LECTURE 02

02

SOA Principles & Characteristics

The eight core service-orientation design principles that define what makes a service well-designed: loose coupling, reusability, autonomy, statelessness, and more.

Agenda

1. Overview of the service-orientation design principles
2. Standardized contract, abstraction, and loose coupling
3. Reusability, autonomy, and statelessness
4. Discoverability and composability
5. Service granularity and design trade-offs
6. Case study and common anti-patterns

The Eight Service-Orientation Design Principles

#	Principle	In One Line
1	Standardized Contract	Services follow an agreed, documented contract type
2	Loose Coupling	Minimize dependency between service and consumer
3	Abstraction	Hide logic from the outside world beyond the contract
4	Reusability	Design logic to be usable by multiple consumers
5	Autonomy	A service has control over its own logic and resources
6	Statelessness	Minimize resource consumption by deferring state
7	Discoverability	Services are described and findable
8	Composability	Services can be combined into larger solutions

DEFINITION

Standardized Service Contract

“Services within the same service inventory adhere to the same contract design standards, so consumers only need to learn one style of interface to work with many services.”

- The contract includes the functional interface (operations, messages) and non-functional policies (security, quality of service)
- Standardization applies across an organization's whole 'service inventory', not just one service

What Belongs in a Service Contract?

- **Functional description**
  - Operations offered, input/output message formats (e.g., defined in WSDL or an OpenAPI spec)
- **Data types and schemas**
  - Structure of the messages exchanged (e.g., defined in XML Schema — XSD)
- **Non-functional policies**
  - Security requirements, transactional behavior, performance/quality-of-service commitments
- A well-designed contract is technology-neutral — it should not leak details of the implementation language or platform

DEFINITION

Service Loose Coupling

“The relationship between a service and its consumers (and among services themselves) is designed to minimize dependency, so that changes on one side have minimal impact on the other.”

- Coupling is not binary — it is a spectrum from tightly coupled to loosely coupled
- The goal is not zero dependency (impossible) but manageable, well-defined dependency through the contract

Types of Coupling to Minimize

- **Technology coupling**
  - Consumer should not depend on the service's programming language or platform
- **Functional coupling**
  - Consumer should not depend on internal business logic details, only the published contract
- **Data model coupling**
  - Internal database schema changes should not break the consumer
- **Location coupling**
  - Consumer should find the service through a registry/endpoint reference, not a hard-coded address

Tightly Coupled vs. Loosely Coupled Integration

Tightly Coupled

- Consumer calls internal methods directly
- Shared code libraries between systems
- Hard-coded network addresses
- A schema change breaks all consumers
- Consumer must match service's language/platform

Loosely Coupled

- Consumer calls only the published contract
- No shared internal code, only the interface
- Location resolved via registry/discovery
- Contract versioning absorbs schema changes
- Any platform can consume the service

DEFINITION

Service Abstraction

“A service exposes only what is described in its contract; all other information about the service — its logic, technology, and internal data — remains hidden from the consumer.”

- Abstraction is what makes loose coupling possible — you cannot depend on what you cannot see
- Related to the general software-engineering principle of encapsulation, applied at the service level

What Should Be Hidden vs. Exposed?

- **Exposed (in the contract)**
  - Operation names, input/output message formats, security requirements
- **Hidden (implementation detail)**
  - Programming language, internal algorithms, database structure, internal error-handling logic, third-party libraries used
- A common mistake: leaking internal database column names or internal error codes into the service's response messages
- Rule of thumb: if changing it internally would force every consumer to change their code too, it should not be exposed

DEFINITION

Service Reusability

“A service is designed as an agnostic, generic unit of logic that can be reused across multiple applications and business processes, rather than being built for a single consumer.”

- Reusability is a design intent — it requires deliberately generalizing logic, not just accidentally reusing code
- Return on investment increases every time a service is reused by a new consumer without modification

Designing Services for Reuse

- **Keep logic agnostic to any single business process**
  - e.g., design 'ValidateAddress' generically, not only for the 'Shipping' module
- **Avoid consumer-specific parameters and behavior branches**
- Separate reusable, generic 'utility' logic from process-specific 'orchestration' logic
- Reusable services are typically coarser-grained utility or entity services, not one-off process steps
- Document services well so future teams discover and reuse them, instead of re-building the same logic

DEFINITION

Service Autonomy

"A service has control and authority over its own runtime environment and logic; its behavior is predictable and not dependent on being controlled by external resources it does not own."

- Autonomy allows a service to be independently deployed, versioned, and scaled
- The more control a service has over its own resources, the more reliable and predictable it becomes for consumers

Levels of Service Autonomy

- **Runtime autonomy**
  - Control over the service's own execution environment (its own process, container, or server)
- **Design-time autonomy**
  - Control over how the service's logic and contract evolve, without depending on another team's release schedule
- **Common obstacles to autonomy:**
  - Shared databases between multiple services
  - Shared application servers with resource contention
  - Tightly synchronized deployment schedules across teams

DEFINITION

Service Statelessness

“A service minimizes the amount of state information it holds between requests, deferring or delegating state management to avoid consuming resources unnecessarily.”

- A stateless service treats each request independently, without relying on memory of past interactions
- This makes services easier to scale horizontally — any instance can serve any request

Why Statelessness Matters — and How to Achieve It

- **A stateful service holding session data in memory cannot easily be load-balanced across multiple instances**
- **Strategies to defer state:**
  - Store session/context data in a database or distributed cache, not in the service's memory
  - Pass all necessary context in the request message itself (e.g., a token, a full order object)
  - Use the consumer or a dedicated state-management service to carry state between calls
- Complete statelessness is not always possible (e.g., long-running business transactions), but it is the default goal

DEFINITION

Service Discoverability

“Services are supplemented with meaningful metadata so that they can be effectively discovered and interpreted by consumers, both at design time and at runtime.”

- Discoverability is what makes the 'Find' step of publish-find-bind possible
- Poor documentation is one of the most common reasons a perfectly reusable service is never actually reused

Making Services Discoverable

- **Publish rich metadata: description, owner, version, sample requests/responses**
- **Register the service in a central registry/catalog (e.g., UDDI historically, or a modern API catalog/API gateway today)**
- Use clear, descriptive naming conventions across the service inventory
- Provide human-readable documentation alongside the machine-readable contract
- Tag services by business domain to make browsing and searching effective

DEFINITION

Service Composability

“A service is designed so it can be effectively used as a member of more than one composition — combined with other services to build larger business processes.”

- Composability depends on the other principles: a service that is not loosely coupled, autonomous, or standardized is hard to compose reliably
- Composed services are coordinated through orchestration or choreography — the subject of Lecture 8

What Makes a Service Composition-Friendly?

- **Predictable, well-documented behavior (no hidden side effects)**
- **Reasonable performance and reliability under repeated calls**
- Clear error-handling and fault contracts, so a calling process can react appropriately
- Idempotency where possible — calling an operation twice should not cause unintended duplicate effects
- A composition is only as reliable as its least reliable member service

How the Principles Reinforce Each Other

Composability	**Built on top of everything below**
Reusability & Discoverability	**Depend on a clear, published contract**
Loose Coupling & Abstraction	**Depend on hiding implementation detail**
Autonomy & Statelessness	**Foundation: control over own logic and state**

Key Insight

These principles are not independent checkboxes — they form a stack. A service cannot be truly reusable or composable if it is not first autonomous, stateless, loosely coupled, and well abstracted.

Service Granularity

- **Granularity refers to the scope and size of functionality a service exposes**
- **Fine-grained service**
  - Exposes a small, specific operation (e.g., 'ValidateZipCode')
- **Coarse-grained service**
  - Exposes a broader business capability (e.g., 'ProcessOrder', which internally validates, prices, and confirms)
- Granularity is a design decision with real trade-offs in performance, reusability, and complexity

Fine-Grained vs. Coarse-Grained Services

Fine-Grained	Coarse-Grained
• Highly reusable, flexible building blocks • More network round-trips for a full task • More design and governance overhead • Good for utility/entity-level operations	• Fewer network calls, better performance • Less flexible — harder to reuse partially • Simpler for consumers to call • Good for exposing whole business processes

Common Service Layers by Granularity

Layer	Granularity	Example
Utility Services	Fine-grained, technology-focused	LogEvent, ValidateEmailFormat
Entity Services	Medium, data/domain-focused	CustomerService, ProductService
Business/Task Services	Coarse-grained, process-focused	ProcessOrder, ApproveLoan
Orchestration Services	Coarsest — composes other services	OrderFulfillmentProcess

Design Trade-offs Among the Principles

- **Reusability vs. Autonomy**
  - A highly generic, reusable service may need to support many use cases, complicating its own internal logic
- **Statelessness vs. Performance**
  - Fetching context on every call (instead of holding it in memory) can add latency
- **Granularity vs. Network Overhead**
  - Fine-grained services increase reusability but increase the number of network calls needed
- Good SOA design is about making conscious, documented trade-offs — not blindly maximizing every principle

Case Study: Designing an 'Order Service' for E-Commerce

- **Standardized contract:**
  - Define OrderService.wsdl with clear CreateOrder, GetOrderStatus, CancelOrder operations
- **Loose coupling & abstraction:**
  - Internal order database schema can change without affecting consumers
- **Autonomy & statelessness:**
  - Order state is persisted in a database, not held in the service's memory between calls
- **Composability:**
  - OrderService is composed with PaymentService and ShippingService in a larger checkout process

Common Anti-Patterns in SOA Design

- **The 'God Service'**
  - One service tries to do everything, violating autonomy and reusability
- **Chatty Interfaces**
  - Too many fine-grained calls needed to complete a simple task, hurting performance
- **Leaky Abstraction**
  - Internal database or error details exposed directly in the service contract
- **Hidden Data Coupling**
  - Two services silently sharing the same database table, breaking autonomy

Key Takeaways

1. The eight service-orientation principles define what makes a service well-designed
2. Loose coupling and abstraction work together to minimize dependency between services and consumers
3. Reusability, autonomy, and statelessness make services scalable and broadly usable
4. Discoverability and composability enable services to be found and combined into larger solutions
5. Granularity decisions create real trade-offs between reusability and performance

DISCUSSION / REVIEW QUESTION

Your team is designing a 'ShippingCostCalculator' service used by both the web checkout and the mobile app. One developer wants to hard-code the tax rate table inside the service's response format. Which principles does this violate, and how would you redesign it?

Think about:

- Think about abstraction — what should the contract expose vs. hide
- Consider what happens when tax rates change next year
- Think about reusability across different consumers with different needs

References & Further Reading

- Erl, T. — SOA Principles of Service Design
- Erl, T. — Service-Oriented Architecture: Concepts, Technology, and Design, Ch. 4–5
- OASIS Reference Model for SOA — Section on Service Description

NEXT LECTURE

Lecture 3 — Web Services Fundamentals: XML, WSDL, and SOAP Basics