requires that each client establish its own database connection.

Communication is synchronous, and these connections are persistent (they are generated upon user login and kept active until the user exits the application).
Proprietary database connections are expensive, and the resource demands overwhelm database servers, imposing processing latency on all users.
Client-side executables are stateful and consume a chunk of PC memory. User workstations are required to run client programs so that all available resources can be offered to the application.
SOA Application Processing

- Processing with SOA is distributed. Each service has functional boundaries and resource requirements.
- In modeling a technical SOA, you have many choices as to how you can position and deploy services.
- Enterprise solutions consist of multiple servers, each hosting with sets of Web services and supporting middleware.
- There is no fixed processing ratio for SOA.

Communication between service and requestor can be synchronous or asynchronous.
This flexibility allows processing to be streamlined, when asynchronous message patterns are utilized.
By placing a large amount of intelligence into the messages, options for achieving message-level context management are provided.
This promotes the stateless and autonomous nature of services and alleviates processing by reducing the need for runtime caching of state information.
d)Client-Server vs SOA Application Technology
Client-server applications promoted the use of 4GL programming languages, such as Visual Basic and PowerBuilder.
Traditional 3GL languages, such as C++, are used, for solutions that had rigid performance requirements.
On the back-end, database vendors, such as Oracle, Informix, IBM, Sybase, and Microsoft, provided robust RDBMSs that could manage multiple connections, while providing flexible data storage and data management features.
In addition to the set of Web technologies (HTML, CSS, HTTP, etc.) contemporary SOA brings with it the requirement that XML data representation architecture be established, along with a SOAP messaging framework, and a service architecture comprised of the ever-expanding Web services platform.
e)Client-Server Vs SOA Application Security

- Client-server architecture that is centralized at the Server level is security.
- Databases manage user accounts and groups and to assign these to individual parts of the physical data model.

Security can be controlled within the client executable, when it relates to business rules that dictate the execution of application logic.
Operating system-level security can be incorporated to achieve a single sign-on, where application clearance is derived from the user's OS login account information.
architects envy the simplicity of client-server security. Corporate data is protected via a single point of authentication, establishing a single connection between client and server. In the distributed world of SOA, this is not possible.
f)Client-Server Application Vs SOA Administration
The client-server era ended was the increasingly large maintenance costs associated with the distribution and maintenance of application logic across user workstations.
Because each client housed the application code, each update to the application required a redistribution of the client software to all workstations.

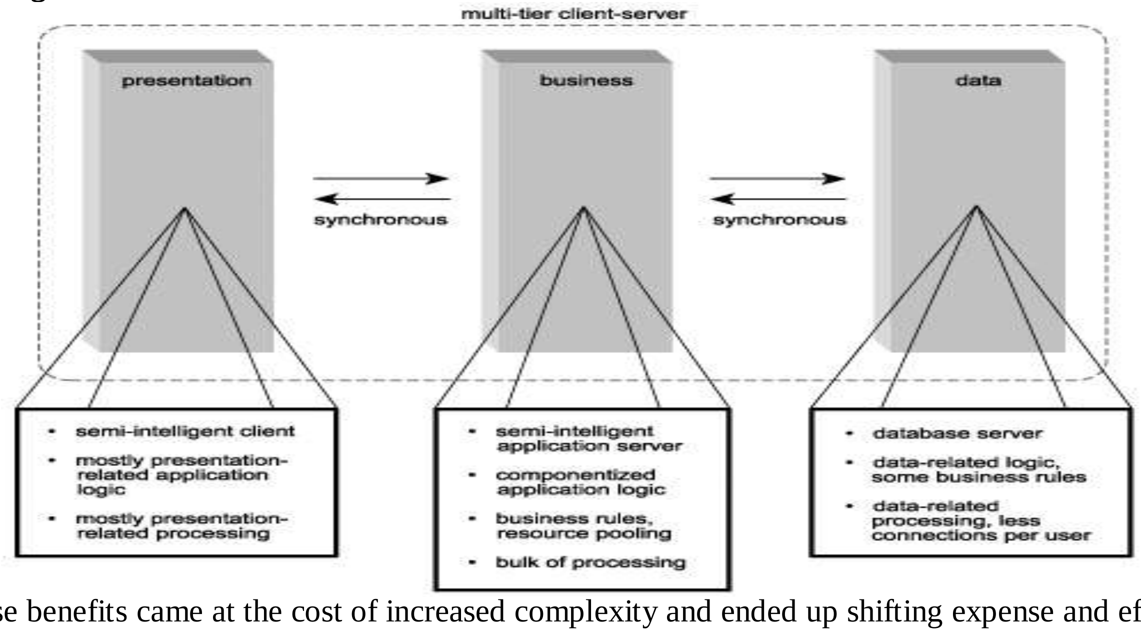
In larger environments, this resulted in a highly burdensome administration process.
Maintenance issues spanned client and server ends. Client workstations were subject to environment-specific problems because different workstations could have different software programs installed or may have been purchased from different hardware vendors.
There were increased server-side demands on databases, when a client-server application expanded to a larger user base.
Because service-oriented solutions can have a variety of requestors, they are not immune to client-side maintenance challenges.
While their distributed back-end does accommodate scalability for application and database servers, new administration demands can be introduced.
For example, once SOAs evolve to a state where services are reused and become part of multiple service compositions, the management of server resources and service interfaces can require powerful administration tools, including the use of a private registry.

IX. COMPARE SOA TO DISTRIBUTED ARCHITECTURE

A.SOA vs Traditional Distributed Internet Architecture

In response to the costs and limitations associated with the two-tier client server architecture, the concept of building component-based applications hit the mainstream.
Multi-tier client-server architectures surfaced, breaking up the monolithic client executable into components designed to varying extents of compliance with object-orientation.
Distributing application logic among multiple components (some residing on the client, others on the server) reduced deployment headaches by centralizing a greater amount of the logic on servers.
Server-side components located on dedicated application servers, would share and manage pools of database connections, alleviating the burden of concurrent usage on the database server (Figure.1). A single connection could facilitate multiple users.

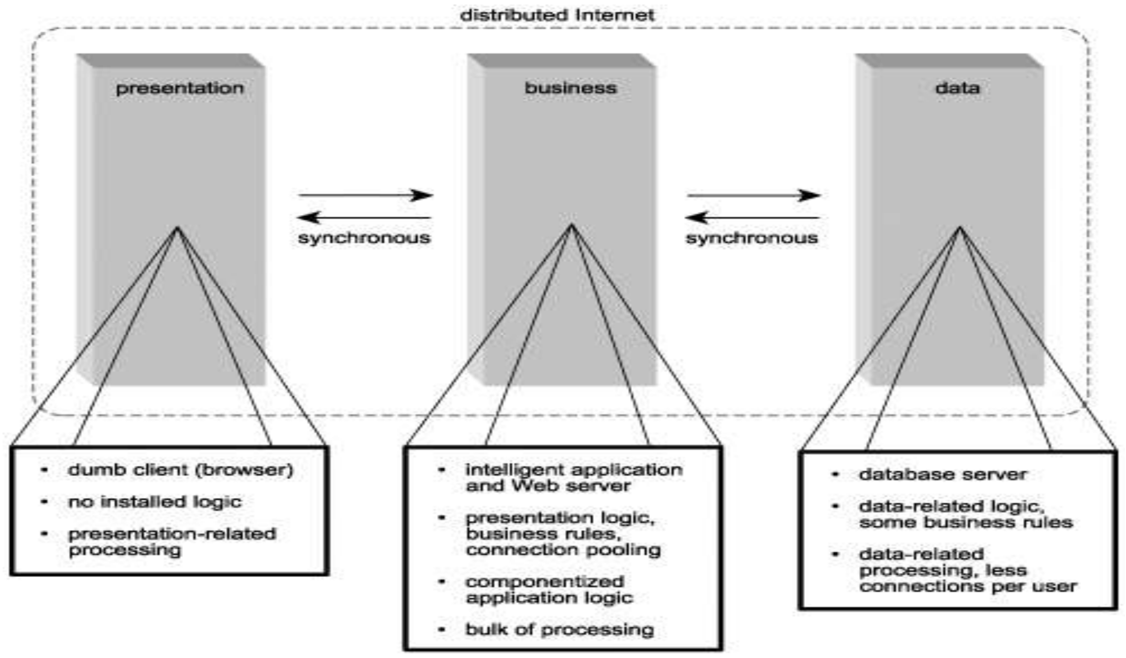
Figure 1. Multi-tier Client Server Architecture


multi-tier client-server
presentation
business
data
synchronous
synchronous
• semi-intelligent client
• mostly presentation-related application logic
• mostly presentation-related processing
• semi-intelligent application server
• componentized application logic
• business rules, resource pooling
• bulk of processing
• database server
• data-related logic, some business rules
• data-related processing, less connections per user

These benefits came at the cost of increased complexity and ended up shifting expense and effort from deployment issues to development and administration processes.
Building components capable of processing multiple, concurrent requests were difficult and problem-ridden than developing a straight-forward executable intended for a single user.
Additionally, replacing client-server database connections was the client-server Remote Procedure Call (RPC) connection.

RPC technologies such as CORBA and DCOM allowed for remote communication between components residing on client workstations and servers.
Issues similar to the client-server architecture problems involving resources and persistent connections emerged.
Adding to this was an increased maintenance effort resulting from the introduction of the middleware layer. For example, application servers and transaction monitors required significant attention in larger environments.
Upon the arrival of the World Wide Web as a viable medium for computing technology in the mid-to-late 90s, the multi-tiered client-server environments began incorporating Internet technology.
Significant was the replacement of the custom software client component with the browser. Not did this change radically alter (and limit) user-interface design, it practically shifted 100% of application logic to the server (Figure. 2).

Figure 2. A typical distributed Internet architecture.


distributed Internet
presentation
business
data
synchronous
synchronous
dumb client (browser)
no installed logic
presentation-related processing
intelligent application and Web server
presentation logic, business rules, connection pooling
componentized application logic
bulk of processing
database server
data-related logic, some business rules
data-related processing, less connections per user

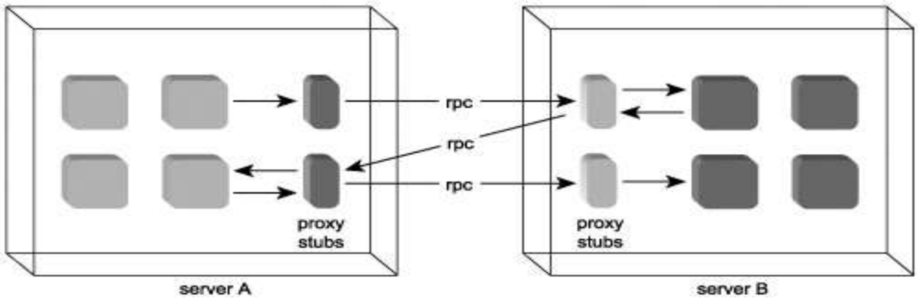
Distributed Internet architecture introduced a new physical tier, the Web server. This resulted in HTTP replacing proprietary RPC protocols used to communicate between the user's workstation and the server.
The role of RPC was limited to enabling communication between remote Web and application servers.
Distributed Internet architectures represented the de facto computing platform for custom developed enterprise solutions.
The commoditization of component-based programming skills and the increasing sophistication of middleware lessened some of the overall complexity.
Although multi-tier client-server is a distinct architecture in its own right, we do not provide a direct comparison between it and SOA.

B.Application logic

Except for some rare applications that embed proprietary extensions in browsers, distributed Internet applications place all of their application logic on the server side.
Even client-side scripts intended to execute in response to events on a Web page are downloaded from the Web server upon the initial HTTP request.
With none of the logic existing on the client workstation, the entire solution is centralized.

The emphasis is on: how application logic should be partitioned where the partitioned units of processing logic should reside how the units of processing logic should interact.
From a physical perspective, service-oriented architecture is very similar to distributed Internet architecture. Provider logic resides on the server end where it is broken down into separate units. The differences lie in the principles used to determine the three design considerations listed.
Traditional distributed applications consist of a series of components that reside on one or application servers.
Components are designed with varying degrees of functional granularity, depending on the tasks they execute, and to what extent they are considered reusable by other tasks or applications. Components residing on the same server communicate via proprietary APIs, as per the public interfaces they expose.
RPC protocols are used to accomplish the same communication across server boundaries. This is made possible through the use of local proxy stubs that represent components in remote locations (Figure.3).

Figure 3. Components rely on proxy stubs for remote communication



At design time, the expected interaction components will have with others is taken into account so much so that actual references to other physical components can be embedded within the programming code.
This level of design-time dependence is a form of tight-coupling. It is efficient in that little processing is wasted in trying to locate a required component at runtime.
The embedded coupling leads to a tightly bound component network that, once implemented, is not altered.
Contemporary SOAs employ and rely on components. The entire modeling approach takes into consideration the creation of services that encapsulate some or all of these components.
These services are designed according to service-orientation principles and are strategically positioned to expose specific sets of functionality.
While this functionality can be provided by components, it can originate from legacy systems and other sources, such as adapters interfacing with packaged software products, or databases.
The purpose of wrapping functionality within a service is to expose that functionality via an open, standardized interface irrespective of the technology used to implement the underlying logic.
The standardized interface supports the open communications framework that sits at the core of SOA.
The use of Web services establishes a loosely coupled environment that runs contrary to many traditional distributed application designs.
When properly designed, loosely coupled services support a composition model, allowing individual services to participate in aggregate assemblies.
This introduces continual opportunities for reuse and extensibility.
Another shift related to the design and behavior of distributed application logic is in how services exchange information.

While traditional components provide methods that, once invoked, send and receive parameter data, Web services communicate with SOAP messages.
Even though SOAP supports RPC-style message structures, the majority of *service-oriented* Web service designs rely on document-style messages.
Messages are structured to be as self-sufficient. Through the use of SOAP headers, message contents can be accompanied by a range of meta information, processing instructions, and policy rules.
In comparison to data exchange in the component world, the messaging framework used by SOA is sophisticated, bulkier, and tends to result in less individual transmissions.
Finally, although reuse is emphasized in distributed design approaches, SOA fosters reuse and cross-application interoperability on a deep level by promoting the creation of solution-agnostic services.

C.Application processing

Regardless of platform, components represent the lion's share of application logic and are responsible for the processing.
Because the technology used for inter-component communication differs from the technology used to accomplish inter-service communication, so do the processing requirements.
Distributed Internet architecture promotes the use of proprietary communication protocols, such as DCOM and vendor implementations of CORBA for remote data exchange.
They can support the creation of stateful and stateless components that interact with synchronous data exchanges.
SOA relies on message-based communication. This involves the serialization, transmission, and deserialization of SOAP messages containing XML document payloads.
Processing steps can involve the conversion of relational data into an XML-compliant structure, the validation of the XML document prior and subsequent to transmission, and the parsing of the document and extraction of the data by the recipient.
Although advancements, such as the use of enterprise parsers and hardware accelerators are on-going, rank RPC communication as being noticeably faster than SOAP.
Because a network of SOAP servers can effectively replace RPC-style communication channels within service-oriented application environments, the incurred processing overhead becomes a significant design issue.
Document and message modeling conventions and the strategic placement of validation logic are important factors that shape the transport layer of service-oriented architecture.
This messaging framework promotes the creation of autonomous services that support a wide range of message exchange patterns.
Though synchronous communication is fully supported, asynchronous patterns are encouraged, as they provide further opportunities to optimize processing by minimizing communication. Supporting the statelessness of services are various context management options that can be employed, including the use of WS-* specifications, such as WS-Coordination and WS-BPEL, as well as custom solutions.

D.Technology

Initial architectures consisted of components, server-side scripts, and raw Web technologies, such as HTML and HTTP.
Improvements in middleware allowed for increased processing power and transaction control.
The emergence of XML introduced sophisticated data representation that gave substance to content transmitted via Internet protocols.
The subsequent availability of Web services allowed distributed Internet applications to cross proprietary platform boundaries.

Because many current distributed applications use XML and Web services, there may be difference between the technology behind these solutions and those based on SOA.
A contemporary SOA will build upon XML data representation and the Web services technology platform.
Beyond a core set of Internet technologies and the use of components, there is no governance of the technology used by traditional Internet applications.
Thus XML and Web services are optional for distributed Internet architecture but not for contemporary SOA.

E.Security

When application logic is strewn across multiple physical boundaries, implementing fundamental security measures such as authentication and authorization becomes difficult.
In a two-tiered client-server environment, an exclusive server-side connection facilitates the identification of users and the safe transportation of corporate data.
When the exclusivity of that connection is removed, and when data is required to travel across different physical layers, new approaches to security are needed.
To ensure the safe transportation of information and the recognition of user credentials, while preserving the original security context, traditional security architectures incorporate approaches such as delegation and impersonation.
Encryption is added to the otherwise wide open HTTP protocol to allow data to be protected during transmission beyond the Web server.
SOAs depart from this model by introducing wholesale changes to how security is incorporated and applied.
Relying heavily on the extensions and concepts established by the WS-Security framework, the security models used within SOA emphasize the placement of security logic onto the messaging level.
SOAP messages provide header blocks in which security logic can be stored. That way, wherever the message goes, so does its security information.
This approach is required to preserve individual autonomy and loose coupling between services, as well as the extent to which a service can remain fully stateless.

F.Administration

Maintaining component-based applications involves keeping track of individual component instances, tracing local and remote communication problems, monitoring server resource demands, and the standard database administration tasks.
Distributed Internet architecture further introduces the Web server and with it an additional physical environment that requires attention while solutions are in operation.
Because clients, whether local or external to an organization, connect to these solutions using HTTP, the Web server becomes the official first point of contact.
It must be designed for scalability a requirement that has led to the creation of Web server farms that pool resources.
Enterprise-level SOAs typically require additional runtime administration. Problems with messaging frameworks can go undetected than with RPC-based data exchanges.
This is because so many variations exist as to how messages can be interchanged.
RPC communication requires a response from the initiating component, indicating success or failure. Upon encountering a failure condition, an exception handling routine kicks in.
Exception handling with messaging frameworks can be complex and less robust.
Although WS-* extensions are being positioned to better deal with these situations, administration effort is expected to remain high.
Other maintenance tasks, such as resource management, are required.

To best foster reuse and composability, a useful part of an administration infrastructure for enterprises building large amounts of Web services is a private registry.
UDDI is one of the technologies used for standardizing this interface repository, which can be manually or programmatically accessed to discover service descriptions.

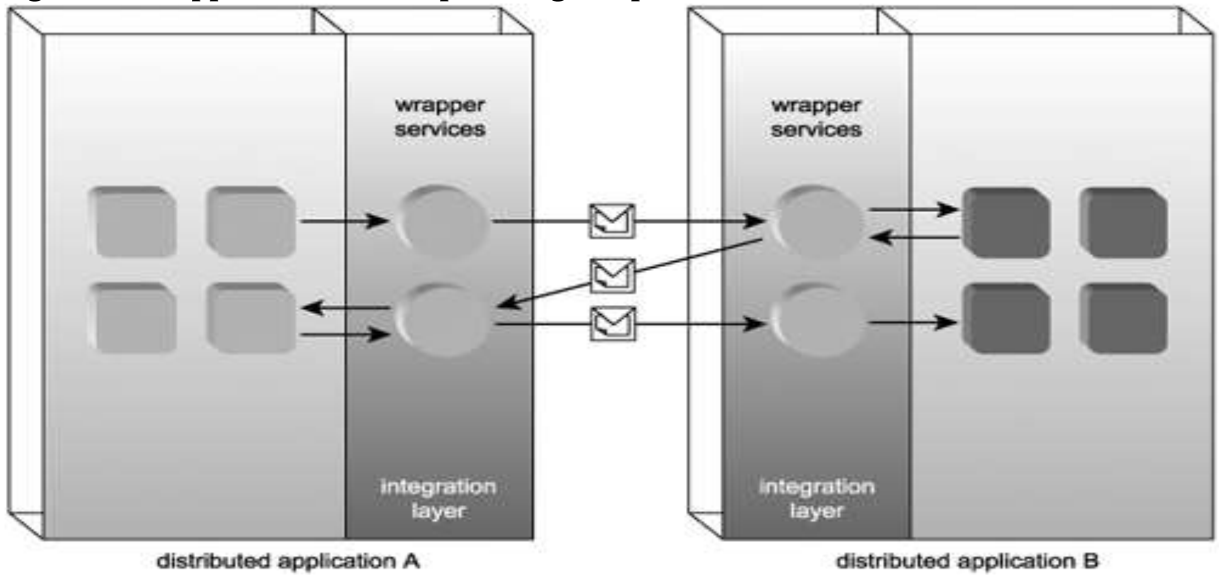
X.COMPARE SOA vs. hybrid Web service architecture

The use of Web services within traditional architectures is completely legitimate. Due to the development support for Web services in many established programming languages, they can be positioned to fit in with older application designs.

A. Web services as component wrappers

The role of Web services introduces an integration layer that consists of wrapper services that enable synchronous communication via SOAP-compliant integration channels (Figure.1).
The initial release of the SOAP specification and the first generation of SOAP servers were specifically designed to duplicate RPC-style communication using messages.

Figure 1. Wrapper services encapsulating components



These integration channels are utilized in integration architectures to facilitate communication with other applications or outside partners.
They are used to enable communication with other solutions and to take advantage of the features offered by third-party utility Web services.
Regardless of their use within traditional architectures, it clarify that a distributed Internet architecture that incorporates Web services in this manner does not qualify as a true SOA.
It is a distributed Internet architecture that uses Web services.
Instead of mirroring component interfaces and establishing point-to-point connections with Web services, SOA provides strong support for a variety of messaging models.
Web services within SOAs are subject to specific design requirements, such as those provided by service-orientation principles.
These and other characteristics support the pursuit of consistent loose coupling. Once achieved, a single service is never limited to point-to-point communication; it can accommodate any number of current and future requestors.

B.Web services within SOA

While SOAs can vary in size and quality, there are tangible characteristics that distinguish an SOA from other architectures that use Web services.

SOAs are built with a set of Web services designed to collectively automate one or business processes and that SOA promotes the organization of these services into specialized layers that abstract specific parts of enterprise automation logic.
By standardizing on SOA across an enterprise, a natural interoperability emerges that transcends proprietary application platforms.
This allows for previously disparate environments to be composed in support of new and evolving business automation processes.

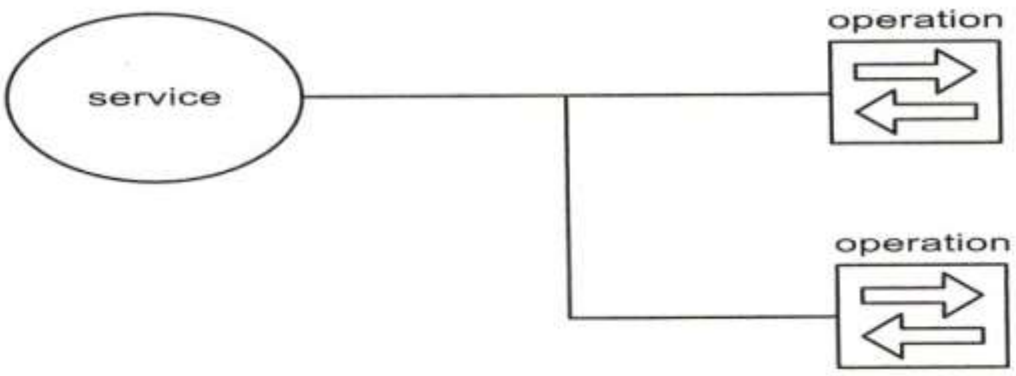
XI. THE COMPONENTS OF SOA AND THEIR INTER-RELATION

To understand what constitutes a true SOA, we need to abstract the key components of the Web services framework and study their relationships closely.

a)Logical components of the Web services framework.

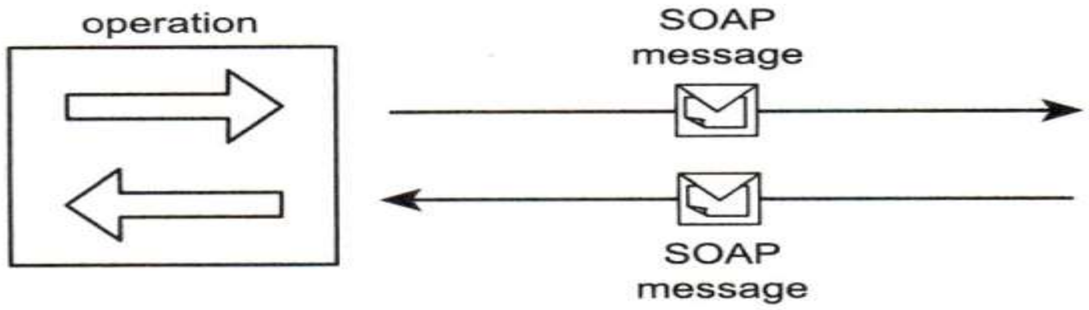
As shown in Figure.1, each web services contain one or more operations.

Figure1. A Web service sporting two operations



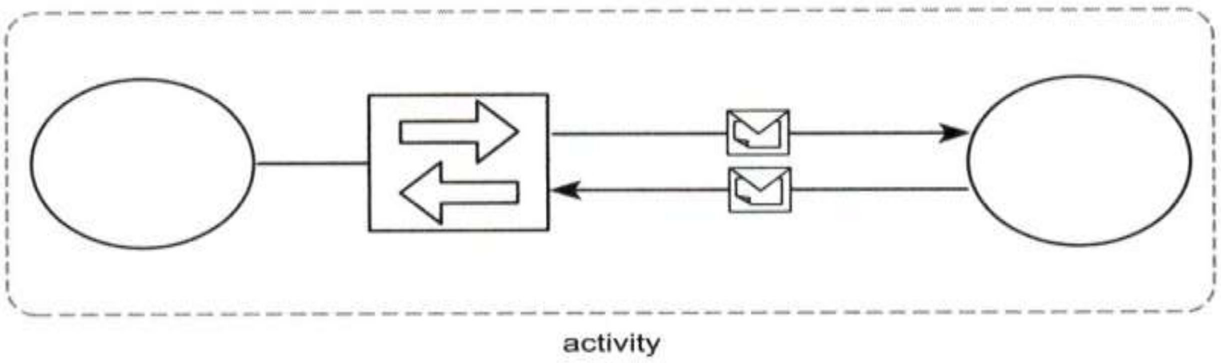
- A new symbol represents operations separately from the service.
- Each operation governs the process of a **specific function** the web service is capable of performing. In Figure.2, an operation sends and receives SOAP messages.

Figure 2. An operation processing outgoing and incoming SOAP messages



As shown in Fig.3, Web services form an *activity* through which they can collectively automate a task.

Figure 3. A communication scenario between Web services.



b)Logical components of automation logic

The Web services framework provides us a technology base for enabling connectivity; it establishes a modularized perspective of how automation logic can be comprised of independent units.

To illustrate the modularity of Web services, let's abstract the following parts of the framework:

- SOAP messages
- Web service operations
- Web services
- Activities

The latter three items represent units of logic that perform work and communicate using SOAP messages. To better illustrate this in a service-oriented perspective, these terms are as follows:

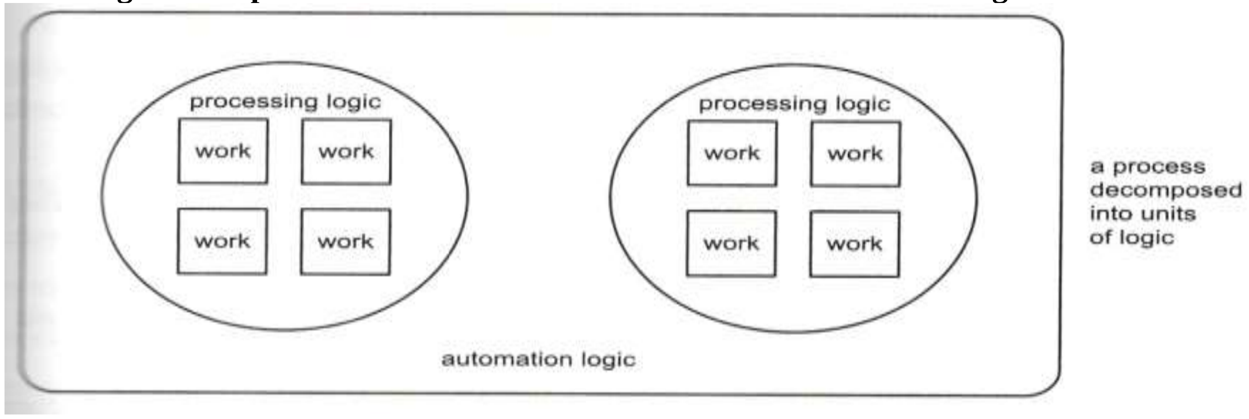
- Messages
- Operations
- Services
- Processes

We qualify automation logic parts by relating each to different sized units of logic, as follows:

- Messages = units of communication
- Operations = units of work
- Services = units of processing logic
- Processes = units of automation logic

Figure.4 provides us with a primitive view of how operations and services represent units of logic that can be assembled to comprise a unit of automation logic.

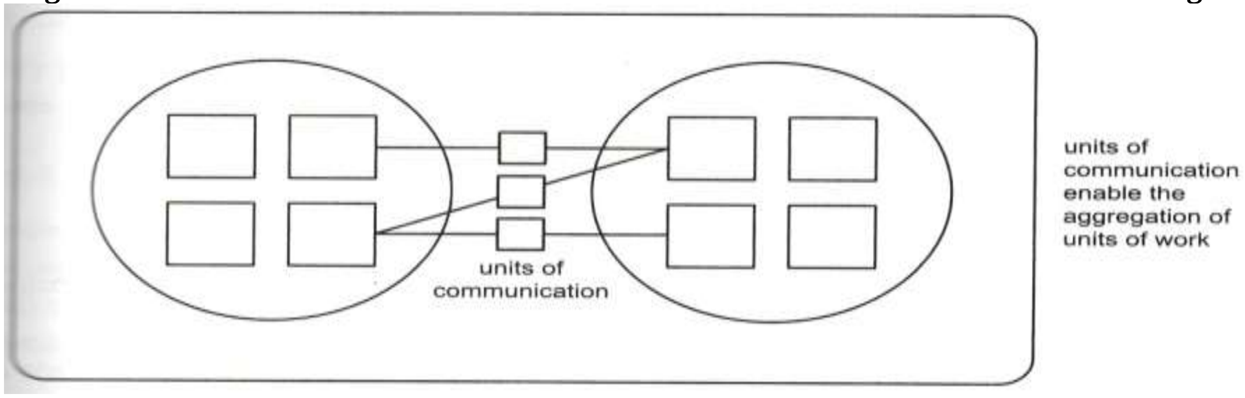
Figure 4. A primitive view of how SOA modularizes automation logic into units



In Figure 5, we establish that messages are a means by which all units of processing logic (services) communicate.

This illustrates that no actual processing of that logic can be performed without issuing units of communication (in this case, messages).

Figure 5. A view of how units of communication enable interaction between units of logic


units of communication
units of communication enable the aggregation of units of work

The purpose of these views is to express that processes, services, and operations, on the fundamental level, provide a flexible means of partitioning and modularizing logic.
To derive this view from the Web services framework, we demonstrated the suitability of the Web services platform as a means of implementation for SOA.

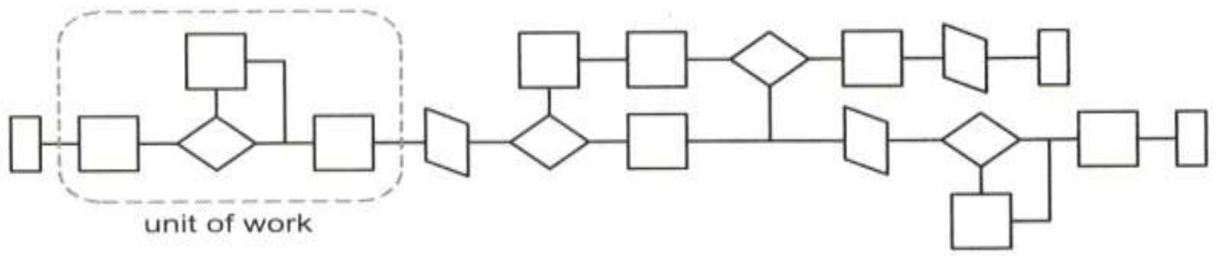
c) Components of an SOA

Components establish a level of enterprise logic abstraction, as follows:

i)A *message* represents the data required to complete some or all parts of a unit of work.

ii)An *operation* represents the logic required to process messages in order to complete a unit of work (Figure.6).

Figure 6. The scope of an operation within a process.

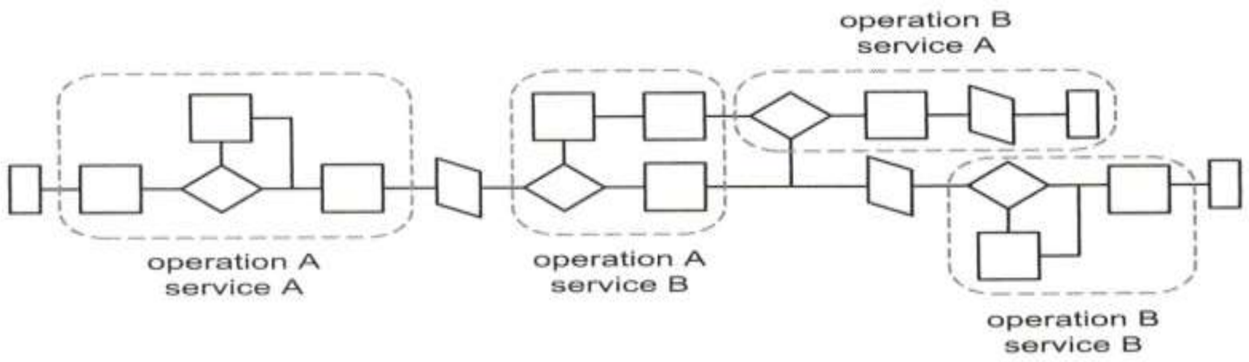


iii)A *service* represents a logically grouped set of operations capable of performing related units of work.

iv)A *process* contains the business rules that determine which service operations are used to complete a unit of automation.

A process represents a large piece of work that requires the completion of smaller units of work (Figure.7).

Figure 7. Operations belonging to different services representing various parts of process logic



d)How components in an SOA inter-relate

The components are required to relate to each other:

- An operation sends and receives messages to perform work.
- A service group is a collection of related operations.
- A process instance can compose service.
- A process instance invokes a series of operations to complete its automation.
- An operation is defined by the message it processes.
- A service is defined by the operations that comprise it.
- A process instance is not defined by its service because it may require a subset of the functionality offered by the services.
- Every process instance is partially defined by the operation it uses.

A service-oriented architecture is an environment standardized according to the principles of service-orientation in which a process that uses services (a service-oriented process) can execute.

Figures 8 and 9 illustrate these relationships.

Figure 8. How the components of SOA relate

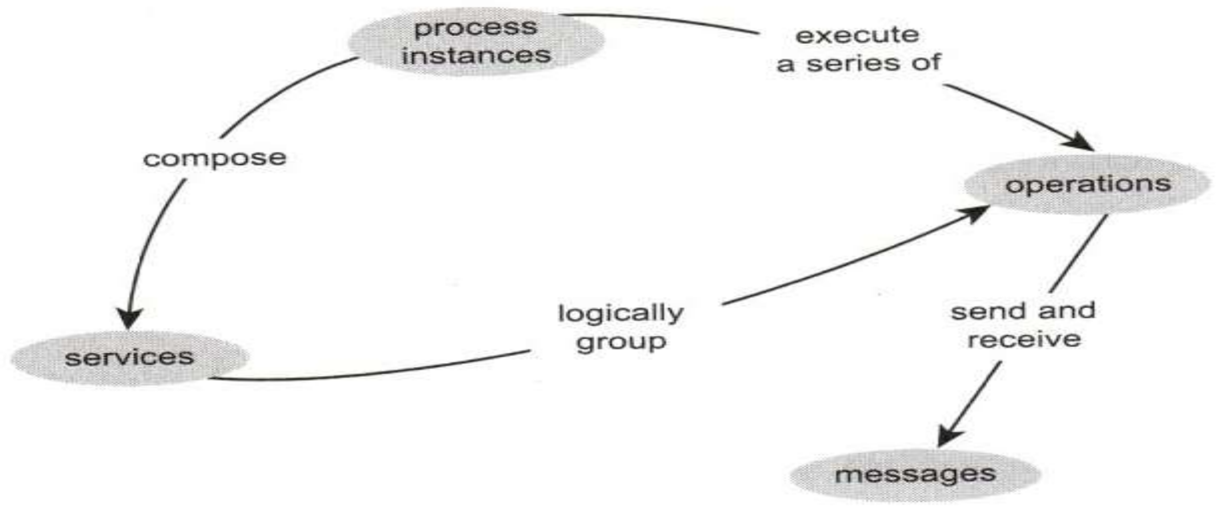
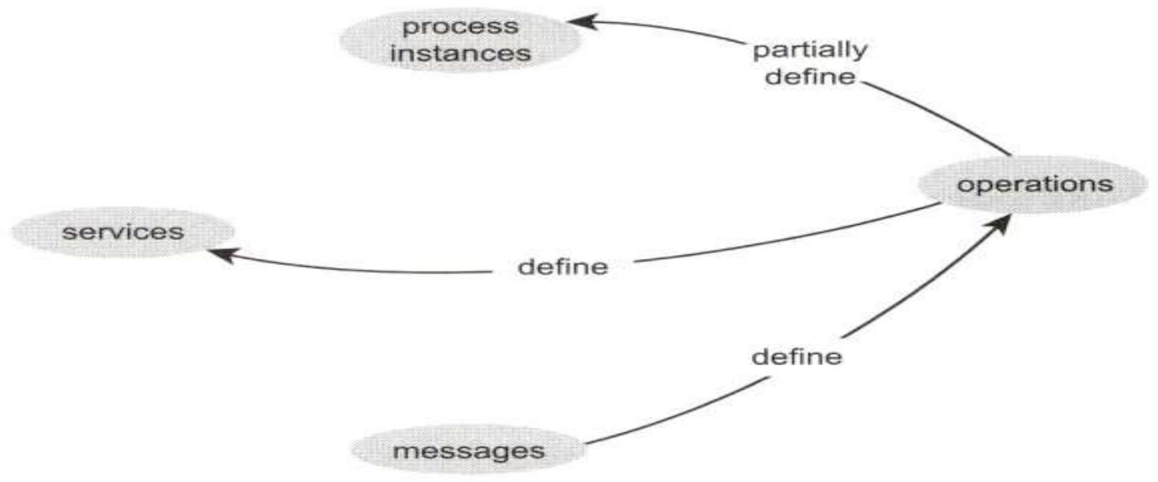


Figure 9. How the components of a SOA define each other



XII. THE PRINCIPLES OF SERVICE ORIENTATION

Service-orientation can be viewed as a manner in which to realize a separation of concerns.

Common principles of service-orientation

Services are reusable Regardless of whether immediate reuse opportunities exist; services are designed to support reuse.

Services share a formal contract: For services to interact, they need not share anything but a formal contract that describes each service and defines the terms of information exchange.

Services are loosely coupled Services must be designed to interact without the need for tight, cross-service dependencies.

Services abstract underlying logic: The part of a service that is visible to the outside world is what is exposed via the service contract. Underlying logic, beyond what is expressed in the descriptions that comprise the contract, is invisible and irrelevant to service requestors.

Services are composable Services may compose other services. This allows logic to be represented at different levels of granularity and promotes reusability and the creation of abstraction layers.

Services are autonomous: The logic governed by a service resides within an explicit boundary. The service has control within this boundary and is not dependent on other services for it to execute its governance.

Services are stateless Services should not be required to manage state information, as that can block their ability to remain loosely coupled. Services should be designed to maximize statelessness even if that means deferring state management elsewhere.

Services are discoverable Services should allow their descriptions to be discovered and understood by humans and service requestors that may be able to make use of their logic.

Of these eight, autonomy, loose coupling, abstraction, and the need for a formal contract can be considered the core principles that form the baseline foundation for SOA.

There are other qualities associated with services and service-orientation. Examples include self-descriptive and coarse-grained interface design characteristics.

1. Services are reusable

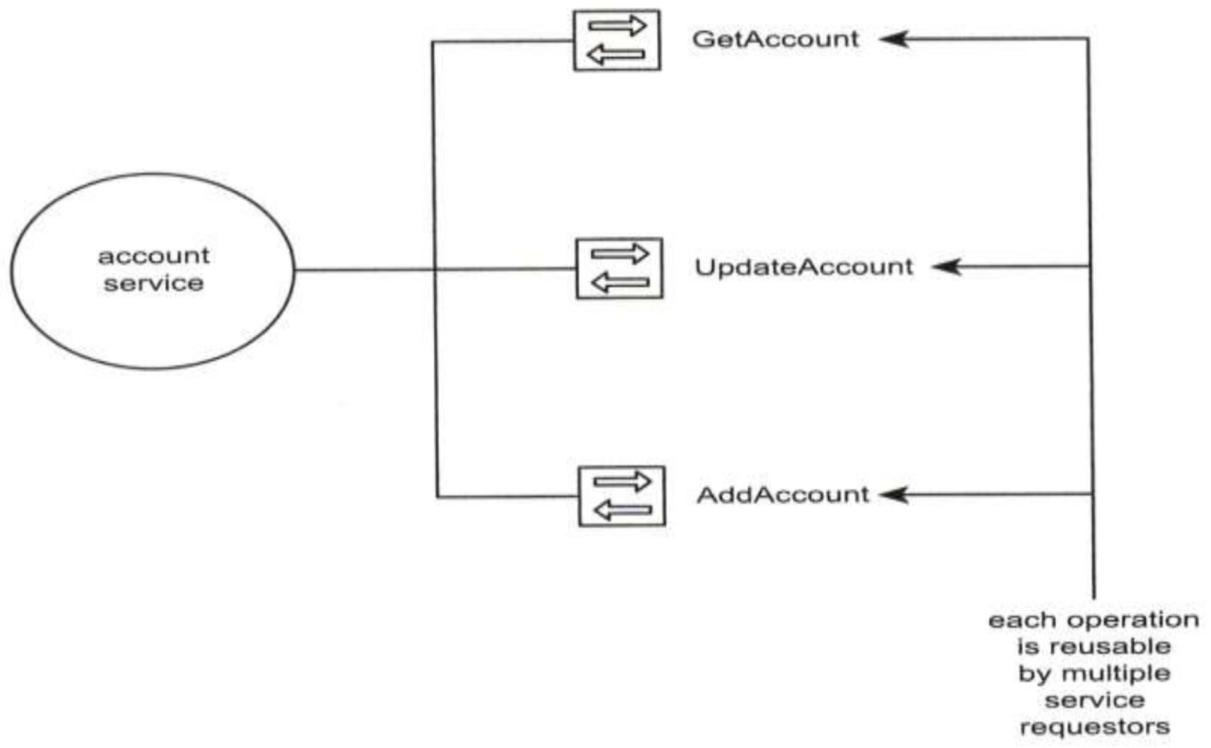
Service-orientation encourages reuse in all services, regardless if immediate requirements for reuse exist. By applying design standards that make each service reusable, the chances of being able to accommodate future requirements with less development effort are increased.

Inherently reusable services reduce the need for creating wrapper services that expose a generic interface over top of less reusable services.

This principle facilitates all forms of reuse, including inter-application interoperability, composition, and the creation of cross-cutting or utility services.

A service is a collection of related operations. It is the logic encapsulated by the individual operations that must be deemed reusable to warrant representation as a reusable service (Figure 1).

Figure 1. A reusable service exposes reusable operations.



Messaging indirectly supports service reusability through the use of SOAP headers. These allow for messages to become increasingly self-reliant by grouping metadata details with message content into a single package (the SOAP envelope).

Messages can be equipped with processing instructions and business rules that allow them to dictate to recipient services how they should be processed.

The processing-specific logic embedded in a message alleviates the need for a service to contain this logic. It imposes a requirement that service operations become less activity-specific or generic. The generic a service's operations are the reusable the service.

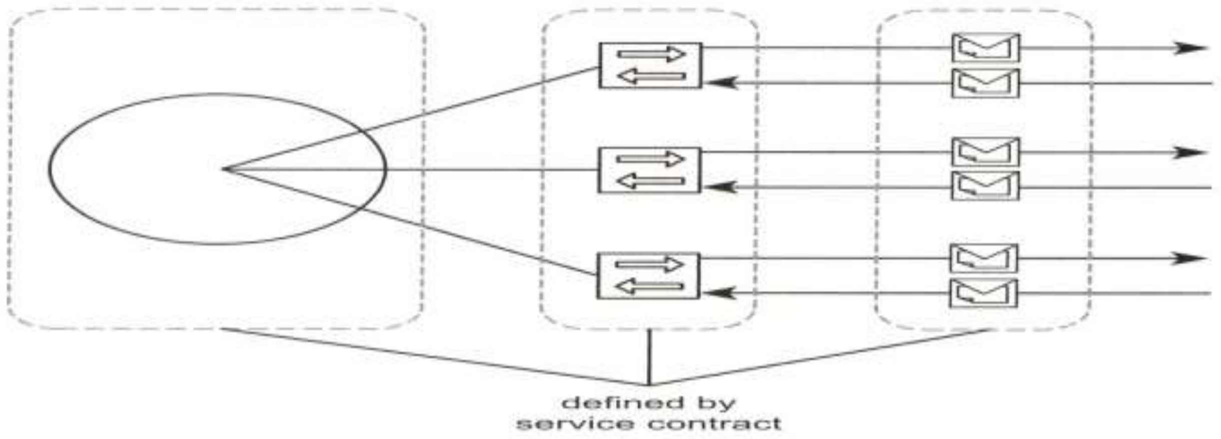
2. Services share a formal contract

Service contracts provide a formal definition of:

- the service endpoint
- each service operation
- every input and output message supported by each operation
- rules and characteristics of the service and its operations

Service contracts define all of the parts of an SOA (Figure 2). Good service contracts may provide semantic information that explains how a service may go about accomplishing a particular task. This information establishes the agreement made by a service provider and its service requestors.

Figure 2. Service contracts formally define the service, operation, and message components of a service-oriented architecture.



Because this contract is shared among services, its design is extremely important. Service requestors that agree to this contract can become dependent on its definition. Contracts need to be carefully maintained and versioned after their initial release.

Service description documents, such as the WSDL definition, XSD schemas, and policies, can be viewed collectively as a communications contract that expresses exactly how a service can be programmatically accessed.

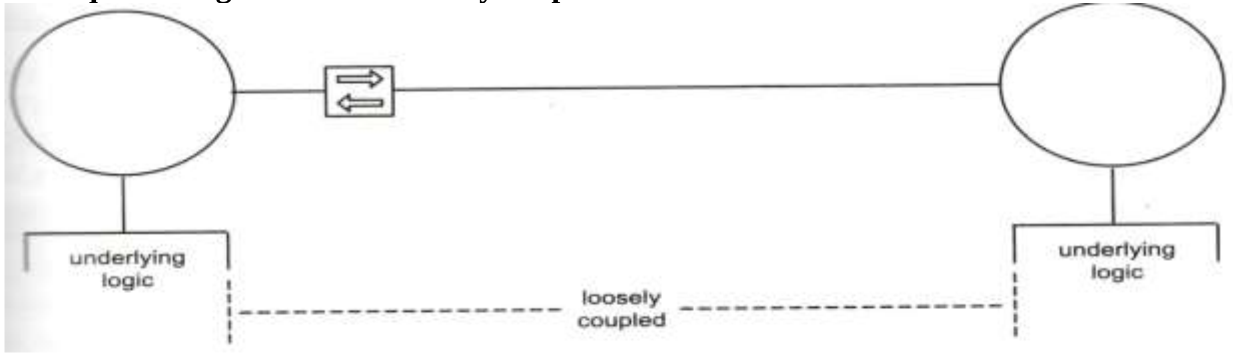
3. Services are loosely coupled

No one can predict how an IT environment will evolve. How automation solutions grow, integrate, or are replaced over time can never be accurately planned out because the requirements that drive these changes are al always external to the IT environment.

Being able to ultimately respond to unforeseen changes in an efficient manner is a key goal of applying service-orientation.

Realizing this form of agility is directly supported by establishing a loosely coupled relationship between services (Figure. 3).

Figure 3. Services limit dependencies to the service contract, allowing underlying provider and requestor logic to remain loosely coupled.



Loose coupling is a condition wherein a service acquires knowledge of another service while remaining independent of that service.

Loose coupling is achieved through the use of service contracts that allow services to interact within predefined parameters.

Within a loosely coupled architecture, service contracts tightly couple operations to services. When a service is formally described as being the location of an operation, other services will depend on that operation-to-service association.

4. Services abstract underlying logic

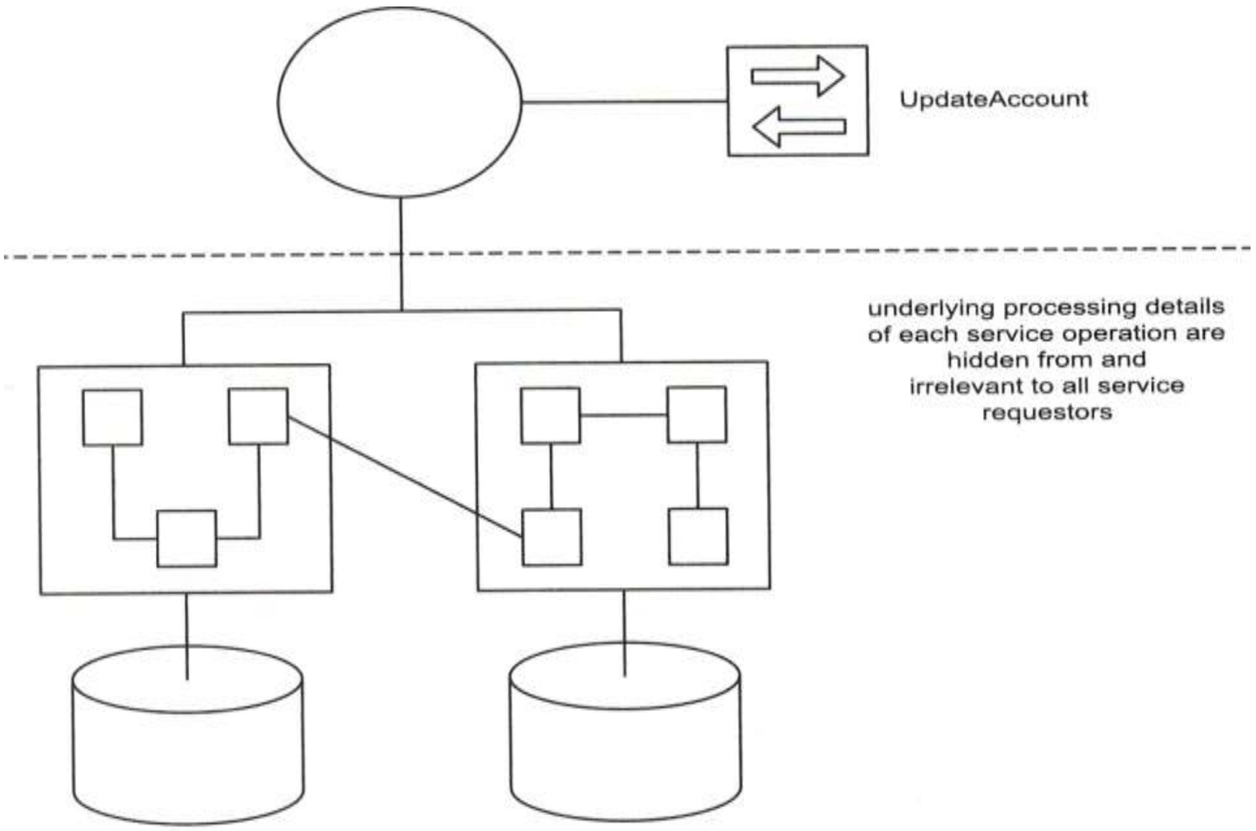
Service interface-level abstraction principle allows services to act as black boxes, hiding their details from the outside world.

The scope of logic represented by a service significantly influences the design of its operations and its position within a process.

There is no limit to the amount of logic a service can represent. A service may be designed to perform a simple task, or it may be positioned as a gateway to an entire automation solution.

There is no restriction as to the source of application logic a service can draw upon. For example, a single service can, technically, expose application logic from two different systems (Figure.4).

Figure 4. Service operations abstract the underlying details of the functionality they expose.


UpdateAccount
underlying processing details of each service operation are hidden from and irrelevant to all service requestors

Operation granularity is a design consideration that is directly related to the range and nature of functionality being exposed by the service.

It is the individual operations that collectively abstract the underlying logic. Services act as containers for these operations.

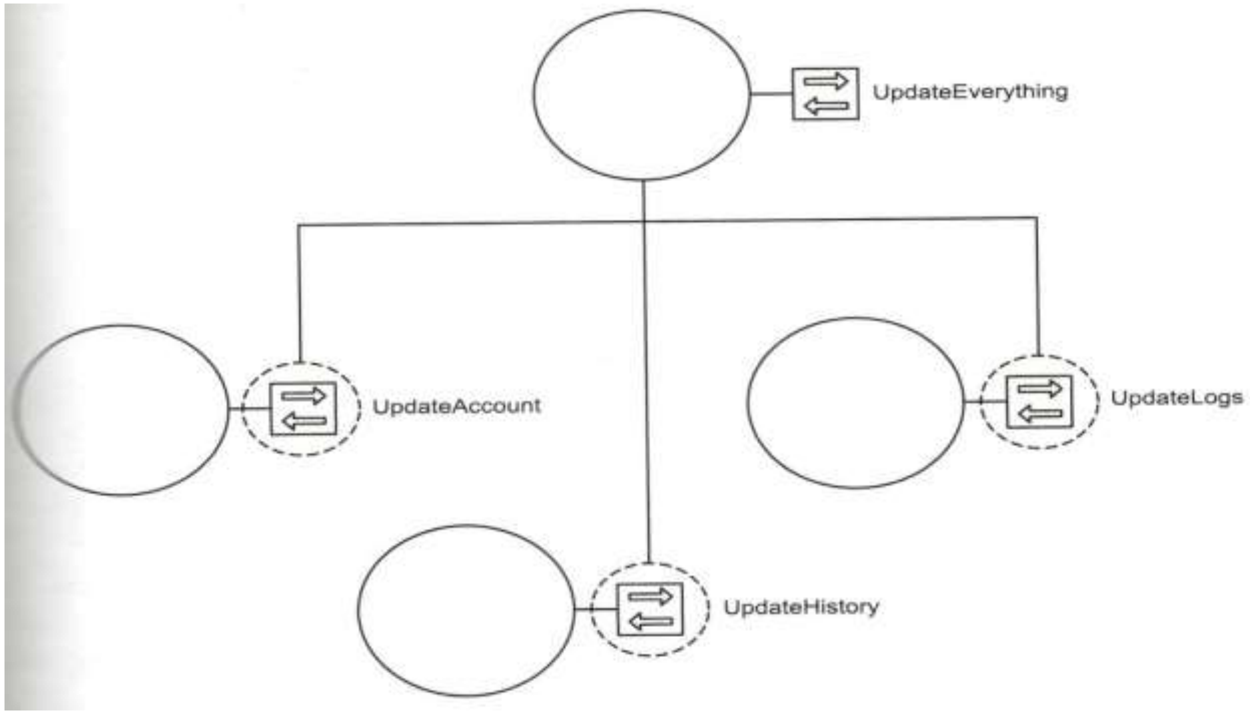
Service interface-level abstraction is one of the inherent qualities provided by Web services. The loosely coupled communications structure requires that the piece of knowledge services need to interact is each others' service descriptions.

We don't know or care what the Business License Office needs to do to process our application. We are interested in the expected outcome: the issuance of our license.

5. Services are composable

A service can represent any range of logic from any types of sources, including other services. The main reason to implement this principle is to ensure that services are designed so that they can participate as effective members of other service compositions if ever required. This requirement is irrespective of whether the service itself composes others to accomplish its work (Figure 5).

Figure 5. The UpdateEverything operation encapsulating a service composition



A common SOA extension that underlines composability is the concept of orchestration. A service-oriented process (a service composition) is controlled by a parent process service that composes process participants.

The requirement for any service to be composable places an emphasis on the design of service operations.

Composability is another form of reuse, and operations need to be designed in a standardized manner and with an appropriate level of granularity to maximize composition opportunities.

6. Services are autonomous

Autonomy requires that the range of logic exposed by a service exist within an explicit boundary. This allows the service to execute self-governance of all its processing.

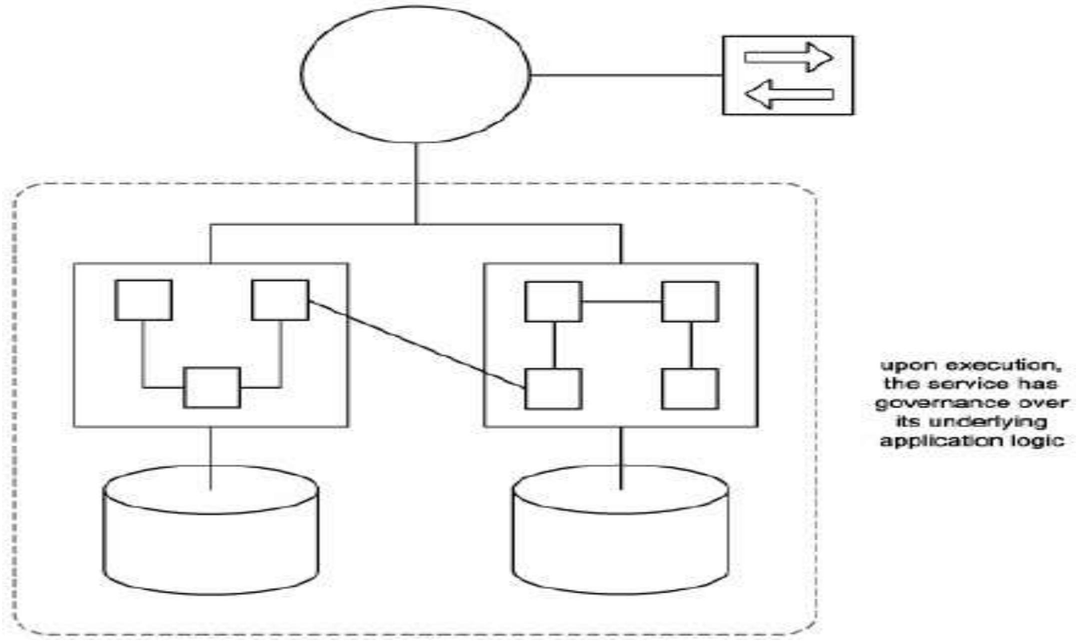
It eliminates dependencies on other services, which frees a service from ties that could inhibit its deployment and evolution (Figure.6). Service autonomy is a consideration when deciding how application logic should be divided up into services and which operations should be grouped together within a service context.

Deferring the location of business rules is one way to strengthen autonomy and keep services generic.

Processes assume this role by owning the business rules that determine how the process is structured and how services are composed to automate the process logic.

Autonomy does not necessarily grant a service exclusive ownership of the logic it encapsulates. It guarantees that at the time of execution, the service has control over whatever logic it represents. We can make a distinction between two types of autonomy.

Figure 6. Autonomous services have control over underlying resources


upon execution, the service has governance over its underlying application logic

Service-level autonomy Service boundaries are distinct from each other, but the service may share underlying resources. For example, a wrapper service that encapsulates a legacy environment that is used independently from the service has service-level autonomy. It governs the legacy system but shares resources with other legacy clients.

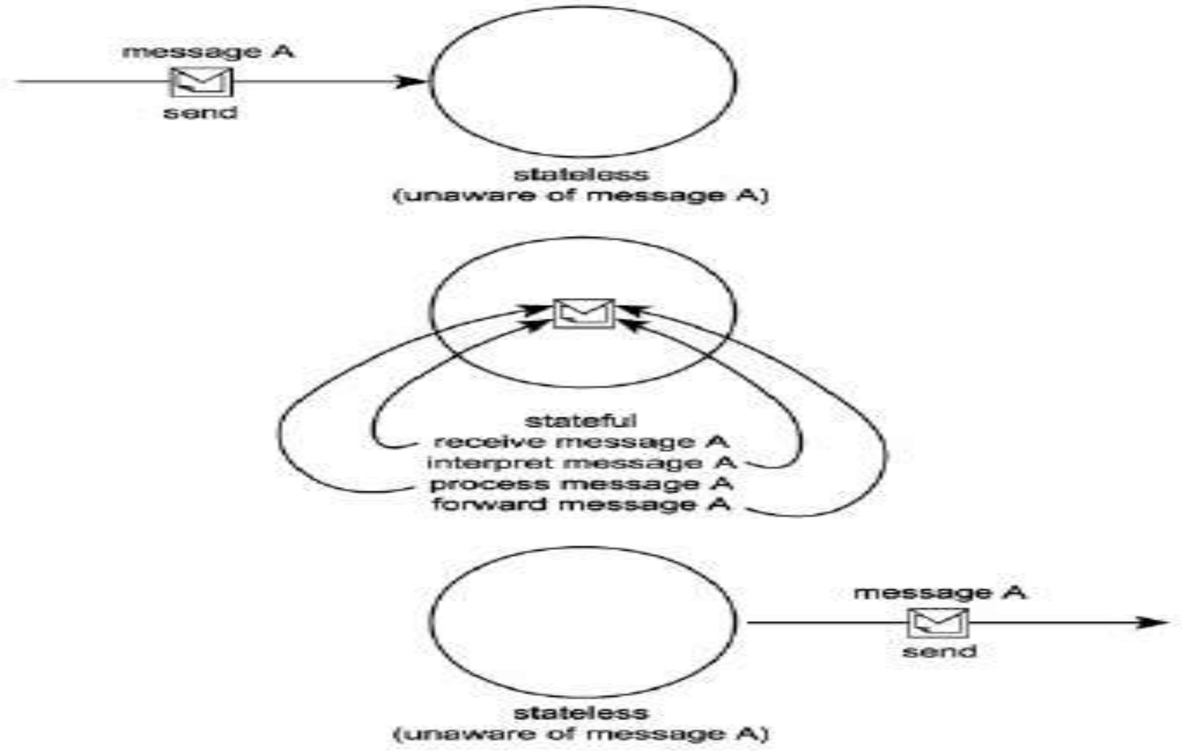
Pure autonomy The underlying logic is under complete control and ownership of the service. This is typically the case when the underlying logic is built from the ground up in support of the service.

7. Services are stateless

Services should minimize the amount of state information they manage and the duration for which they hold it. State information is data-specific to a current activity.

While a service is processing a message, for example, it is temporarily stateful (Figure.7).

Figure 7. Stateless and stateful stages a service passes through while processing a message.



If a service is responsible for retaining state for longer periods of time, its ability to remain available to other requestors will be impeded.

Statelessness is a preferred condition for services and one that promotes reusability and scalability.

For a service to retain as little state as possible, its individual operations need to be designed with stateless processing considerations.

A quality of SOA that supports statelessness is the use of document-style messages.

The intelligence added to a message, the independent and self-sufficient it remains.

8. Services are discoverable

Discovery helps avoid the accidental creation of redundant services or services that implement redundant logic.

Because each operation provides a reusable piece of processing logic, metadata attached to a service needs to sufficiently describe not the service's overall purpose, but the functionality offered by its operations.

This service-orientation principle is related to but distinct from the contemporary SOA characteristic of discoverability.

On an SOA level, discoverability refers to the architecture's ability to provide a discovery mechanism, such as a service registry or directory.

This effectively becomes part of the IT infrastructure and can support implementations of SOA.

On a service level, the principle of discoverability refers to the design of an individual service so that it can be as discoverable as possible.

XIII. SERVICE-ORIENTATION PRINCIPLES INTER-RELATE

1. Service reusability

When a service encapsulates logic that is useful than one service requestor, it can be considered reusable.

The concept of reuse is supported by a number of complementary service principles, as follows.

Service autonomy establishes an execution environment that facilitates reuse because the service has independence and self-governance.

The less dependencies a service has, the broader the applicability of its reusable functionality.

Service statelessness supports reuse because it maximizes the availability of a service and promotes a generic service design that defers activity-specific processing outside of service logic boundaries.

Service abstraction fosters reuse because it establishes the black box concept, where processing details are hidden from requestors.

This allows a service to express a generic public interface.

Service discoverability promotes reuse, as it allows requestors to search for and discover reusable services.

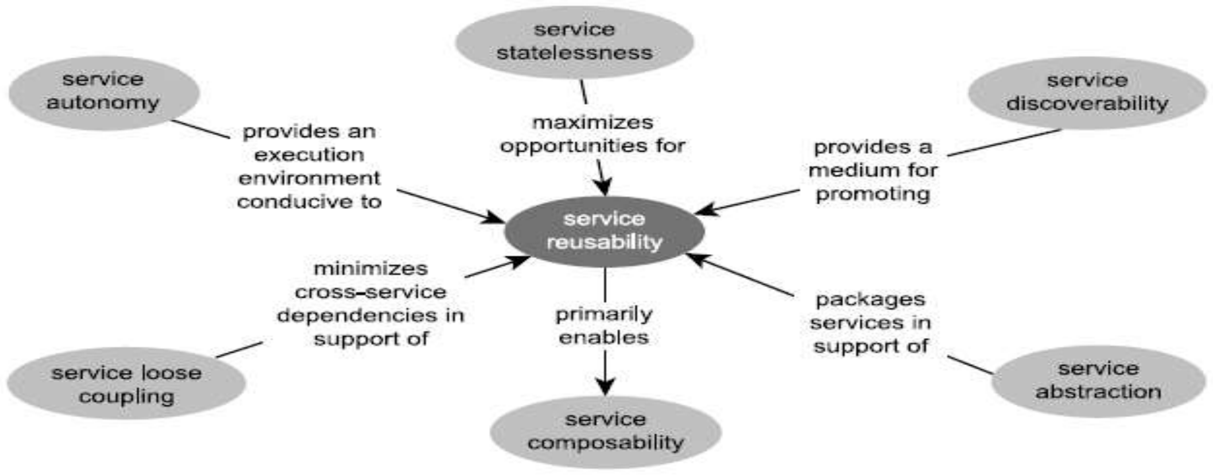
Service loose coupling establishes an inherent independence that frees a service from immediate ties to others. This makes it a deal easier to realize reuse.

The principle of service reuse enables the following related principle:

Service composability is possible because of reuse. The ability for one service to compose an activity around the utilization of a collection of services is feasible when those services being composed are built for reuse.

Figure 1 provides a diagram showing how the principle of service reusability relates to others.

Figure 1. Service reusability and its relationship with other service-orientation principles


service statelessness
service autonomy
service discoverability
provides an execution environment conducive to
maximizes opportunities for
provides a medium for promoting
service reusability
minimizes cross-service dependencies in support of
primarily enables
packages services in support of
service loose coupling
service abstraction
service composability

2. Service contract

A service contract is a representation of a service's collective metadata. It standardizes the expression of rules and conditions that need to be fulfilled by any requestor wanting to interact with the service.

Service contracts represent a cornerstone principle in service-orientation and support other principles in various ways, as follows:

Service abstraction is realized through a service contract, as it is the metadata expressed in the contract that defines the information made available to service requestors. All additional design, processing, and implementation details are hidden behind this contract.

Service loose coupling is made possible through the use of service contracts. Processing logic from different services does not need to form tight dependencies; they need an awareness of each other's communication requirements, as expressed by the service description documents that comprise the service contract.

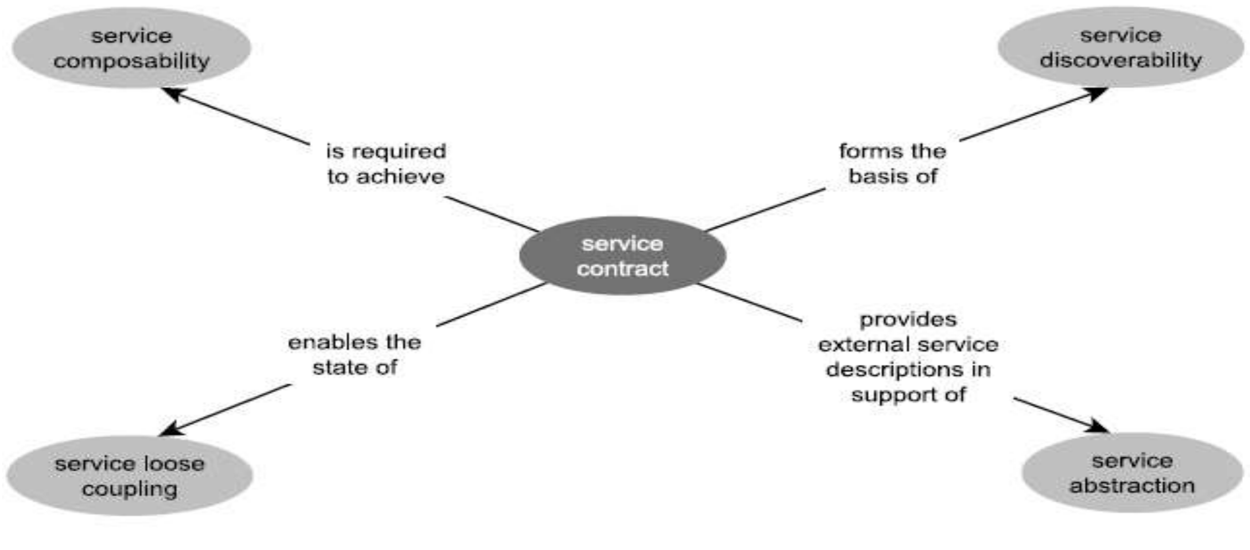
Service composability is indirectly enabled through the use of service contracts. It is via the contract that a controller service enlists and uses services that act as composition members.

Service discoverability is based on the use of service contracts.

While some registries provide information supplemental to that expressed through the contract, it is the service description documents that are searched for in the service discovery process.

The diagram in Figure.2 illustrates how the principle of service contract usage relates to others.

Figure 2. The service contract and its relationship with other service-orientation principles



3. Service loose coupling

Loose coupling is a state that supports a level of independence between services (or between service providers and requestors).

Independence or non-dependency is a fundamental aspect of services and SOA.

The principle of persisting loose coupling across services supports the following other service-orientation principles:

Service reusability is supported through loose coupling because services are freed from tight dependencies on others. This increases their availability for reuse opportunities.

Service composability is fostered by the loose coupling of services, especially when services are dynamically composed.

Service statelessness is directly supported through the loosely coupled communications framework established by this principle.

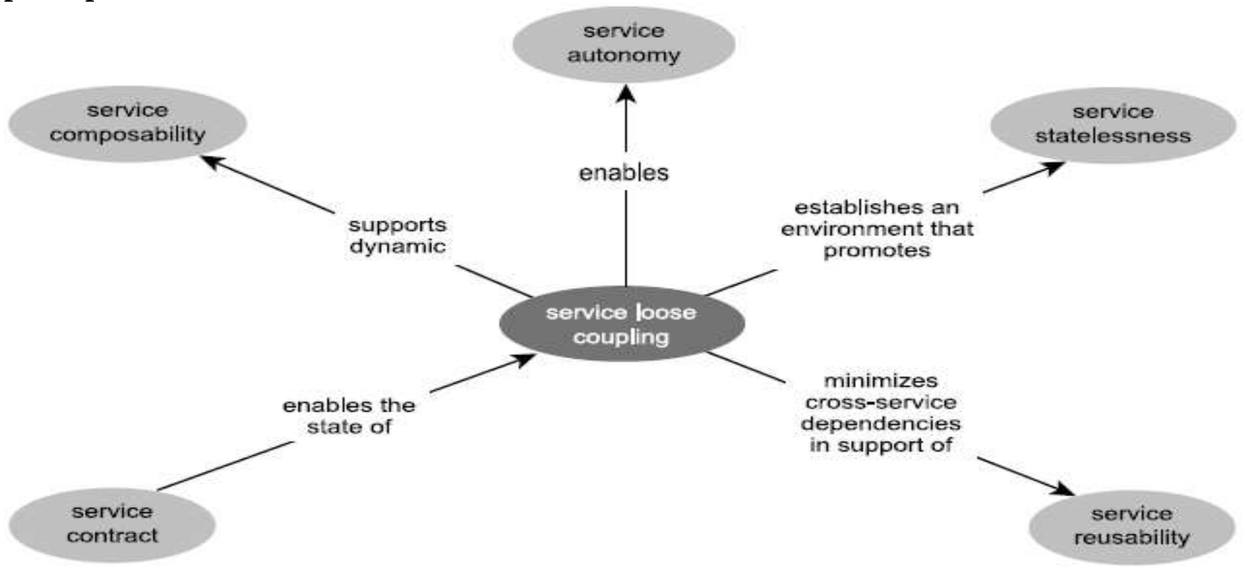
Service autonomy is made possible through this principle, as *loose coupling* minimizes cross-service dependencies.

Service loose coupling is directly implemented through the application of a related service-orientation principle:

Service contracts are what enable loose coupling between services, as the contract is the piece of information required for services to interact.

Figure.3 demonstrates these relationships.

Figure 3. Service loose coupling and its relationship with other service-orientation principles



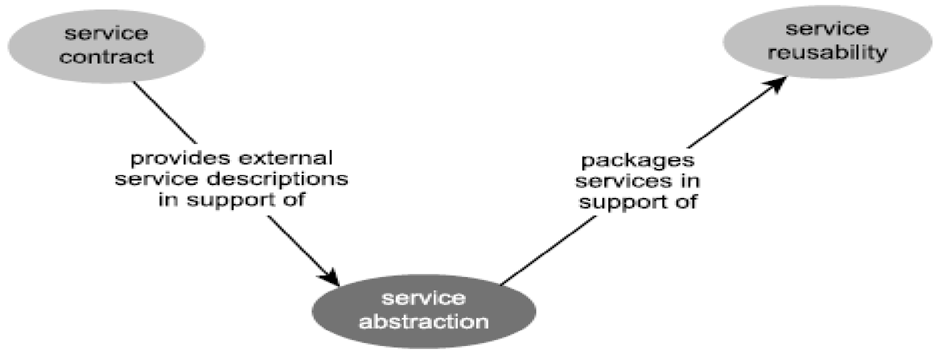
4. Service abstraction

Part of building solutions with independent services is allowing those services to encapsulate complex processing logic and exposing that logic through a generic and descriptive interface.

Service contracts implement service abstraction by providing the official description information that is made public to external service requestors.

Service reusability is supported by abstraction, as long as what is being abstracted is reusable. These relationships are shown in Figure 4.

Figure 4. Service abstraction and its relationship with other service-orientation principles


service contract
provides external service descriptions in support of
service abstraction
packages services in support of
service reusability

5. Service composability

Designing services so that they support composition by others is fundamental to building service-oriented solutions.

Service composability is tied to service-orientation principles that support the service composition, as follows:

Service reusability is what enables one service to be composed by others.

Reusable services can be incorporated within different compositions or reused independently by other service requestors.

Service loose coupling establishes a communications framework that supports the dynamic service composition.

Because services are freed from many dependencies, they are available to be reused via composition.

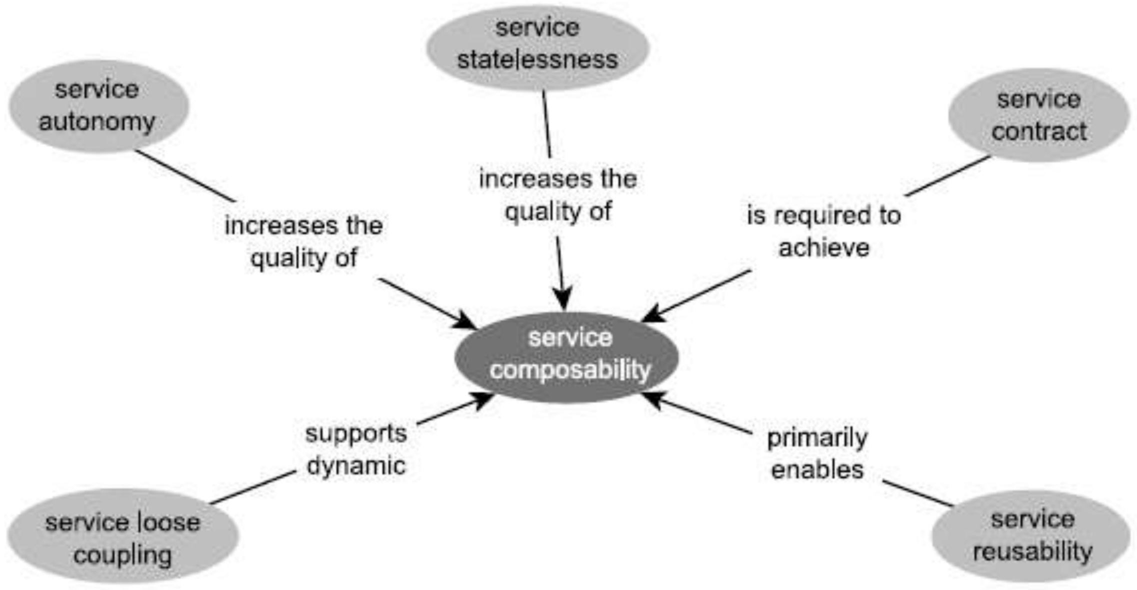
Service statelessness supports service composability in larger compositions. A service composition is dependent on the design quality and commonality of its collective parts.

If all services are stateless (deferring activity-specific logic to messages), the overall composition executes harmoniously.

Service autonomy held by composition members strengthens the overall composition, but the autonomy of the controller service is decreased due to the dependencies on its composition members.

Service contracts enable service composition by formalizing the runtime agreement between composition members. Figure.5 further illustrates these relationships.

Figure 5. Service composability and its relationship with other service-orientation principles



6. Service autonomy

This principle applies to a service's underlying logic. By providing an execution environment over which a service has complete control, service autonomy relates to several other principles:

Service reusability is achieved when the service offering reusable logic has self-governance over its own logic.

Service Level Agreement (SLA) type requirements that come to the forefront for utility services with multiple requestors, such as availability and scalability, are fulfilled by an autonomous service.

Service composability is supported by service autonomy for much of the same reasons autonomy supports service reusability.

A service composition consisting of autonomous services is robust and collectively independent.

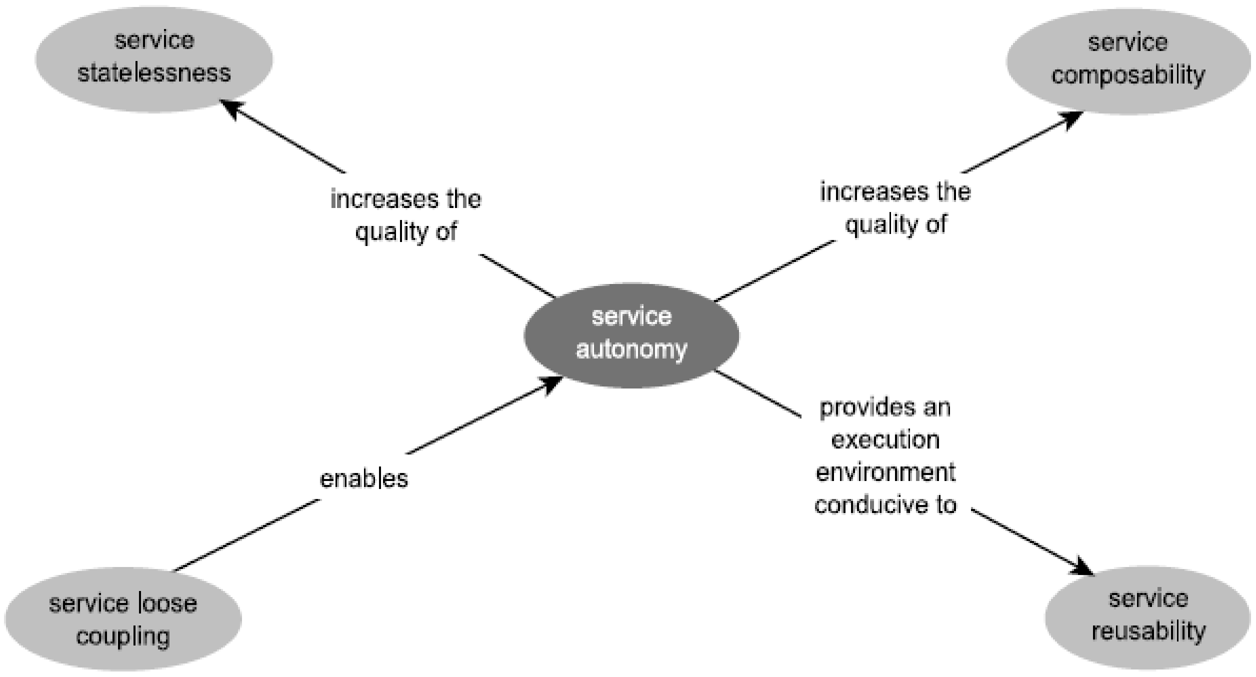
Service statelessness is best implemented by a service that can execute independently.

Autonomy indirectly supports service statelessness.

Service autonomy is a quality that is realized by leveraging the loosely coupled relationship between services.

Service loose coupling is an enabler of this principle. The diagram in Figure.6 shows how service autonomy relates to these other principles.

Figure 6. Service autonomy and its relationship with other service-orientation principles



7. Service statelessness

To successfully design services not to manage state requires the availability of resources surrounding the service to which state management responsibilities can be delegated.

The principle of statelessness is indirectly supported by the following service-orientation principles:

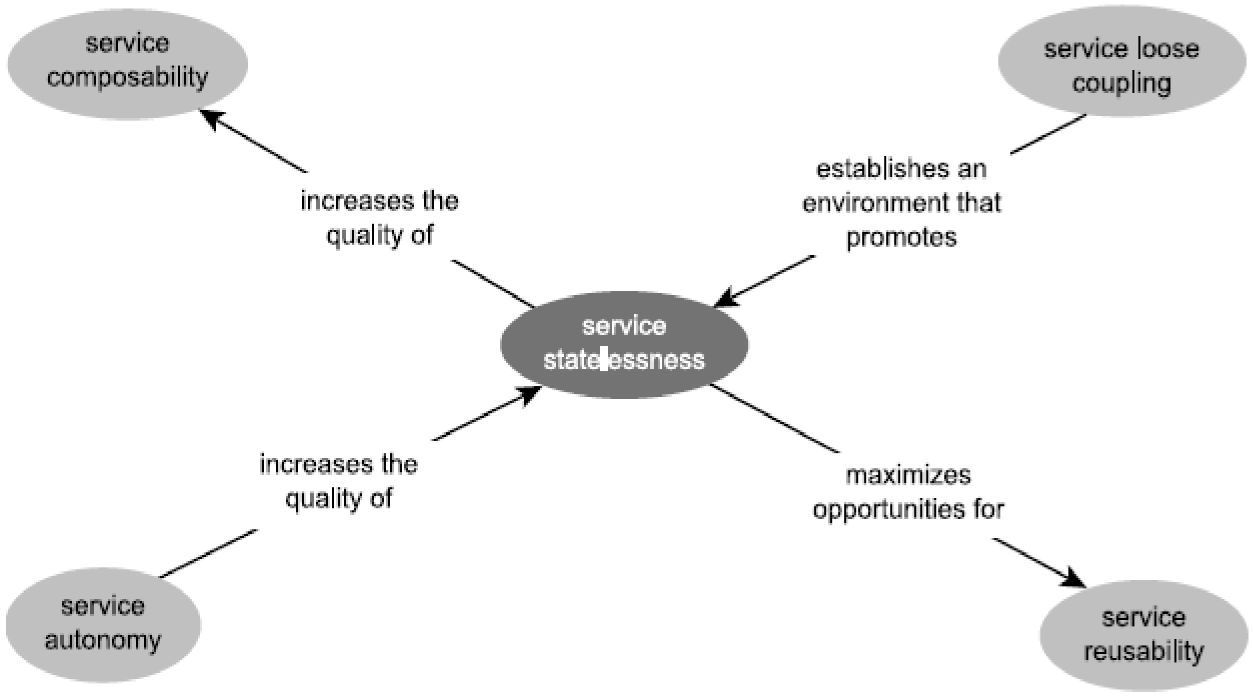
Service autonomy provides the ability for a service to control its own execution environment.

By removing or reducing dependencies it becomes easier to build statelessness into services, because the service logic can be fully customized to defer state management outside of the service logic boundary.

Service loose coupling and the overall concept of loose coupling establishes a communication paradigm that is fully realized through messaging.

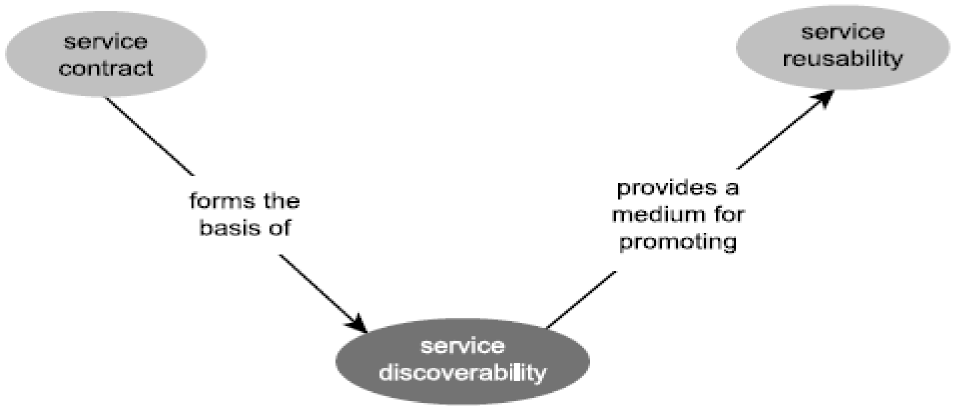
This supports service statelessness, as state information can be carried and persisted by the messages that pass through the services.

Service statelessness supports the following principles:
Service composability benefits from stateless composition members, as they reduce dependencies and minimize the overhead of the composition as a whole.
Service reuse becomes a reality for stateless services, as availability of the service to multiple requestors is increased and the absence of activity-specific logic promotes a generic service design.
Figure.7 illustrates how service statelessness relates to the other service-orientation principles.
Figure 7. Service statelessness and its relationship with other service-orientation principles



8. Service discoverability
Designing services so that they are discoverable enables an environment whereby service logic becomes accessible to new service requestors. This is why service discoverability is tied closely to the following service-orientation principles:
Service contracts are what service requestors (or those that create them) discover and subsequently assess for suitability.
The extent of a service's discoverability can typically be associated with the quality or descriptiveness of its service contract.
Service reusability is what requestors are looking for when searching for services and it is what makes a service useful once it has been discovered.
A service that isn't reusable would likely never need to be discovered because it would have been built for a specific service requestor in the first place.
The diagram in Figure 8 shows how service discoverability fits in with the other service-orientation principles.
SUMMARY:
Service-orientation principles are not realized in isolation; principles relate to and support other principles in different ways.
Principles, such as service reusability and service composability, benefit from the support of other implemented principles.
Principles, such as service loose coupling, service contract, and service autonomy, provide support for the realization of other principles.

Figure 8. Service discoverability and its relationship with other service-orientation principles.



XIV. A COMPARATIVE STUDY OF SERVICE ORIENTATION WITH OBJECT ORIENTATION

- ❖ SO emphasizes loose coupling between units of processing logic (services)
- ❖ OO supports reusability of loosely coupled programming routines, much of it is based on predefined class dependencies, resulting in tightly bound processing logic (objects)
- ❖ SO encourages coarse-grained interfaces (service descriptions) with information loaded messages
- ❖ OO supports fine-grained interfaces (APIs) so units of communication (RPC or local API calls) can perform various tasks
- ❖ SO expects the scope of a service to vary significantly
- ❖ OO objects tend to be smaller and specific in scope
- ❖ SO promotes activity-agnostic units of processing logic (services) that are driven into action by intelligent messages
- ❖ OO encourages the binding of processing logic with data into objects
- ❖ SO prefers services be designed to remain as stateless as possible
- ❖ OO promotes binding data and logic into stateful objects
- ❖ SO supports loosely coupled services
- ❖ OO supports composition but inheritance among classes leads to tightly coupled class dependencies.

Service-Orientation Principle	**Object-Orientation Principles**
Service Reusability	Much of object-orientation is geared toward the creation of reusable classes. The object-orientation principle of modularity standardized decomposition as a means of application design. Abstraction and encapsulation support reuse by requiring a separation of interface and implementation logic. Service reusability is a continuation of this goal.
Service Contract	The requirement for a service contract is very comparable to the use of interfaces when building object-oriented applications. Much like WSDL definitions, interfaces provide a means of abstracting the description of a class. And, much like the ÒWSDL firstÓ approach encouraged within SOA, the Òinterface firstÓ approach is considered an object-orientation best practice.

Service Loose Coupling	Although the creation of interfaces somewhat decouples a class from its consumers, coupling in general is one of the qualities of service-orientation that deviates from object-orientation. The use of inheritance and other object-orientation principles encourages a much tightly coupled relationship between units of processing logic when compared to the service-oriented design approach.
Service Abstraction	The object-orientation principle of abstraction requires that a class provide an interface to the external world and that it be accessible via this interface. Encapsulation supports this by establishing the concept of information hiding, where any logic within the class outside of what is exposed via the interface is not accessible to the external world. Service abstraction accomplishes much of the same as object abstraction and encapsulation. Its purpose is to hide the underlying details of the service so that the service contract is available and of concern to service requestors.
Service Composability	Object-orientation supports association concepts, such as aggregation and composition. These are supported by service orientation within a loosely coupled context. For example, the same way a hierarchy of objects can be composed, a hierarchy of services can be assembled through service composability.
Service Autonomy	The quality of autonomy is emphasized in service oriented design than it has been with object-oriented approaches. Achieving a level of independence between units of processing logic is possible through service orientation, by leveraging the loosely coupled relationship between services. Cross-object references and inheritance-related dependencies within object-oriented designs support a lower degree of object-level autonomy.
Service Statelessness	Objects consist of a combination of class and data and are naturally stateful. Promoting statelessness within services tends to deviate from typical object oriented design. Although it is possible to create stateful services and stateless objects, the principle of statelessness is emphasized with service-orientation.
Service Discoverability	Designing class interfaces to be consistent and self-descriptive is another object-orientation best practice that improves a means of identifying and distinguishing units of processing logic. These qualities support reuse by allowing classes to be discovered. Discoverability is another principle emphasized by the service-orientation paradigm. It is encouraged that service contracts be as communicative as possible to support discoverability at design time and runtime.

SUMMARY:

Several principles of service-orientation are related to and derived from object-orientation principles.

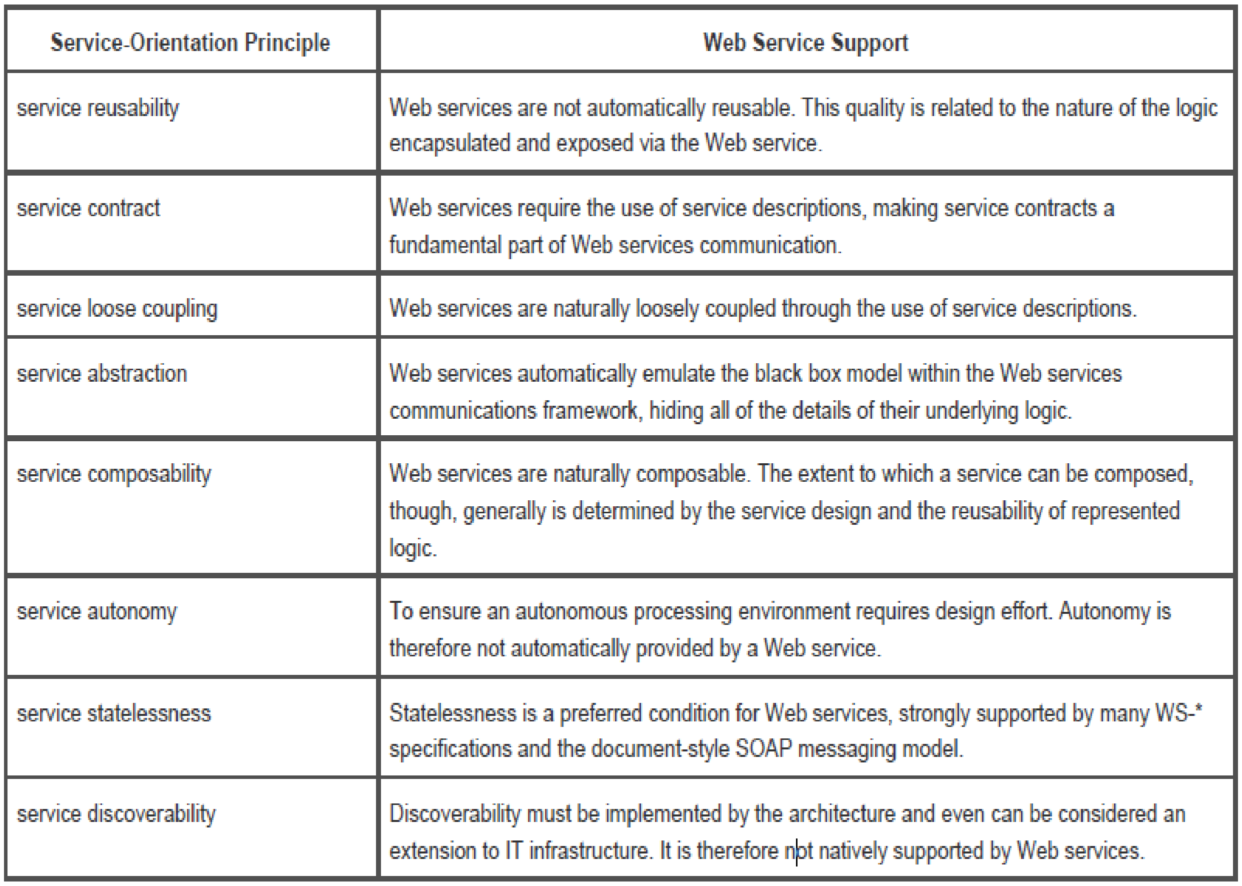
Some object-orientation principles, such as inheritance, do not fit into the service-oriented world.

Some service-orientation principles, such as loose coupling and autonomy, are not directly promoted by object-orientation.

XV.NATIVE WEB SERVICE SUPPORT FOR SERVICE-ORIENTATION PRINCIPLES

Web services provide inherent support for some of service-orientation principles. Table recaps the principles of service-orientation and explains to what extent they are natively supported by Web services.

A look at which service-orientation principles are automatically supported by Web services.



Service-Orientation Principle	Web Service Support
service reusability	Web services are not automatically reusable. This quality is related to the nature of the logic encapsulated and exposed via the Web service.
service contract	Web services require the use of service descriptions, making service contracts a fundamental part of Web services communication.
service loose coupling	Web services are naturally loosely coupled through the use of service descriptions.
service abstraction	Web services automatically emulate the black box model within the Web services communications framework, hiding all of the details of their underlying logic.
service composability	Web services are naturally composable. The extent to which a service can be composed, though, generally is determined by the service design and the reusability of represented logic.
service autonomy	To ensure an autonomous processing environment requires design effort. Autonomy is therefore not automatically provided by a Web service.
service statelessness	Statelessness is a preferred condition for Web services, strongly supported by many WS-* specifications and the document-style SOAP messaging model.
service discoverability	Discoverability must be implemented by the architecture and even can be considered an extension to IT infrastructure. It is therefore not natively supported by Web services.

Half of the principles of service-orientation are characteristics of Web services. This underlines the interaction of the marriage between SOA and the Web services technology platform and gives us a good indication as to why Web services have been so successful in realizing SOA.

It highlights the principles that require special attention when building service-oriented solutions. The four principles identified as *not* being automatically provided by Web services are service reusability, service autonomy, service statelessness and service discoverability.

Web services require the use of service contracts has no bearing on how well individual service descriptions are designed.

SUMMARY:

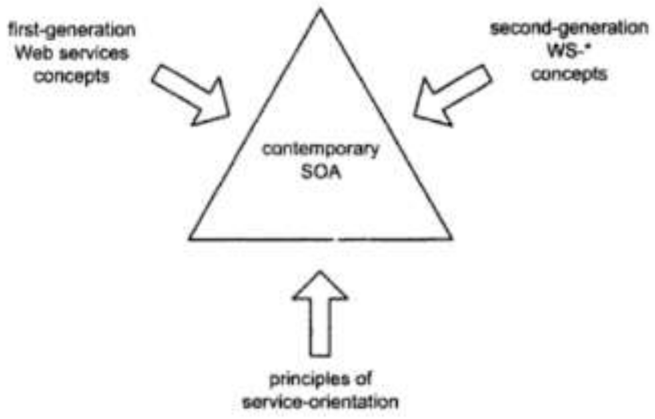
Service abstraction, composability, loose coupling, and the need for service contracts are native characteristics of Web services that are in full alignment with the corresponding principles of service-orientation.

Service reusability, autonomy, statelessness, and discoverability are not automatically provided by Web services. Realizing these qualities requires a conscious modeling and design effort.

XVI.SERVICE-ORIENTATION AND CONTEMPORARY SOA

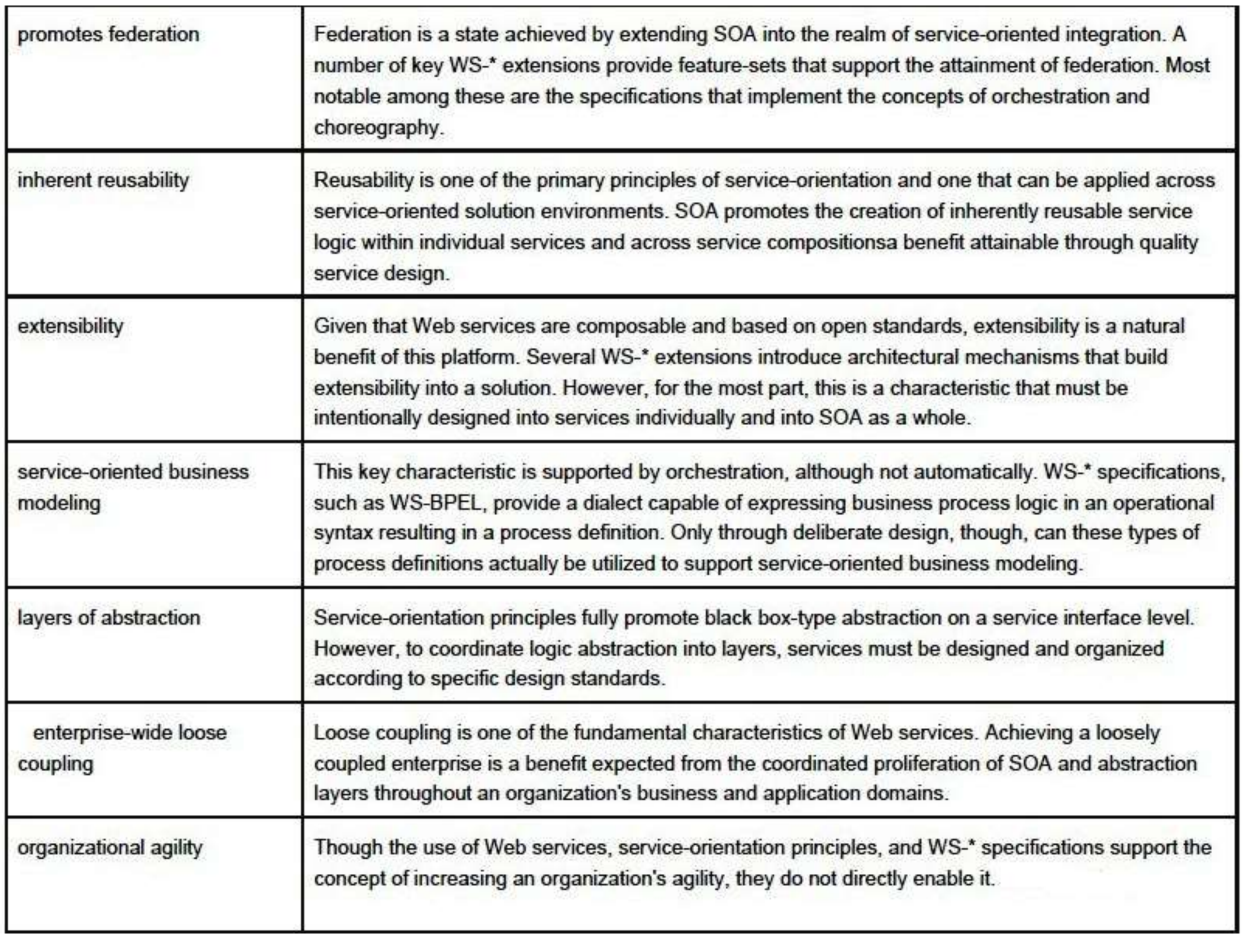
An approach to structuring and delivering specialized service layers around the delivery of key contemporary SOA characteristics are discussed.

Contemporary SOA is a complex and sophisticated architectural platform that offers solution to many historic and current IT problems. It achieves this by leveraging existing paradigms and technologies that support its overall vision.
The majority of what it has to offer can be harnessed thorough analysis and targeted design. Figure1 show three of the influences of contemporary SOA:
First-generation Web services concepts, second-generation (WS-*) Web Services concepts and principles of service-orientation.
Figure1. External influences that form and support contemporary SOA


first-generation Web services concepts
second-generation WS-* concepts
contemporary SOA
principles of service-orientation

1. Mapping the origins and supporting sources of concrete SOA characteristics
We identify those characteristics not supported by these external influences so that we can discuss how they can be realized. The WS-* specifications referenced in Table.
Table. Show a review of how contemporary SOA characteristics are influenced by Web services specifications and service-orientation principles.

Characteristic	Origin and/or Supporting Source
fundamentally autonomous	Autonomy is one of the core service-orientation principles that can be applied to numerous parts of SOA. Pursuing autonomy when building and assembling service logic supports other SOA characteristics.
based on open standards	This is a natural by-product of basing SOA on the Web services technology platform and its ever-growing collection of WS-* specifications. The majority of Web services specifications are open and vendor-neutral.
QoS capable	The quality of service improvements provided by contemporary SOA are, for the most part, realized via vendor implementations of individual WS-* extensions.
architecturally composable	While composability, on a service level, is one of our service-orientation principles, for an architecture to be considered composable requires that the technology from which the architecture is built support this notion. For the most part, the specifications that comprise the WS-* landscape fully enable architectural composability by allowing a given SOA to only implement extensions it actually requires.
vendor diversity	This is really more of a benefit of SOA than an actual characteristic. Regardless, it is primarily realized through the use of the open standards provided by the Web services platform.
intrinsic interoperability	The standardized communications framework provided by Web services establishes the potential to foster limitless interoperability between services. This is no big secret. To foster intrinsic interoperability among services, though, requires forethought and good design standards. Although supported by a number of WS-* specifications, this characteristic is not directly enabled by our identified influences.
discoverability	Service-level discoverability is one of our fundamental principles of service-orientation. Implementing discoverability on an SOA level typically requires the use of directory technologies, such as UDDI (one of the first-generation Web services specifications).



promotes federation	Federation is a state achieved by extending SOA into the realm of service-oriented integration. A number of key WS-* extensions provide feature-sets that support the attainment of federation. Most notable among these are the specifications that implement the concepts of orchestration and choreography.
inherent reusability	Reusability is one of the primary principles of service-orientation and one that can be applied across service-oriented solution environments. SOA promotes the creation of inherently reusable service logic within individual services and across service compositionsa benefit attainable through quality service design.
extensibility	Given that Web services are composable and based on open standards, extensibility is a natural benefit of this platform. Several WS-* extensions introduce architectural mechanisms that build extensibility into a solution. However, for the most part, this is a characteristic that must be intentionally designed into services individually and into SOA as a whole.
service-oriented business modeling	This key characteristic is supported by orchestration, although not automatically. WS-* specifications, such as WS-BPEL, provide a dialect capable of expressing business process logic in an operational syntax resulting in a process definition. Only through deliberate design, though, can these types of process definitions actually be utilized to support service-oriented business modeling.
layers of abstraction	Service-orientation principles fully promote black box-type abstraction on a service interface level. However, to coordinate logic abstraction into layers, services must be designed and organized according to specific design standards.
enterprise-wide loose coupling	Loose coupling is one of the fundamental characteristics of Web services. Achieving a loosely coupled enterprise is a benefit expected from the coordinated proliferation of SOA and abstraction layers throughout an organization's business and application domains.
organizational agility	Though the use of Web services, service-orientation principles, and WS-* specifications support the concept of increasing an organization's agility, they do not directly enable it.

The concrete characteristics associated with contemporary SOA represent the current state and expectations surrounding this architectural platform and are a reflection of the technology being developed in its support.

2. Unsupported SOA characteristics

Having removed the concrete SOA characteristics that receive support from our identified external influences, we left with the following six: intrinsic interoperability, extensibility, enterprise-wide loose coupling, service-oriented business modeling, organizational agility and layers of abstraction.

The first two are somewhat enabled by different WS-* extensions. , realizing these characteristics within SOA is a direct result of standardized, quality service design.

This leaves us with four remaining characteristics of contemporary SOA that are not directly supported or provided by the external influences we identified:

enterprise-wide loose coupling, support for service-oriented business modeling, organizational agility and layers of abstraction.

These four characteristics provide the crucial benefits of this architecture. This translates into extra up-front work that comes with the territory of building contemporary SOA.

Incorporating these key qualities into SOA requires that some very fundamental decisions be made, long before the building process of individual services can commence.

SOA around the creation of specialized service layers directly determines the extent to which these characteristics can be manifested.

SUMMARY:

The external influences that shape and affect contemporary SOA are first- and second-generation Web services specifications and the principles of service-orientation.

Many of the characteristics that define contemporary SOA are provided by these external influences.
Those characteristics not directly supplied by these influences must be realized through dedicated modeling and design effort.

XVII. THE PROBLEMS SOLVED BY LAYERING SERVICES AND LAYERS OF ABSTRACTION.

In enterprise model, the service interface layer is located between the business process and application layers.
This is where service connectivity resides and is the area of our enterprise wherein the characteristics of SOA are prevalent.
To implement the characteristics we identified in an effective manner, some larger issues need to be addressed. We provide some preliminary answers to the following questions.

What logic should be represented by services?

The enterprise logic can be divided into two domains: business logic and application logic. Services can be modeled to represent either or types of logic, as long as the principles of service-orientation can be applied.
To achieve enterprise-wide loose coupling physically separate layers of services are required. When individual collections of services represent corporate business logic and technology-specific application logic, each domain of the enterprise is freed of direct dependencies on the other.
This allows the automated representation of business process logic to evolve independently from the technology-level application logic responsible for its execution. This establishes a loosely coupled relationship between business and application logic.

How should services relate to existing application logic?

Much of this depends on whether existing legacy application logic needs to be exposed via services or whether new logic is being developed in support of services.
Existing systems can impose any number of constraints, limitations, and environmental requirements that need to be taken into consideration during service design.
Applying a service layer on top of legacy application environments may require that service-orientation principles be compromised.
When building solutions with service layers, as this affords a control with which service-orientation can be directly incorporated into application logic.
Services designed specifically to represent application logic should exist in a separate layer. This group of services belongs to the *application service layer*.

How can services best represent business logic?

Business logic is defined within an organization's business models and business processes. When modeling services to represent business logic, the service representation of this logic is in alignment with existing business models.
It is useful to separately categorize services that are designed in this manner. Services have been modeled to represent business logic as belonging to the *business service layer*.
By adding a business service layer, we implement the second of our four SOA characteristics, which is support for service-oriented business modeling.

How can services be built and positioned to promote agility?

The key to building an agile SOA is in minimizing the dependencies each service has within its own processing logic.
Services that contain business rules are required to enforce and act upon these rules at runtime. This limits the service's ability to be utilized outside of environments that require these rules.
Controller services that are embedded with the logic required to compose other services can develop dependencies on the composition structure.
Introducing a parent controller layer on top of specialized service layers would allow us to establish a centralized location for business rules and composition logic related to the sequence in which services are executed.

Orchestration is designed specifically for this purpose. It introduces the concept of a process service, capable of composing other services to complete a business process according to predefined workflow logic. Process services established refer to as the *orchestration service layer*.

While the addition of an orchestration service layer increases organizational agility, it is not in realizing this quality.

All forms of organized service abstraction contribute to establishing an agile enterprise, which means that the creation of separate application, business, and orchestration layers collectively fulfills this characteristic.

Abstraction is the key

One element to all of the answers happens to be the layers of abstraction.

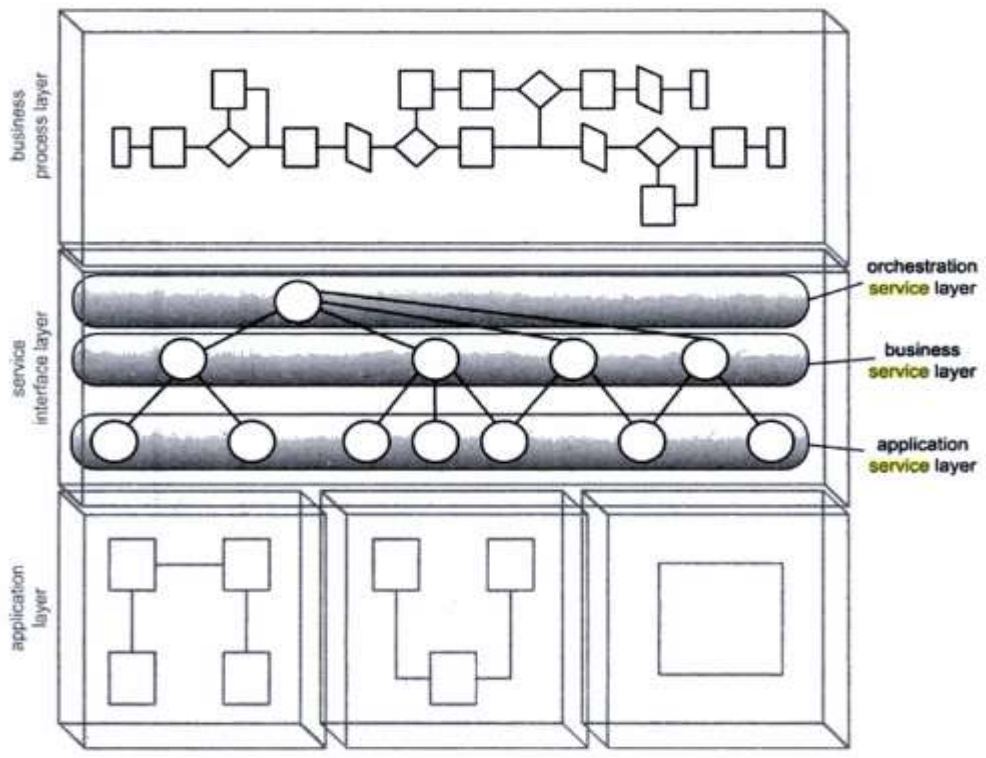
We have established how, by leveraging the composition, we can build specialized layers of services.

Each layer can abstract a specific aspect of our solution, addressing one of the issues we identified. This alleviates us from having to build services that accommodate business, application, and agility considerations all at once.

The three layers of abstraction as shown in Figure 1 we identified for SOA:

The application service layer, the business service layer and the orchestration service layer.

Figure 1. The three service layers



Summary

Through abstraction implemented via distinct service layers, key contemporary SOA characteristics can be realized increased organizational agility.

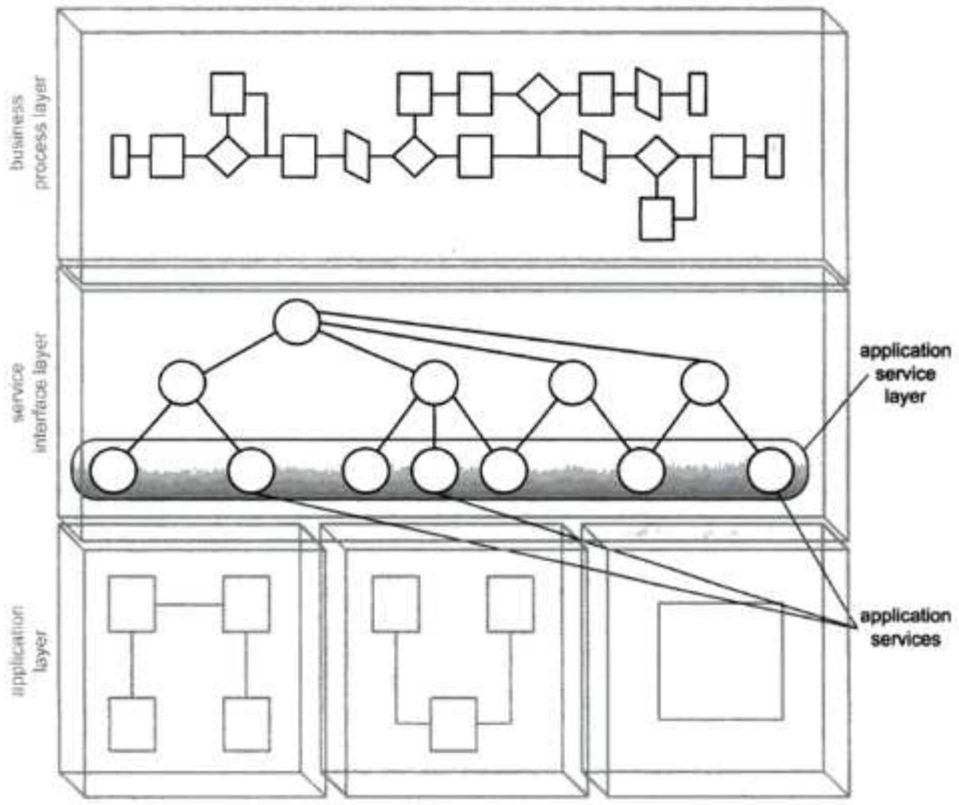
The three common layers of SOA are the application service layer, the business service layer, and the orchestration service layer.

A.Application service layer

The application service layer establishes the ground level foundation that exists to express technology-specific functionality.

Services that reside within this layer can be referred to as *application services* (Figure.2). Their purpose is to provide reusable functions related to processing data within new or legacy application environments.

Figure.2. The application service layer



Application services commonly have the following characteristics:

- they expose functionality within a specific processing context
- they draw upon available resources within a given platform
- they are solution-agnostic
- they are generic and reusable
- they can be used to achieve point-to-point integration with other application services
- they are inconsistent in terms of the interface granularity they expose
- they may consist of a mixture of custom-developed services and third-party services that have been purchased or leased

Examples of service models implemented as application services include the following: utility service and wrapper service.

When a separate business service layer exists, there is a strong motivation to turn all application services into generic utility services. This way they are implemented in a solution-agnostic manner, providing reusable operations that can be composed by business services to fulfill business-centric processing requirements.

If business logic does not reside in a separate layer, application services may be required to implement service models associated with the business service layer.

For example, a single application service can be classified as a business service if it interacts directly with application logic and contains embedded business rules.

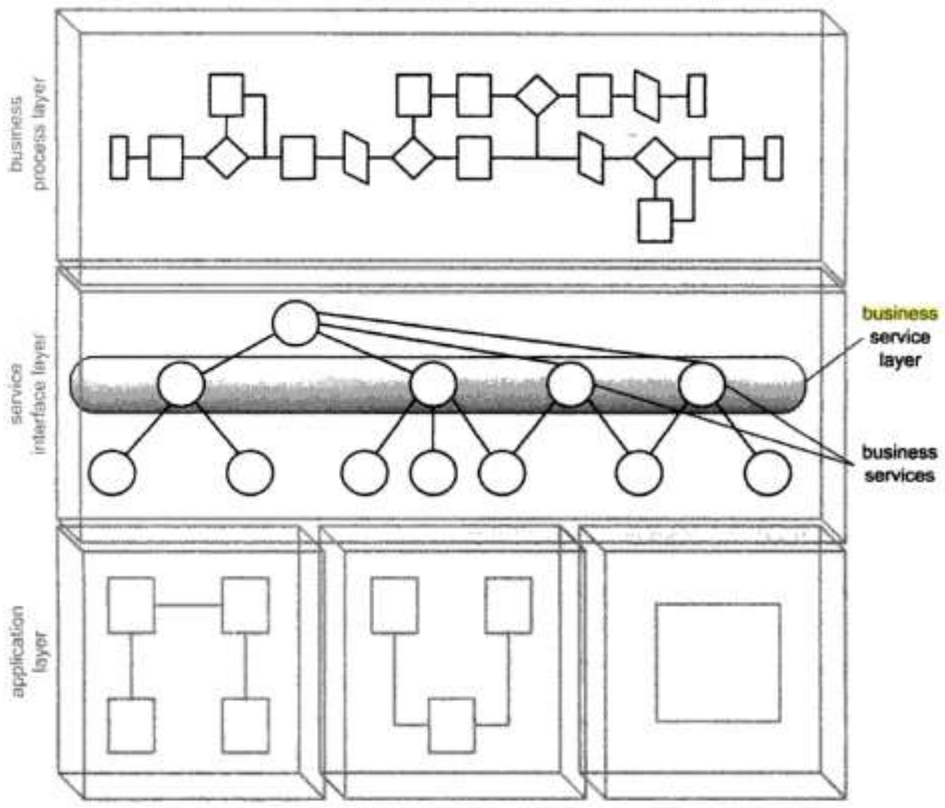
Services that contain application and business logic can be referred to as *hybrid application services* or *hybrid services*. This service model is commonly found within traditional distributed architectures.

It is not a recommended design when building service abstraction layers.

Finally, an application service can compose other, smaller-grained application services (such as proxy services) into a unit of coarse-grained application logic.

Aggregating application services is done to accommodate integration requirements.
Application services that exist solely to enable integration between systems are referred to as *application integration services* or *integration services*. Integration services are implemented as controllers.
Because they are residents of the application service layer, now is a good time to introduce the *wrapper service* model.
Wrapper services are utilized for integration purposes. They consist of services that encapsulate ("wrap") some or all parts of a legacy environment to expose legacy functionality to service requestors.
The frequent form of wrapper service is a service adapter provided by legacy vendors. This type of out-of-the-box Web service establishes a vendor-defined service interface that expresses an underlying API to legacy logic.
Another variation of the wrapper service model is the *proxy service,* also known as an *auto-generated WSDL*.
This provides a WSDL definition that mirrors an existing component interface. It establishes an endpoint on the component's behalf, allowing it to participate in SOAP communication.
The proxy service should not be confused with a *service proxy*, which is used by service requestors to contact service providers.
Summary:
The application service layer consists of application services that represent technology-specific logic.
Manifestations of application services are the utility and wrapper models.
Application services are reusable utility services composed by business services, but can exist as hybrid services that contain business and application logic.
B.Business service layer
While application services are responsible for representing technology and application logic, the business service layer introduces a service concerned with representing business logic, called the *business service* (Figure.3).
Business services are the lifeblood of contemporary SOA. They are responsible for expressing business logic through service-orientation and bring the representation of corporate business models into the Web services arena.
Application services can fall into different types of service model categories because they represent a group of services that express technology-specific functionality.
An application service can be a utility service, a wrapper service, or something else.
Business services are an implementation of the business service model.
The purpose of business services intended for a separate business service layer is to represent business logic in the purest form possible. This does not prevent them from implementing other service models. For example, a business service can be classified as a controller service and a utility service.
When application logic is abstracted into a separate application service layer, business services will act as controllers to compose available application services to execute their business logic.
Business service layer abstraction leads to the creation of two further business service models:
Task-centric business service: Service that encapsulates business logic specific to a task or business process. This type of service is required when business process logic is not centralized as part of an orchestration layer. Task-centric business services have limited reuse.
Entity-centric business service: Service that encapsulates a specific business entity (such as an invoice or timesheet). Entity-centric services are useful for creating highly reusable and business process-agnostic services that are composed by an orchestration layer or by a service layer consisting of task-centric business services.

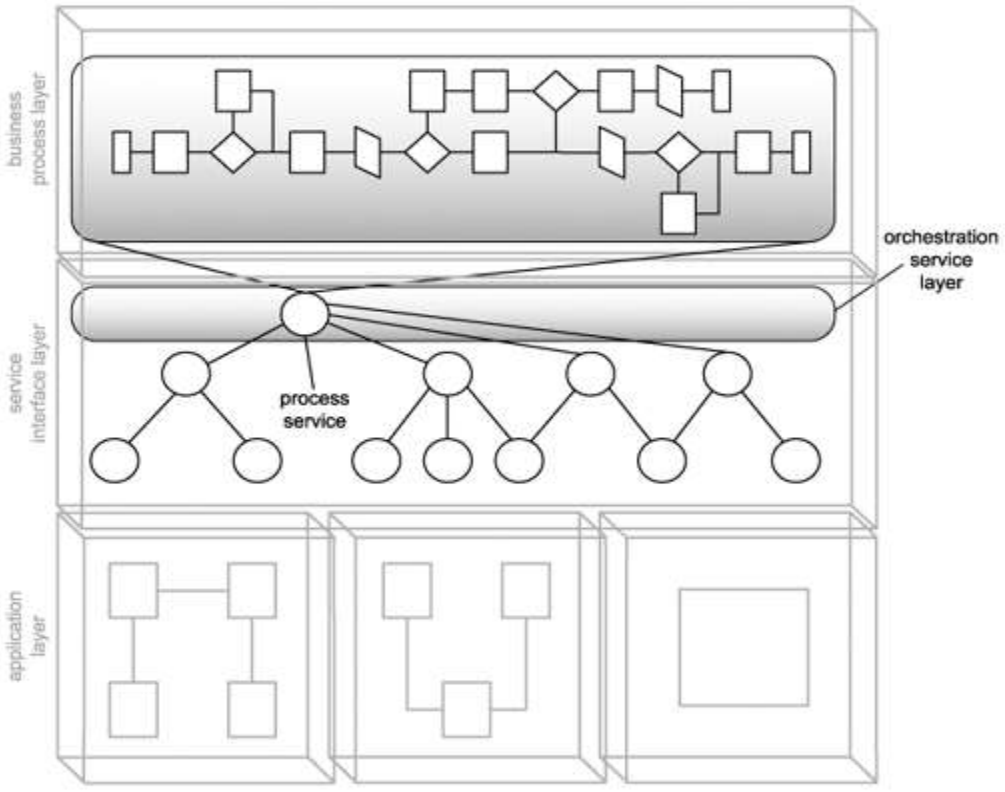
Figure 3. The business service layer



When a separate application service layer exists, these two types of business services can be positioned to compose application services to carry out their business logic.
The hybrid service is a service that contains business and application logic. It is referred to as a type of business service.
For the purpose of establishing specialized service layers, we consider the business service layer reserved for services that abstract business logic. We classify the hybrid service as a variation of an application service, making it a resident of the application service layer.
Summary:
The business service layer is comprised of business services, a direct implementation of the business service model.
Business services are controllers that compose application services to execute their business logic.
Even though hybrid services contain business logic, they are not classified as business services.
C.Orchestration service layer
When incorporated as part of a service-oriented architecture, orchestration assumes the role of the process part we established in the *Anatomy of a service-oriented architecture*.
Orchestration is valuable to us than a standard business process, as it allows us to directly link process logic to service interaction within workflow logic. This combines business process modeling with service-oriented modeling and design.
Because orchestration languages (such as WS-BPEL) realize workflow management through a process service model, orchestration brings the business process into the service layer, positioning it as a master composition controller.
The orchestration service layer introduces a parent level of abstraction that relieves the need for other services to manage interaction details required to ensure that service operations are executed in a specific sequence (Figure.4).

Within the orchestration service layer, *process services* compose other services that provide specific sets of functions, independent of the business rules and scenario-specific logic required to execute a process instance.

Figure.4. The orchestration service layer



All process services are controller services, as they are required to compose other services to execute business process logic.

Process services have become utility services to an extent, if a process, in its entirety, should be considered reusable. A process service that enables orchestration can be orchestrated.

The introduction of an orchestration layer brings with it the requirement to introduce new middleware into the IT infrastructure.

Orchestration servers are by no means a trivial addition and can impose expense and complexity.

Summary

The orchestration service layer consists of one or process services that compose business and application services according to business rules and business logic embedded within process definitions.

Orchestration abstracts business rules and service execution sequence logic from other services, promoting agility and reusability.

D.Agnostic services

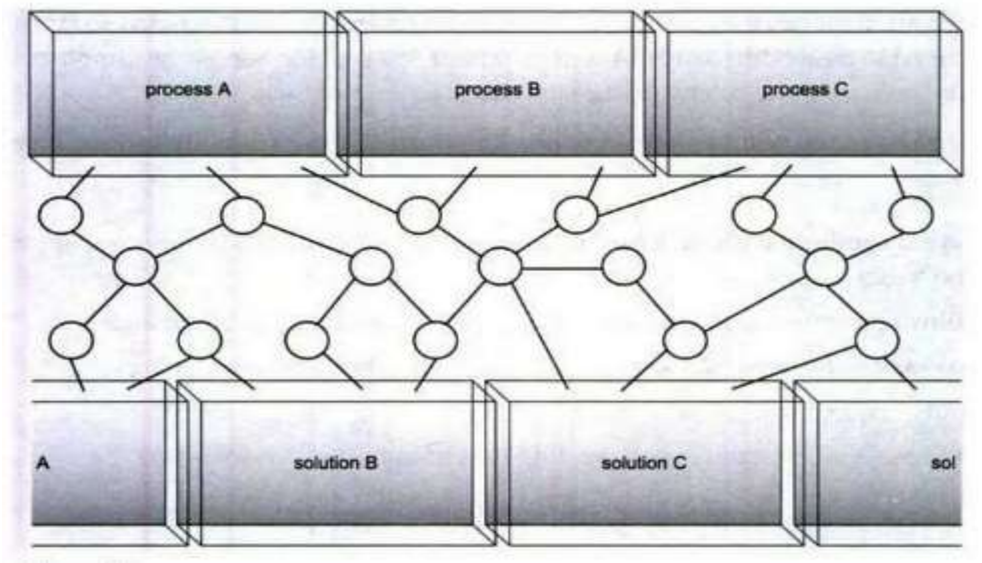
A key aspect of delivering reusable services is that they introduce service layers that are not limited to a single process or solution environment.

An application-level SOA containing solution-agnostic services does extend beyond the application.

Application-level SOA that depends on the use of existing solution-agnostic services does not have a well-defined application boundary.

Entity-centric business services are designed to provide a set of features that provide data management related to their corresponding entities. They are business process-agnostic. The same entity-centric business services can be reused by different process or task-centric business services.

Application services are built according to the utility service model. This makes them highly generic, reusable, and solution-agnostic.
Different service-oriented solutions can reuse the same application services.
As shown in Figure.5, services can be process-and solution-agnostic while being used as part of a service layer that connects different processes and solutions.
Figure.5. Services uniting previously isolated business processes and solution environments



If the services you are delivering collectively represent the logic of an entire solution, then the architectural scope is an application-level SOA.
If you are building services that extend an existing solution, then the architectural scope can vary. An enterprise that invests in agnostic services can end up with an environment in which a great deal of reuse is leveraged.
Building service-oriented solutions can become of a modeling exercise and less of an actual development project.
Summary:
Solution-agnostic service layers relate to and tie together multiple business processes and automation solutions.
These service layers promote reuse but blur the architectural boundaries of individual solutions.

XVIII. SERVICE LAYER CONFIGURATION SCENARIOS

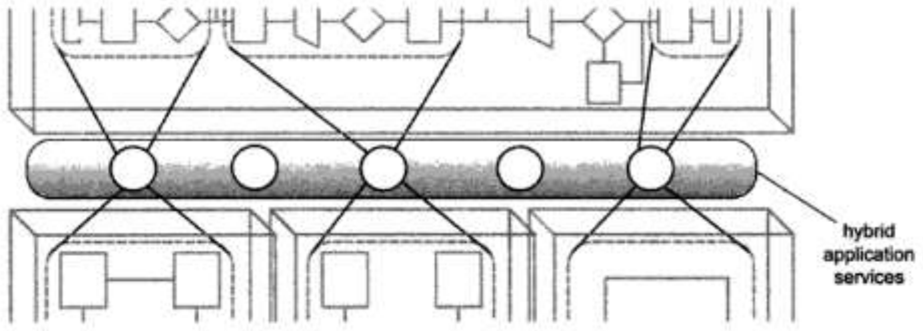
Service layer model for SOA is abstracted through three distinct layers that establish a well-defined composition hierarchy.
Many variations can exist, as the various types of services can be combined to create different SOA configurations. We provide here some sample configuration scenarios and discuss the characteristics of each.
To recap, we will explore scenarios based on the use of the following types of services:

- hybrid application services (services containing business process and application logic)
- utility application services (services containing reusable application logic)
- task-centric business services (services containing business process logic)
- entity-centric business services (services containing entity business logic)
- process services (services representing the orchestration service layer).

Scenario #1: Hybrid application services only
When Web services simply are appended to existing distributed application environments, or when a Web services-based solution is built without any emphasis on reuse or service-oriented

business modeling, the resulting architecture tends to consist of a set of hybrid application services (Figure.1).

Figure 1. Hybrid services encapsulating both business and application logic

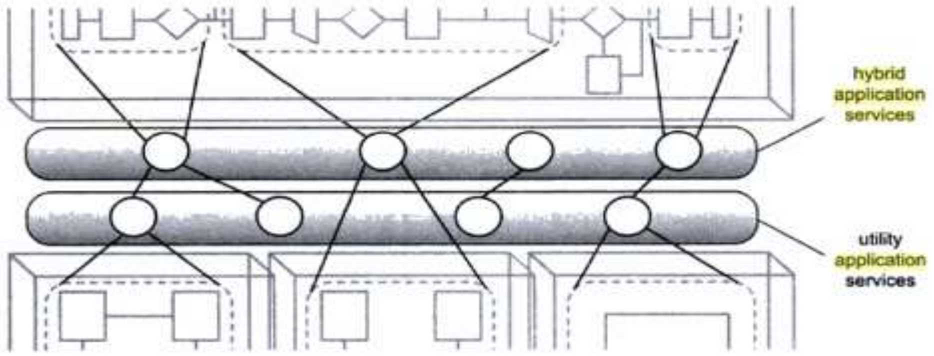


Scenario #2: Hybrid and utility application services

A variation of the previous configuration establishes a Web services-based architecture consisting of hybrid services and reusable application services.

The hybrid services may compose some of the reusable application services. This configuration achieves an extent of abstraction, as the utility services establish a solution-agnostic application layer (Figure.2).

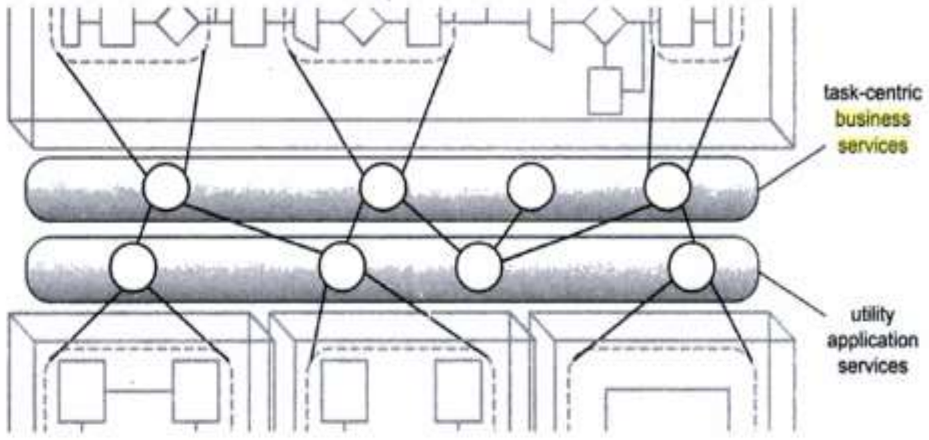
Figure 2. Hybrid services composing available utility application services



Scenario #3: Task-centric business services and utility application services

This approach results in a more coordinated level of abstraction, given that business process logic is entirely represented by a layer of task-centric business services. These services rely on a layer of application services for the execution of all their business logic (Figure.3).

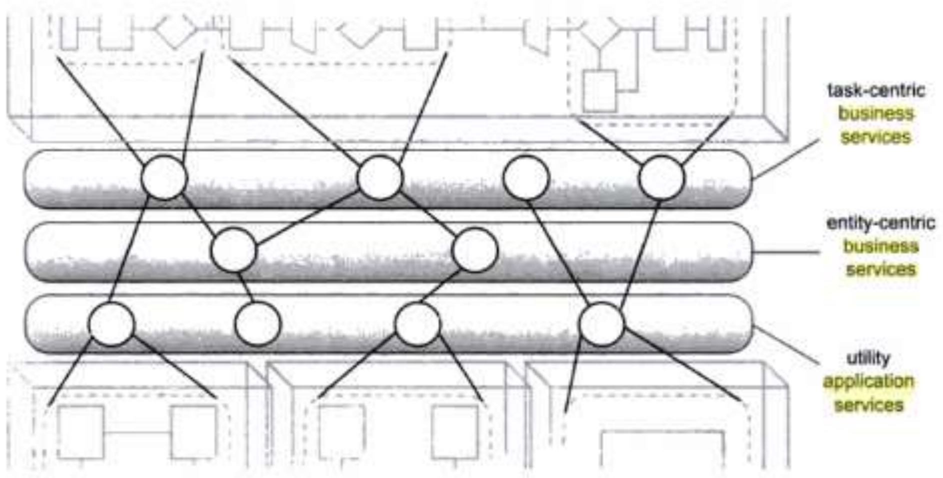
Figure 3. Task-centric business services and utility application services cleanly abstracting business and application logic



Scenario #4: Task-centric business services, entity-centric business services, and utility application services

We've added a layer of abstraction through the introduction of entity-centric business services. This positions task-centric services as parent controllers that may compose both entity-centric and application services to carry out business process logic (Figure.4).

Figure 4. Two types of business services composing application services

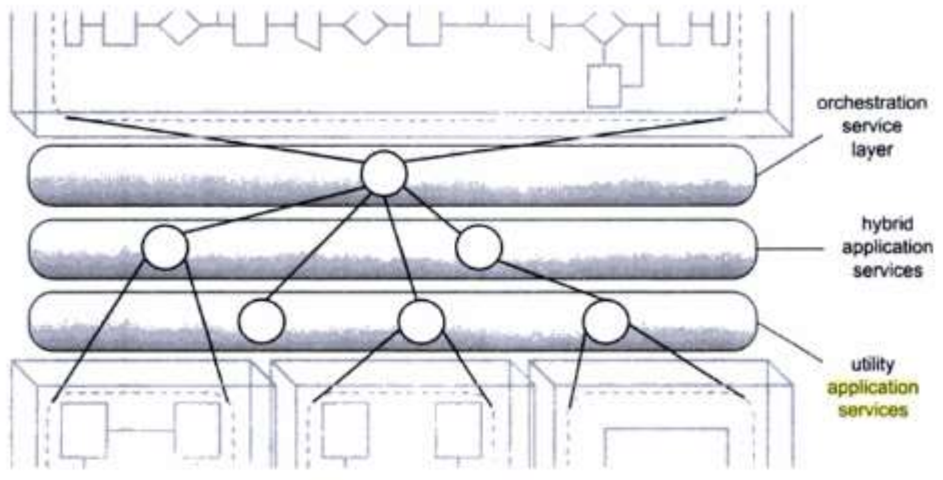


Scenario #5: Process services, hybrid application services, and utility application services

In this scenario, a parent process service composes available hybrid and application services to automate a business process.

This is a common configuration when adding an orchestration layer over the top of an older distributed computing architecture that uses Web services (Figure.5).

Figure 5. An orchestration layer providing a process service that composes different types of application services



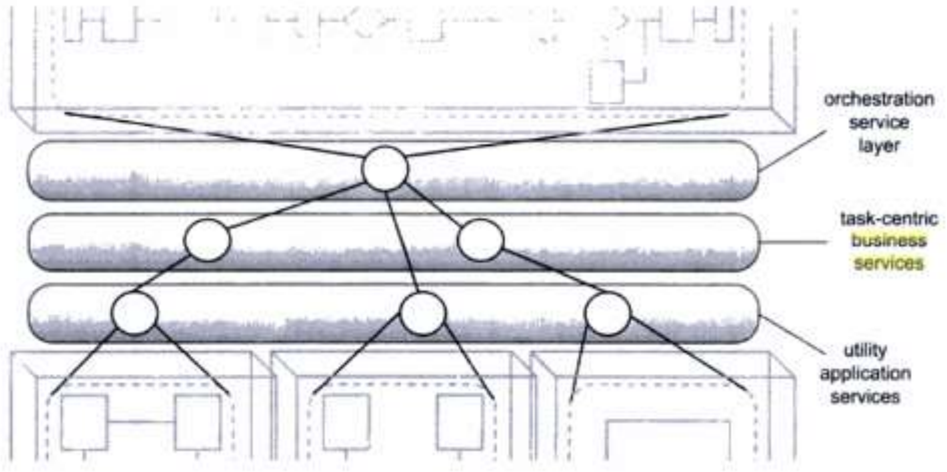
The hybrid service is being composed by the process service in Figure 5, the hybrid service still contains embedded business logic and therefore indirectly represents some of the business logic layer. Note also that the orchestration layer may compose utility application services directly.

Scenario #6: Process services, task-centric business services, and utility application services

Even though task-centric services also contain business process logic, a parent process service can still be positioned to compose task-centric and utility application services.

Though only partial abstraction is achieved via task-centric services, when combined with the centralized business process logic represented by the process services, business logic as a whole is abstracted (Figure.6).

Figure 6. A process service composing task-centric and utility application services.

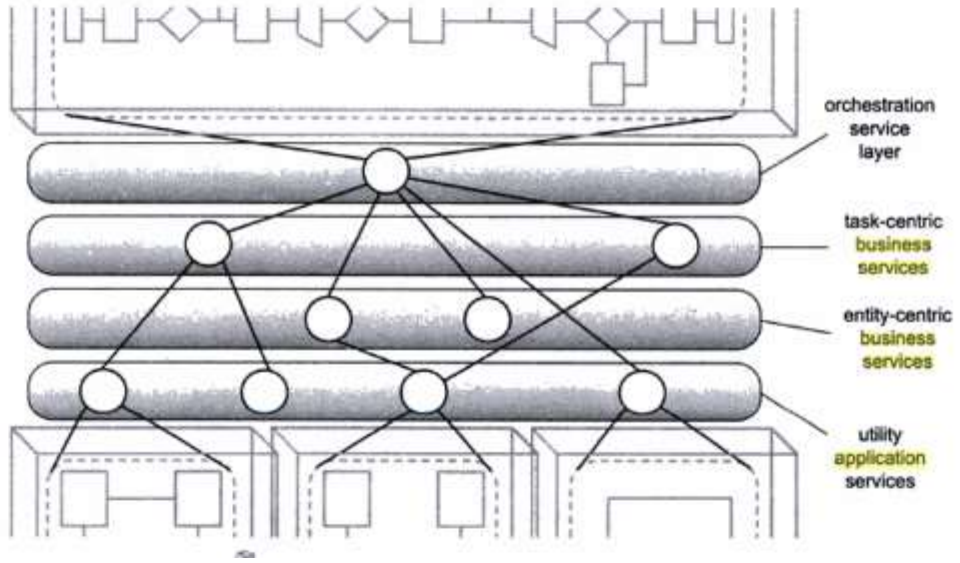

orchestration service layer
task-centric business services
utility application services

Scenario #7: Process services, task-centric business services, entity-centric business services, and utility application services

This variation also introduces entity-centric business services, which can result in a configuration that actually consists of four service layers.

Sub-processes can be composed by strategically positioned task-centric services, whereas the parent process is managed by the process service, which composes both task-centric and entity-centric services (Figure.7).

Figure 7. Process and business services composing utility application services

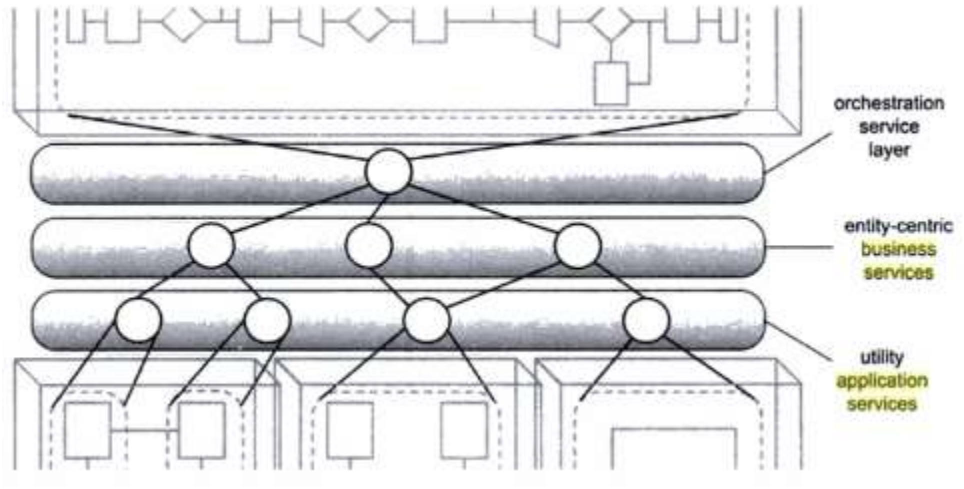

orchestration service layer
task-centric business services
entity-centric business services
utility application services

Scenario #8: Process services, entity-centric business services, and utility application services

This SOA model establishes a clean separation of business and application logic, while maximizing reuse through the utilization of solution and business process-agnostic service layers.

The process service contains all business process-specific logic and executes this logic through the involvement of available entity-centric business services, which compose utility application logic to carry out the required tasks (Figure.8).

Figure 8. A process service composing entity-centric services, which compose utility application services


orchestration service layer
entity-centric business services
utility application services

SUMMARY

SOAs are configured in different shapes and sizes, depending primarily on the types of services from which they are comprised.

Hybrid application services are found commonly when service-oriented environments include legacy distributed application logic.

By strategically positioning business and process services, an enterprise's business logic can be abstracted successfully from the underlying application logic.