Principles of Service Orientation

What is SOA as illustrated by what it is not

Version 1.2
March 21, 2010

Author:

Pras Biswas prasenjit_biswas@manulife.com

NOTE:

SOA is a design paradigm which has no universally accepted definitions or design standard compliance checklist. However, there is broad consensus on what SO goals are across the industry. Also note that we recognize that SOA is not always a panacea for every problem, but there is a broad consensus that its goals are worthwhile in a further evolution of IT to be 'business-ready'.

Purpose

Often times an abstract new concept is better illustrated by what it is not. Especially when many old times in the industry believe they have seen it all before, and that this new wave is "old wine in new bottles" or the best - "we are already doing SOA".

I have also listed SOA Anti-patterns with the intention of illustrating "**what as explained by what it should not be**"

SOA is emerging from the Trough of Disillusionment within Gartner's Hype Cycle, and it is climbing the Slope of Enlightenment. Enlightenment comes as mainstream organizations begin to experience the benefits of a technology and to establish the best practices necessary to effectively use a technology.

It is important to note that almost every SOA design characteristic we explore is attainable only to a certain measure. This means that it is generally not about whether solution logic does or does not have a certain characteristic; it is almost always about the extent to which a characteristic is realized.

SOA

We define SO as a collection of design principles in support of a design paradigm which aligns IT closer to business. Service-orientation emphasizes the creation of very specific design characteristics, while also de-emphasizing others. SO Design characteristics are:

- **Business alignment** of interfaces to business processes
- **Intrinsic interoperability** over custom integration
- **Shared services** over specific-purpose implementations
- **Flexibility** over optimization

SO Design Principles

1. **Contract First** Contracts are consistent, reliable, and governable, and promote independent design and evolution of a service's logic and implementation while still guaranteeing baseline interoperability with consumers. Services expose schema and contract, not class APIs

2. **Loose coupling** Service contracts impose low consumer coupling requirements and are themselves decoupled from their surrounding environment.

3. **Abstraction** Information about services is limited to what is published in service contracts – assumptions are not necessary. Completely shielded from implementation.

4. **Composability** Services are effective composition participants, irrespective of the size and complexity of the composition. Composition is the process by which services are combined to produce composite applications or composite services. A composite application consists of the aggregation of services to produce a corporate competency or enterprise process.

5. **Autonomy**. Protocol independence. Services are dynamically addressable through URIs, enabling their underlying locations and deployment topologies to change or evolve over time with little impact upon the service itself (this is also true of a service's communication channels). No assumptions about consumer's protocols or communications channels.

6. **Reusability** Services are enterprise resources with agnostic functional contexts - the logic behind the service should not be limited to the individual concerns of an activity within a single business process; it should address multiple activities within multiple enterprise-wide processes.

7. **Discoverability** Runtime service discovery through a service repository (or UDDI) and service description by communicative meta data through which services can be effectively discovered and interpreted can play an essential role in runtime reuse and online commerce.

8. **Statelessness** A service may have a context within its stateless execution, but it will not have an intermediary state waiting for an event or a call-back. The retention of state-related data must not extend beyond a request/response on a service. State management consumes a lot of resources, and this can affect the scalability and availability that are required for a reusable service.

SOA Anti-patterns

An anti-pattern in software is a design pattern that may be commonly used but is ineffective and/or counterproductive in practice. It is the wrong solution to a problem but may have caught on due to various reasons such as:

- **Misinterpretation** - lack of understanding of the differences between SOA and previous computing paradigms drives skeptical to claim that SOA is just a name for same old techniques. The result is opposition to the adoption of SOA even if such adoption benefits the business.

- **Bad practices** - take short-cuts and do this SOA stuff quickly and cheaply. Ignorance of the difference between SOA and Web Services and lack of understanding the need for good service modeling techniques.
- **Defensiveness** – a common trend where corporations respond "...but we are already doing SOA" but do not follow any commonly understood SOA principles or lack any SOA governance practices.

Here is our list of most commonly observed SOA Anti-Patterns.

1. **CRUDy interface** – Create Read, Update, Delete methods exposed as web services.

```
<WebMethod()> _
Public Sub Create( ByVal CompanyName As String, ByVal  ContactName As String, ByVal ContactTitle As String,  ByVal Address As String,
ByVal City As String, ByVal  State As String, ByVal Zip As String, ByVal Country As  String, ByVal Telephone As String, ByVal Fax As String)
End Sub

<WebMethod()> _
Public Function MoveNext() As Boolean
End Function

<WebMethod()> _
Public Function Current() As Object
End Function

<WebMethod()> _
Public Sub UpdateContactName( ByVal NewName as String)
End Sub

<WebMethod()> _
Public Function CommitChanges()
End Function
```

Symptoms & Consequences:

1.1. The interface design encourages RPC-like behavior, calling Create, MoveNext, and so on, instead of sending a well-defined message that dictates the action to be taken. This is a violation of the Contract First and Statelessness tenets.

1.2. CRUD operations are the wrong level of factoring for a Web service. CRUD operations may be implemented within or across services, but should not be exposed to consumers in such a fashion. This is an example of a service that allowed internal (private) capabilities to bleed into the service's public interface.
1.3. Interface is likely to be overly chatty, since consumers may need to call two or three methods to accomplish their work.
1.4. The interface implies stateful interactions such as enumeration (see the **MoveNext** and **Current** functions). Violation of Statelessness principle.
1.5. Abstract types (such as the Object returned by the **Current** function) result in a weak contract. Weak compliance to Contract First principle.
1.6. Using a **Create** operation without a return parameter means that the consumer will have no idea if the operation succeeds or fails. When designing a service always keep the consumer's expectation in mind—what does the consumer need to know?
1.7. This is a very dangerous service since it could leave the underlying data in an inconsistent state. What would happen if a consumer added a new Contact (or updated an existing Contact) and never called the **CommitChanges** function? As stated earlier, service providers cannot trust consumers to "do the right thing."

2. **Loosey Goosey** – loosely-typed methods accept strings and have no contract and frequently expose implementation

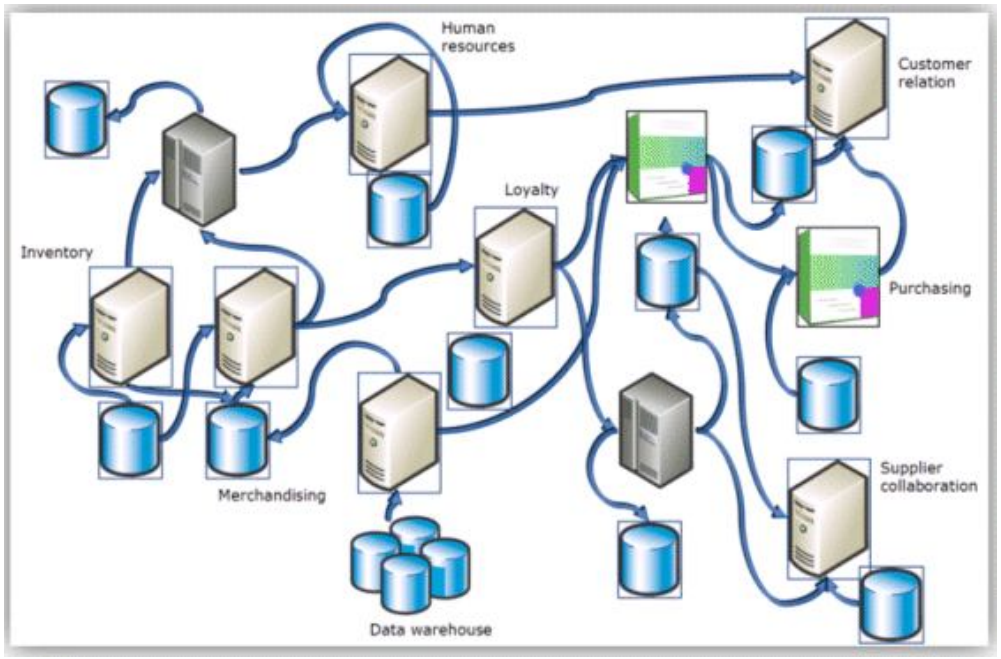
```
<WebMethod()> _
Public Function QueryDatabase (ByVal Database as String, SQLQuery as string) As DataSet

<WebMethod()> _
Public Function Execute(ByVal Command as Integer, Arguments as string) As Boolean
```

Symptoms & Consequences:

2.1. There is virtually no contract. A service consumer has no idea how to use the service (for example, what are valid Command arguments, encoding expectations, and so on).
2.2. The interface errs on the side of being too liberal in what it will accept. The contract is both unclear and a high security risk, susceptible to a SQL injection attack.
2.3. The contract does not provide enough information to consumers on how to use the service. If a consumer must read something other than the service's signature to understand how to use the service, the factoring of the service should be reviewed.

2.4. Consumers are expected to be familiar with the database and table structures prior to consuming the Web service. This results in a tight coupling between service providers and consumers.
2.5. Performance will suffer due to dependencies on late binding and encoding/decoding between boundaries within the same service.

3. **Point-to-Point** – In point-to-point integration, data flows directly from system to system over web services. Web Service point-to-point is STILL point to point; doing a bad practice over SOAP and XML doesn't make it better. Point-to-point integration starts simple, but becomes complicated as it gets larger. This anti-pattern is the strongest deterrent to the "BUS" design pattern where applications get on the "BUS" based on compliance with a contract or by adherence to a standard.



Symptoms & Consequences:

3.1. There is virtually no reusability.
3.2. No Discoverability or Composability

3.3. Lack of long term systems integration vision with emphasis on short term results.
3.4. An indication that this antipattern is being exercised is the use of XML or SOAP over HTTP between internal applications.
3.5. No implementation of an policy-based intelligent connector such as a Service Bus
3.6. Web Service calls are invoked directly, using a URI which is hard-coded into the WSDL. After a period of time the multiple web service calls and dependencies lead to the Spaghetti infrastructure that EAI aimed to solve. Having a highly disorganised and interdependent systems model leads to increased cost of change and a high degree of fragility of the enterprise. When one service is changed a number of other services either fail or behave in unpredictable ways. There is a lack of clarity as to how one service depends on another and what the impact of change is across these services. Consumers then start demanding multiple versions of services, and these add further to the spaghetti in the enterprise.

4. **Chatty Interfaces** – when a service is realized by implementing a number of Web Services where each communicates a tiny piece of data. This will result in the implementation of a large number of services leading to degradation in performance and costly development.

Symptoms & Consequences:

4.1. Violation of Loose coupling, Statelessness and Abstraction principles
4.2. The need to implement a large number of too finely grained services is an indication that this antipattern is being applied.
4.3. Degradation in performance and costly development are the major consequences of this antipattern. Additionally, consumers have to expend extra effort to aggregate these too finely grained services to realize any benefit as well as have the knowledge of how to use these services together.
4.4. An approach many developers take is to mimic API implementation in the form of Web services. This is very similar to the situation we encountered at the beginning of the object-oriented paradigm shift, where developers used object-oriented languages to develop procedural programs. Everything becomes a *web service* with no benefit and excessive cost.

5. **Web Services Equal SO** When architects equate SOA with Web services they run the risk of replacing existing APIs with Web services without proper architecture. This will result in identifying many services that are not business aligned. To many, SOA is just another name for Web services.

Symptoms & Consequences:

5.1. Violation of most SO principles except Autonomy, arising from migration to a different protocol rather than from design principles
5.2. Replacing existing APIs with Web services without proper architecture as well as implementation of services that are not business aligned are obvious symptoms of this antipattern.

6. **Component-less Services.** Web services are developed without any regard to the concepts of layering (n-tier approach) and separation of concerns – such as Business Logic Layer, Data Access Layer, Presentation Layer, etc. Web Services are not supported underneath by coherent n-tier architecture or flexible component-based layer.

Symptoms & Consequences:

6.1. Violation of most SO principles except Autonomy, arising from lack of good design
6.2. Inflexibility - modular design, information hiding and logical structuring principles are all violated

7. **Interface Bloat.** Over-specification of the interface where one interface ties to do too much. The net consequence of these conditions was that too much data is exchanged, and there is strong cohesion to the component layers. This is the opposite of the Chatty interface anti-pattern.

Symptoms & Consequences:

7.1. Violation of most SO principles like composability and loose coupling, arising from lack of good design
7.2. Additional query and data transfer time required.
7.3. Intensive processing was required to parse data in the Services layer and extract the sub-set of data needed for a specific operation.
7.4. Performance issues arose through slower response time due to marshaling and passing the data.

8. **Utilities as Methods.** Utilities like filling lists, class contractors, class overrides, UI methods, string manipulation and helper methods and all other kinds of code construction methods in the interface

Symptoms & Consequences:

8.1. Violation of most SO principles like loose coupling, abstraction

SOA References

- Oracle SOA Principles
- MSDN – Principle of SOA Design
- Thomas Erl