Service Oriented Architecture & Design Patterns

Jan Jusko

ATG, FEE, CTU

October 24, 2012

Goals of Service Oriented Computing

- increased intrinsic interoperability
  - services within a given boundary are designed to be naturally compatible; they can be effectively assembled and reconfigured if needed
  - e.g. same ontology
- increased federation
  - services have established uniform contract layer which hides the underlying complexity
  - in turn, they can be administered individually
- increased vendor diversification options
  - service-oriented environment is vendor-neutral
  - the architecture can evolve together with business without any vendor locks
- increased business & technology domain alignment
  - services are developed with business in mind
  - easy mirroring and evolution of services in accordance to business

Goals of Service Oriented Computing (II)

- increased return of investments,
  - services are to be reusable, thus reduction of costs
- increased organizational agility,
  - new requirements can be incorporated faster since services are designed with reusability in mind — can be augmented or reassembled
  - reduced delivery time
- reduced IT burden
  - single IT environment with agreed standards
  - particular parts are build to cooperate
  - reduced integration costs

Service-Orientation Design Principles

- set of principles that should be followed to make (enterprise) application service-oriented
- standardized service contract
  - services within one directory are in compliance with the contract design standard
- service loose coupling
  - services are loosely coupled with their clients (via contracts)
- service abstraction
  - services contain only necessary information in the contract
  - internal workings are hidden
  - eases the substitution of the service
- service reusability
  - services are "agnostic" and reusable
- service autonomy
  - services have high level of control over their underlying runtime environment

Service-Orientation Design Principles (II)

- service statelessness
  - services are stateless unless necessary
  - lowers resources consumption
- service discoverability
  - services publish meta-data based on which they can be discovered and interpreted
- service composability
  - services can be composed, regardless of the size or complexity of the composition

Physical Design Characteristics

- having goals and paradigms explained, the question is — what should be the physical design like
- any practical form, implementation of SOA should have the following properties:
  - *business-driven* — technology architecture is aligned with the business architecture; this is kept as the business evolves over time
  - *vendor-neutral* — the architecture model is not based on a single vendor technology, allowing different vendor technologies to be combined
  - *enterprise-centric* — scope of the architecture represents a reasonable part of the enterprise, enabling reuse
  - composition-centric — service aggregation is inherently supported thus allowing for rapid changes in service composition

Business-Driven

- traditionally, application in enterprises were developed on as-needed bases to fulfill short-term goals
- these weren't necessarily in accordance with the long term goals in the enterprise
- this resulted in a slow divergence of technology solutions from the business model
- in contrast to that, business-driven architecture uses long-term goals as a basis and inspiration for architectural model
- maximizes the overlap of technology and business in the long term

Vendor-Neutral

- relying on a technology provided by a single vendor can be inhibitive in the future when new requirement are to be met, but the current vendor architecture cannot support it
- it is thus important to design a SOA model that is independent of any vendor-specific architecture and neutral to all of them

Enterprise-Centric

- traditionally, applications, even the distributed ones, were designed only for small parts of the enterprise, never reused anywhere else
- when delivering service-oriented solution, services are positioned as enterprise-resources and have the following characteristics
  - logic is available beyond the current application boundary
  - logic is designed according to established design principles and enterprise standards

Composition-Centric

- services must not be only reusable, but also easily composable, i.e. able to participate in an aggregation, whether this aggregation was planned from the beginning or is an additional requirement
- this native composability must be supported by the underlying technology architecture

Architecture Types

- technology architecture can be of several forms
- service architecture
  - technology architecture limited to the physical design of a program designed as a service
- service composition architecture
  - the architecture of set of services assembled into a service composition
  - composition can be nested
- service directory architecture
  - the architecture that supports a collection of related services that are independently standardized and governed

Design Patterns

- a proven solution to a common problem individually documented in a consistent format and usually as part of a larger collection
- in general, they represent field-tested solutions to common design problems
- can be either simple or compound
- design patterns can be applied in variety of creative application sequences

Design Patterns Categories

- Service Inventory Design Patterns
- Service Design Pattern
- Service Composition Design Patterns

Enterprise Inventory

- **Problem** Delivering services independently establishes a risk of producing inconsistent service and architecture implementations, compromising recomposition opportunities
  - happens especially when there are several independent development efforts, each focusing on its own goals
  - might result in disparate service clusters and technology architectures
  - causes serious issues when attempting to compose services across various initial architectural boundaries
- **Solution** Standardized, enterprise-wide inventory architecture wherein services can be freely and repeatedly recomposed
  - services are designed specifically for implementation within the enterprise directory
  - ensures wide-spread standardization and intrinsic interoperability

Enterprise Inventory (II)

- **Application** Modeled in advance, enterprise-wide standards are applied
  - if the scope of the inventory is significant, the enterprise might not have resources for upfront modeling
  - depends on maturity of available technology, number of legacy environments, financial resources, cultural/political obstacles
  - recommended for small- to mid- enterprises with enough resources or tightly controlled IT environment; super-rich enterprises
- **Impacts** upfront analysis, organizational impacts
  - depending on the size of the inventory, the upfront analysis might be huge
  - instead of top-down-analysis, meet-in-the-middle approach can be chosen

Domain Inventory

- **Problem** Enterprise directory is unmanageable
  - enterprise directory might be impractical or even unrealistic for large enterprises
  - objections from e.g. data analysts, developers, business analyst
- **Solution** Grouping services into manageable domain-specific inventories, independent of each other

Domain Inventory (II)

- **Application** Inventory domain boundaries need to be carefully established
  - appropriate when global data models are unavailable, and creating them is impossible; the organization is not capable of changing their IT model;
  - specific inventory domains must be established in advance
  - can be based on department structure, geographic location, etc.
  - ideally, domains correspond to business domains
- **Impacts** Standardization disparity between domain service inventories imposes transformation requirements and reduces the benefit of the SOA adoption
  - various inventories can have different standards, inter-inventory service calls must undergo transformation — can complicate design efforts and performance

Service Normalization

- **Problem** When delivering services, there is a risk that services will be created with overlapping functionality, making reuse difficult
  - typical when multiple development teams are participating
  - leads to a denormalization of an inventory
  - results in inability to establishing official (unique) endpoints for various logic
  - services providing the same functionality can go out of sync - different results
- **Solution** The service inventory needs to be designed with an emphasis on service boundary alignment
  - services are collectively modeled before the implementation takes place

Service Normalization (II)

- **Application** Functional service boundaries are modeled as part of a formal analysis process
  - identifying and decomposing the business process within the domain
  - allocating individual parts of the process into appropriate services
  - making sure that service boundaries do not overlap
- **Impacts** Ensuring that service boundaries are and remain well-aligned introduces extra up-front analysis
  - requires that all processes are modeled before implementation
  - continual governance effort is required to ensure normalization after every services adjustment

Service Design Patterns

- these patterns in fact represent the most essential steps required to partition and organize solution logic into services and capabilities in support of subsequent composition
- *Service Identification Patterns* — The overall solution logic required to solve a given problem is first defined, and the parts of this logic suitable for service encapsulation are subsequently filtered out
  - Functional Decomposition
  - Service Encapsulation
- *Service Definition Patterns* – services are further partitioned into individual capabilities
- *Capability Composition Patterns* — services and their capabilities are further composed

Functional Decomposition

- **Problem** To solve a large, complex business problem a corresponding amount of solution logic needs to be created — self contained application
  - self contained monolithic applications are single-purpose, solid implementation boundaries
  - introduces challenges associated with extensibility and cross-application connectivity
  - many of such applications remained in modernized technical environment as legacy applications
- **Solution** The large business problem can be broken down into a set of smaller, related problems
  - functional decomposition is essentially a divide & conquer methodology
  - for each separated sub-part a solution logic can be built

Functional Decomposition (II)

- **Application** Service oriented analysis is used to decompose the large problem
  - service-oriented approach to functional decomposition differs from other distributed approaches in the manner in which separation is achieved
- **Impacts** The ownership of multiple smaller programs can result in increased design complexity
  - each sub-logic requires an individual attention to ensure the reliability, security, maintenance, etc.
  - effectiveness of this pattern is limited by the quality of the problem definition
  - if it is to be properly decomposed it needs to be documented in an accurate way with appropriate granularity

Service Encapsulation

- **Problem** Solution logic designed for a single application environment is typically limited in its potential to interoperate with other parts of an enterprise
  - logic decomposition only splits the original problem, however these subproblems remain restricted to the application domain
  - this introduces redundancy within the enterprise, inefficient application delivery, complex and expensive integration, etc.
- **Solution** Solution logic can be encapsulated by a service so that it is capable of functioning beyond the boundary for which it is initially delivered
  - sub-logic extracted from the process can be encapsulated as a service
  - the logic can serve as a basis for a new service or be incorporated into an existing service

Service Encapsulation (II)

- **Application** Solution logic suitable for service encapsulation needs to be identified
  - Does the logic contain functionality that is useful to parts of the enterprise outside of the immediate application boundary?
  - Does logic designed to leverage enterprise resources also have the potential to become an enterprise resource?
  - Does the implementation of the logic impose hard constraints that make it impractical or impossible to position the logic as an effective enterprise resource?
- **Impacts** None immediate

Redundant Implementation

- **Problem** A service that is being actively reused introduces a potential single point of failure
- **Solution** Reusable services can be deployed via redundant implementations
- **Application** The same service implementation is redundantly deployed or supported by infrastructure with redundancy features
  - different redundant implementation can be deployed for different sets of consumers
  - one service implementation is designated as official, and the underling technology architecture supports backups (load balancing)
- **Impact** Extra effort is required to keep all redundant implementations in sync

Service Facade

- **Problem** The coupling of the core service logic to contracts and implementation resources can inhibit its evolution
  - processing logic can evolve with time — if the processing logic changes its interface, so changes the service contract
  - one processing logic needs to support several contracts, thus needs additional decision logic for processing inputs from various clients
  - service functionality might be decomposed
- **Solution** A service facade component is used to abstract a part of the service architecture

Service Facade (II)

- **Application** A separate facade component is incorporated into the service design
  - it is responsible for providing supplemental/intermediate processing support
  - it is separated into a separate logic within the process
  - relying logic, broker logic, behavior correction
  - facade components can be placed in different places within the service architecture (including a separate service)
  - facade is typically tightly coupled with the contract and enables to processing logic do be loosely coupled
- **Impact** The addition of the facade component introduces design effort and performance overhead

Legacy Wrapper

- **Problem** Legacy application often require non-standard service contract with high technology coupling
  -
- **Solution** A wrapper service with standardized contract is created which eliminates legacy technical details
- **Application** A custom service contract and required service logic need to be implemented
  -
- **Impact** Added layer of processing with associated performance overhead