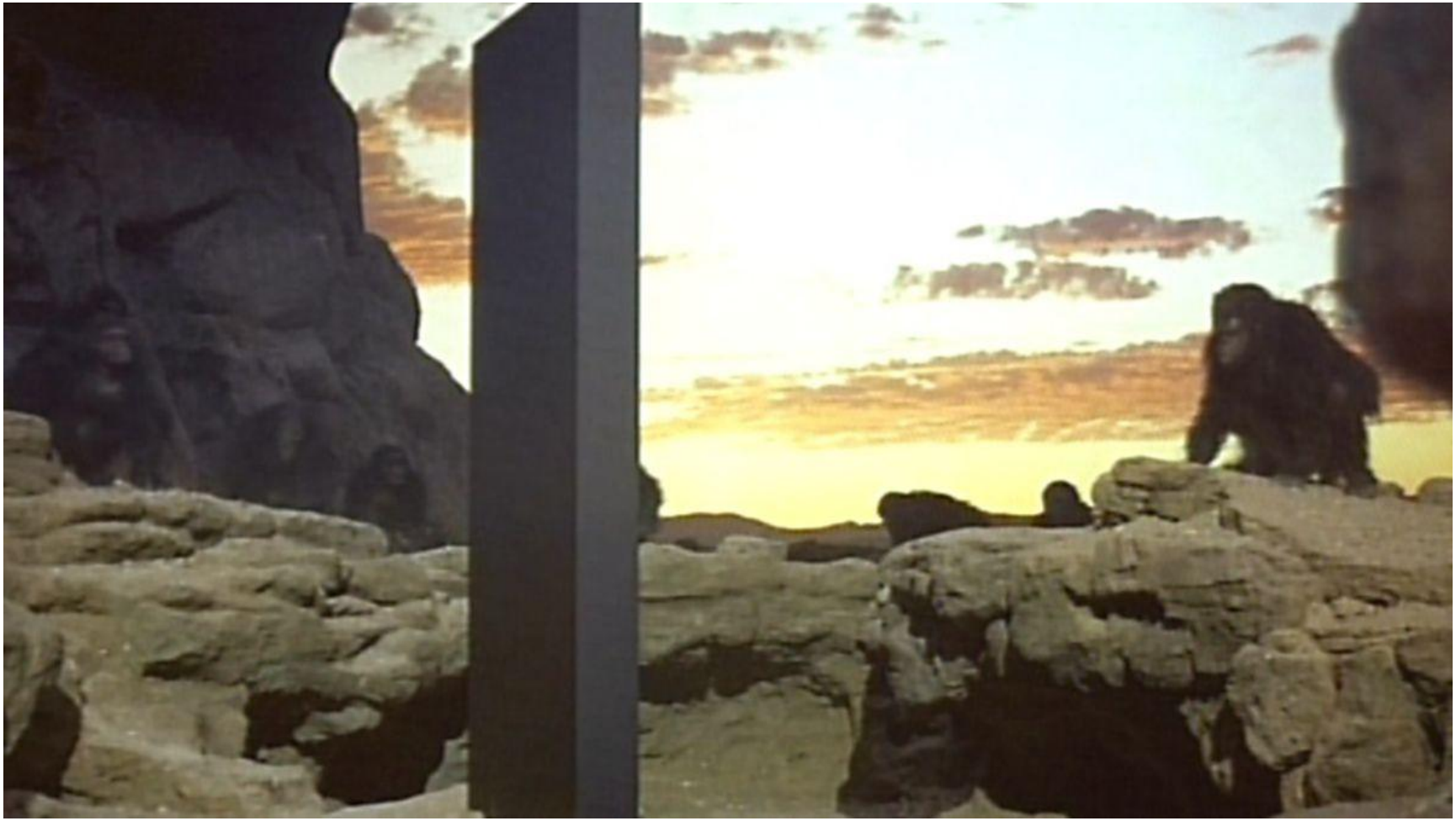
Architectural Style: Service-Oriented Architectures

CSCE 742 - Lecture 22 - 11/29/2018

In the Beginning...



Early Business Software

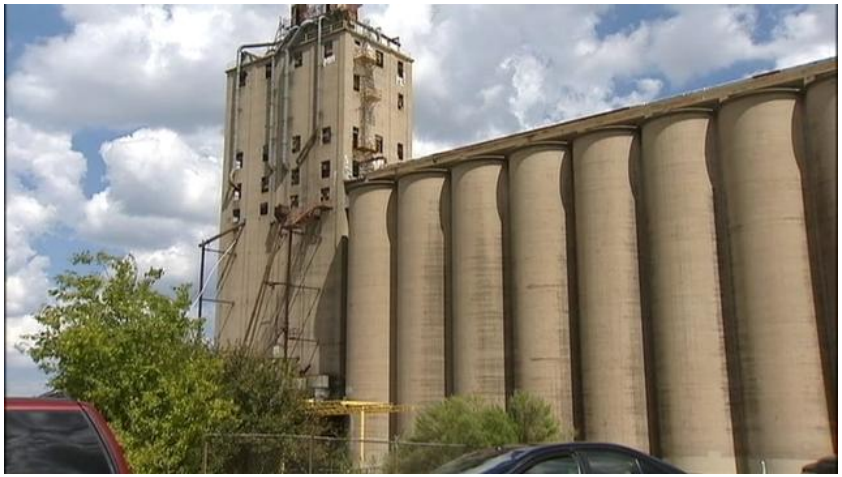
- Business and Government discovered the value of computing.
- Business requirements were captured and programmed.
- Applications were designed for specific departments / business needs.
- **Applications were monolithic.**
  - Designed as one entity, combining the logic of user interface, business processing, and data access.

Problem: “Silo Apps”

- Each application is self contained.
- One view of user interaction.
  - Difficult to find clean integration points
- Because of monolithic design, updates of one kind of logic require testing multiple kinds of behavior.
- Monolithic applications are harder to understand, as logic is generally patched rather than rewritten.
  - Rewrites are risky, “house of cards” effect.

Problem: “Silo Apps”

- Lack of standards makes it difficult to integrate with or to other applications.
- Leads to duplication:
  - If we don’t plan for reuse, reuse will not happen.
  - Applications contain nearly-duplicate functionality:
    - Authentication, business logic, storage management, logging.
  - Business units have nearly-duplicate applications.



The Times, They Are Changing.

- Reuse of existing software assets.
- Integration between separately developed business applications.
  - …Using different languages.
  - …Using heterogeneous hardware.
- Easily support corporate change:
  - Mergers / acquisitions.
  - Reorganization.
  - AKA - can we continue to support and use this application after employee turnover.

Services and Service-Oriented Architecture (SOA)

What is a Service?

- From the dictionary:
  - A facility supplying some public demand.
  - The work performed by one that serves.
    - See also: **help, use, benefit**
- In economics, a service is **the non-material equivalent of a good**.
  - Service provision is an economic activity that does not result in ownership.
  - It is claimed to be **a process that creates benefits** by facilitating either a change in customers, a change in their physical possessions, or a change in their intangible assets.

What is a Service?

- A service handles a business process, a technical task, or provides business data.
  - Process: Calculating an insurance quote.
  - Task: Accessing a database.
  - Data: Details needed to construct a GUI.
- A service can access another service and respond to different kinds of requesters.
- A service is relatively independent.
  - Changes to a requester require few or no changes to the service.
  - Changes to the internal logic of a service require few or no changes to the requester.

Properties of Services

- A service logically represents a business activity with a specified outcome.
- A service is self-contained.
  - Designed to maintain **loose coupling**.
- A service is a black box for its consumers.
  - Only its interface needs to be understood.
  - Can handle interactions within and outside your company, geographically distributed across the world.
- A service may consist of other underlying services.

Service-Oriented Architecture (SOA)

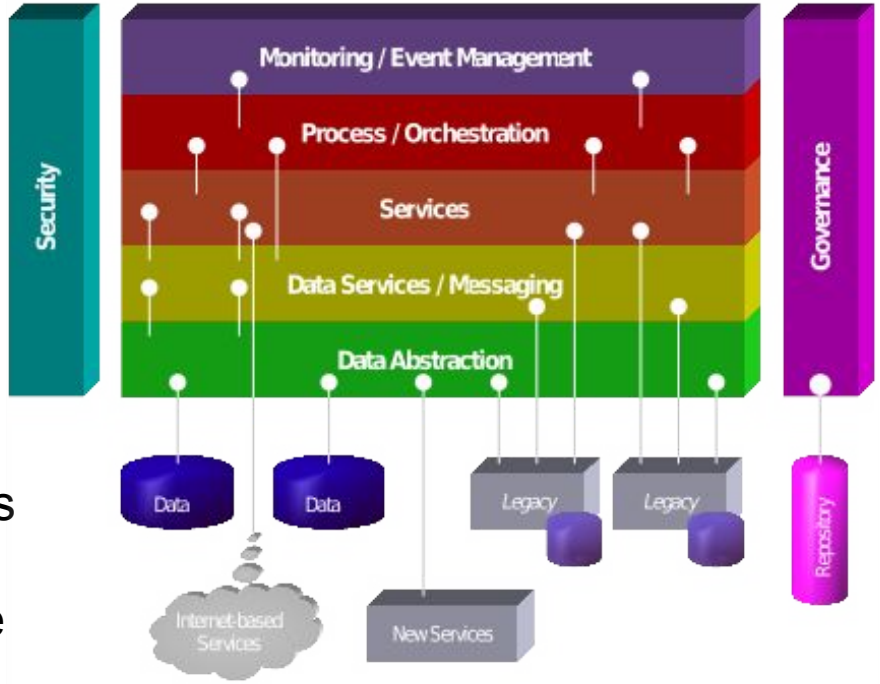
- A way of organizing software so that companies can respond quickly to the changing requirements of the marketplace.
- The architecture of a system links services.
  - Small, customized units of software that run in a network.
  - Developers make services available over a network to allow users to combine and reuse them.
  - Services communicate by passing data in a well-defined, shared format, or by coordinating activity between other services.

SOA Manifesto

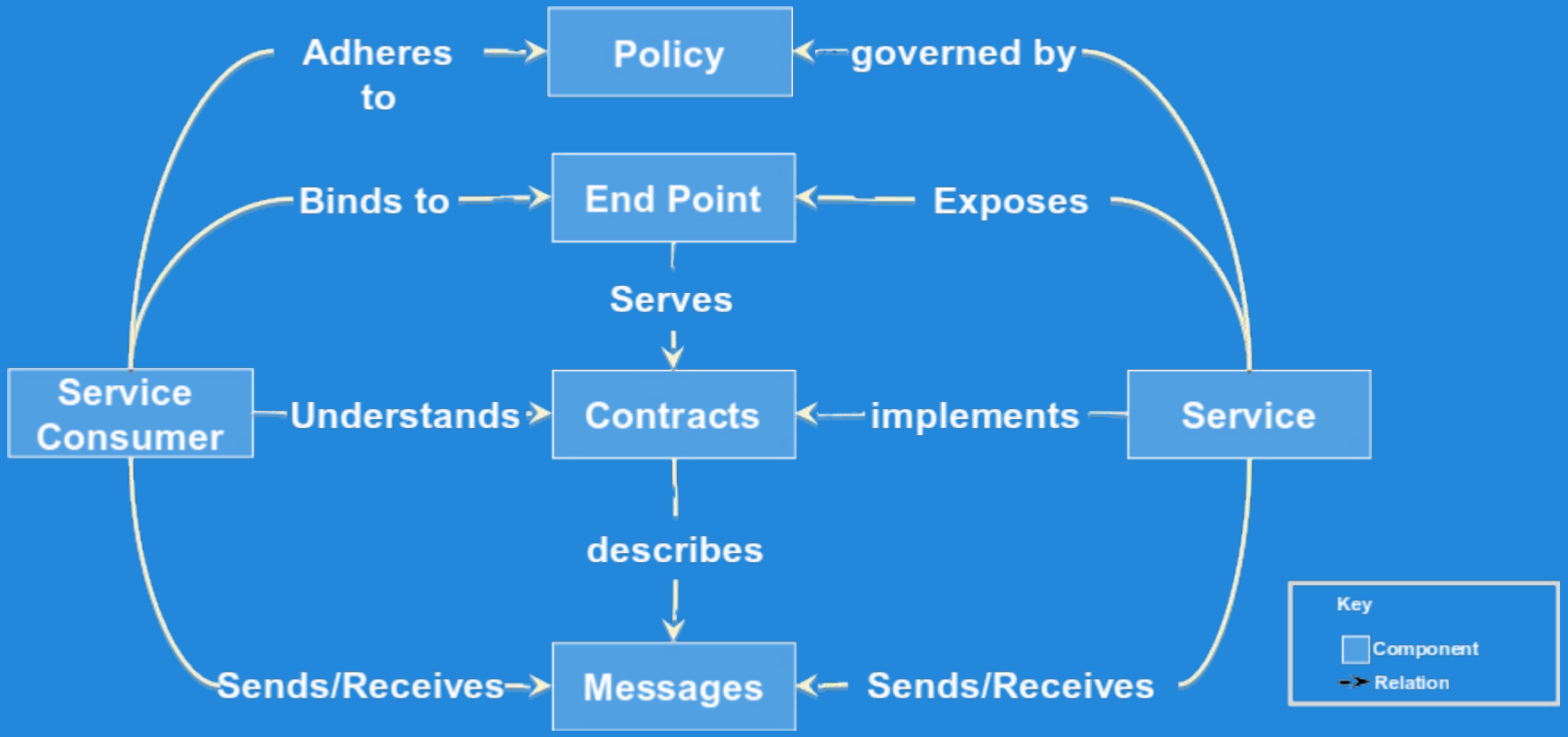
1. **Business value** over technical strategy.
2. **Strategic goals** over project-specific benefits.
3. **Intrinsic interoperability** over custom integration.
4. **Shared services** over specific-purpose implementations.
5. **Flexibility** over optimization.
6. **Evolutionary refinement** over initial perfection.

Service Hierarchy

- Services tend to naturally form a layered architecture.
  - Data abstraction layer retrieves and writes to underlying databases.
  - Data services transform that data and provide messaging queues.
  - Services offer low-level business tasks, may be combined to perform “high-level” tasks by the process/orchestration layer.
  - Top levels perform integration and monitoring of the whole system.
  - Security and governance services work across the layers.



Service Interactions



Slide by Arnon Rotem-Gal-Oz

Standardized Service Contract

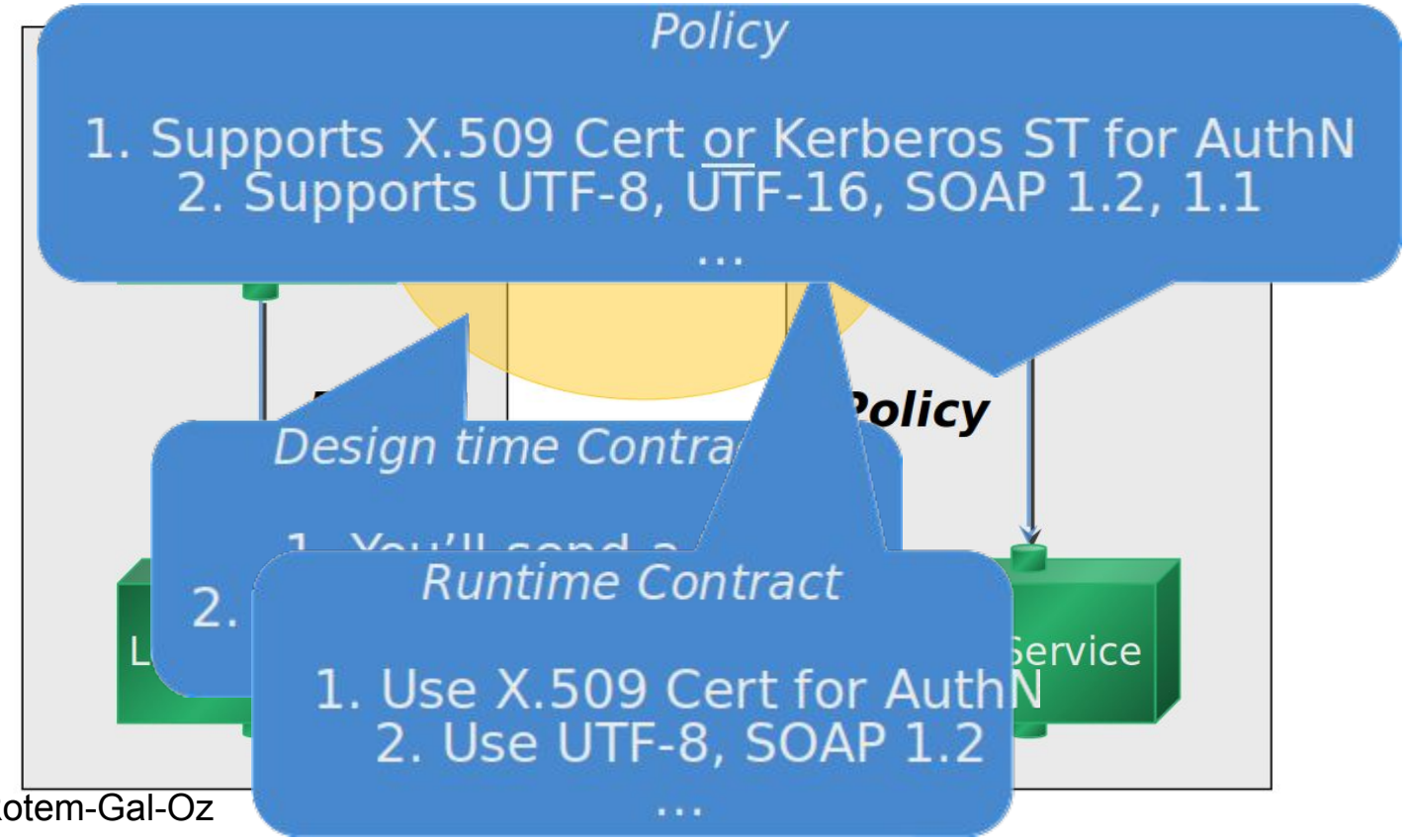
- Services within the same service inventory should be in compliance with the same contract design standards.
  - Services share schema and contract, not class.
  - Service compatibility is based on policy
- A service contract is a promise of the purpose and capabilities of a service.
  - Its public interface.
  - The nature and quantity of content that it will publish.
  - How do services express functionality?
  - How are data types/models defined?
  - How are policies asserted and attached?

Standardized Service Contract

- The service contract should govern all services that you offer in one “inventory”.
  - Consists of a **functional expression standardization** - defining the interface, input, and output (WSDL).
  - A **data model** (XML schema) - defining formats.
  - A **policy document** - defining terms of use.
- Service contracts ensure services are consistent, reliable, and governable.
  - Standards must be applied correctly. Service contracts avoid ambiguity.

Contracts and Policies

Contacts can be established at design and run-time.
Policies are constraints that ensure contracts are met.



Slide by Arnon Rotem-Gal-Oz

Principles of Services and SOA

Service Abstraction

- Public information on a service should be limited to what it required for use.
  - Too much knowledge of the inner workings of a service leads to increased coupling to a particular implementation.
- Functional abstraction: How much of the service logic is exposed to the public?
  - Public vs private functionality - what logic can consumers access?
  - In the service contract, do not discuss inner details of business rules and validation logic.

Service Abstraction

- Technology information abstraction:
  - Do not tell consumers how the service logic and implementation are designed.
- Logic abstraction:
  - Do not provide too much detail on how service performs functionality, as consumers may be designed around that knowledge.
  - Risks hampering logic refactoring.
- Quality abstraction:
  - Only provide details that help in determining reliability and availability, not on other quality attributes.

Service Granularity

- How much does a service do?
  - Business Function:
    - Each service operation maps to a single business function. Can be violated if combination does not add design complexity or increase message size.
  - Performance:
    - The service should use a minimal number of service requests.
  - Message Size:
    - Services should only transmit data required. Try to reduce message sizes.
  - Quality of service characteristics:
    - Each operation should perform a single system transaction and leave cross-border data integrity to the consumer.

Loose Coupling

- Services must be as independent as possible from other services.
- Run-time coupling:
  - Other services may not always be available.
  - Resend messages.
  - Cache results when:
    - The known interval for service updates,
    - Client uptime requirements stricter than service uptime requirements,
    - There are bandwidth problems in distribution.

Loose Coupling

- Interface coupling:
  - Should be able to exchange services with compatible interfaces. Data should be published in standard formats.
  - Interfaces need to evolve over time. Support multiple versions to allow client migration
- Multiple types of interface coupling:
  - Logic-to-contract: Behavior dictated by contract.
  - Contract-to-logic: Contract dictated by existing logic.
  - Contract-to-implementation/technology: Contract dictated by implementation details or technology.
  - Contract-to-consumer: Contract written for a client.

Loose Coupling

- Service reference autonomy:
  - Services should only be aware of the existence of other services.
  - Only all services through their public API.
  - Any services offering the same interface can be swapped.
- Service location transparency:
  - Services can be called from anywhere in the network, no matter where it is present.
  - Online, or on a local network.
  - Services can be located anywhere in the world, as long as they are accessible on the network.

Service Autonomy

- Services should exercise a high level of control over their execution environment.
  - A service should not contain logic dependent on anything external to the service - data models, information systems, shared resources.
  - A service cannot be reusable if its logic is coupled to external artifacts.
- Design-time autonomy: Can the service be evolved without impacting consumers?
  - Enabled by loose coupling and abstraction.
  - Shields contract from logic and implementation, allowing redesign.

Service Autonomy

- Run-time autonomy:
  - Can a service control how their logic is processed by the runtime environment?
  - More control = more reliable behavior.
  - If the service is memory-intensive, deploy to a server with reserved resources.
  - Provide locally cached copies of data to reduce dependency on a shared database.
- Increasing design-time autonomy increases run-time control over the environment.
- May require customized environments.

Service Statelessness

- Scalability requires separating services from their state data whenever possible.
  - Reduces the resources consumed by a service, as state management is delegated to the consumer.
  - Increases the number of requests that can be handled by the service.
- Core tenant of REST (a form of SOA), other SOA styles may relax to varying degrees.
  - May need to retain some business data (i.e., customer records) or session data between tasks.
  - Still, must allow multiple concurrent connections with no side effects.

Service Discoverability

- Services should be supplemented with meta-data that can be used to allow discovery by other services.
  - Supports reuse and composability.
  - Allows developers to identify existing services that fulfill generic requirements of the process being automated.
- Services are registered to a service registry.
  - Java - Maven Repository
  - Bluetooth Service Discovery Protocol

Service Boundaries are Explicit

- A service edge is a natural boundary.
- Services should not cross those boundaries when performing computations or working with data.
- Crossing boundaries is costly:
  - Location of targeted service may be unknown.
  - Security models are likely to differ.
  - Data representations differ publicly and privately.
  - Services evolve and are reconfigured.
  - Consumers are unaware of how internal processes are implemented, and have limited control.

Service Boundaries are Explicit

- Do not use RPCs when crossing borders. Instead, use messages.
  - RPCs trick us into thinking there is no substantial difference between local and remote objects.
- Messages will be lost. Design them to be retransmitted.
  - Idempotence - as long as request is processed at least once, we will see correct behavior.
  - Multiple instances of a request should do the same thing. No side effects.
  - Modern systems must be designed to be idempotent.

Idempotence

Naturally Idempotent
Sweeping the Floor

Naturally Idempotent
Read Record "X"

Not Idempotent
Withdrawing
$1 Billion

Idempotent
If Haven't Yet Done
Withdrawal #XYZ
for $1 Billion,
Then Withdraw
$1 Billion and
Label as #XYZ

Not Idempotent
Baking a Cake
Starting from
Ingredients

Idempotent
Baking a Cake
Starting from
the Shopping
List (If Money
Doesn't Matter)

Slide by Pat Helland

Service Composability

- Services should be designed to be reused as part of systems-of-services.
  - All software should be reusable. We should be able to build a system by webbing together existing parts.
  - Services should be designed to be used either as a service that controls other services, or as a service that provides a function to other services.
  - Service contract must present functionality based on varying levels of input and output.
    - If a composition member, input is more fine-grained than when it is a controller.
  - Services must be as stateless as possible.

Service Composability

- Factors determining composability include:
  - Ability to provide functionality at different levels within a process.
    - Surfacing proper interfaces.
  - Message exchange pattern.
    - One-way (request/reaction) versus duplex (request/reply)?
  - Whether the service supports transactions and rollback/compensation features.
  - Support for exception handling.
  - Availability of meta-data about service capabilities and behavior.
    - (discoverability)

SOAP and REST

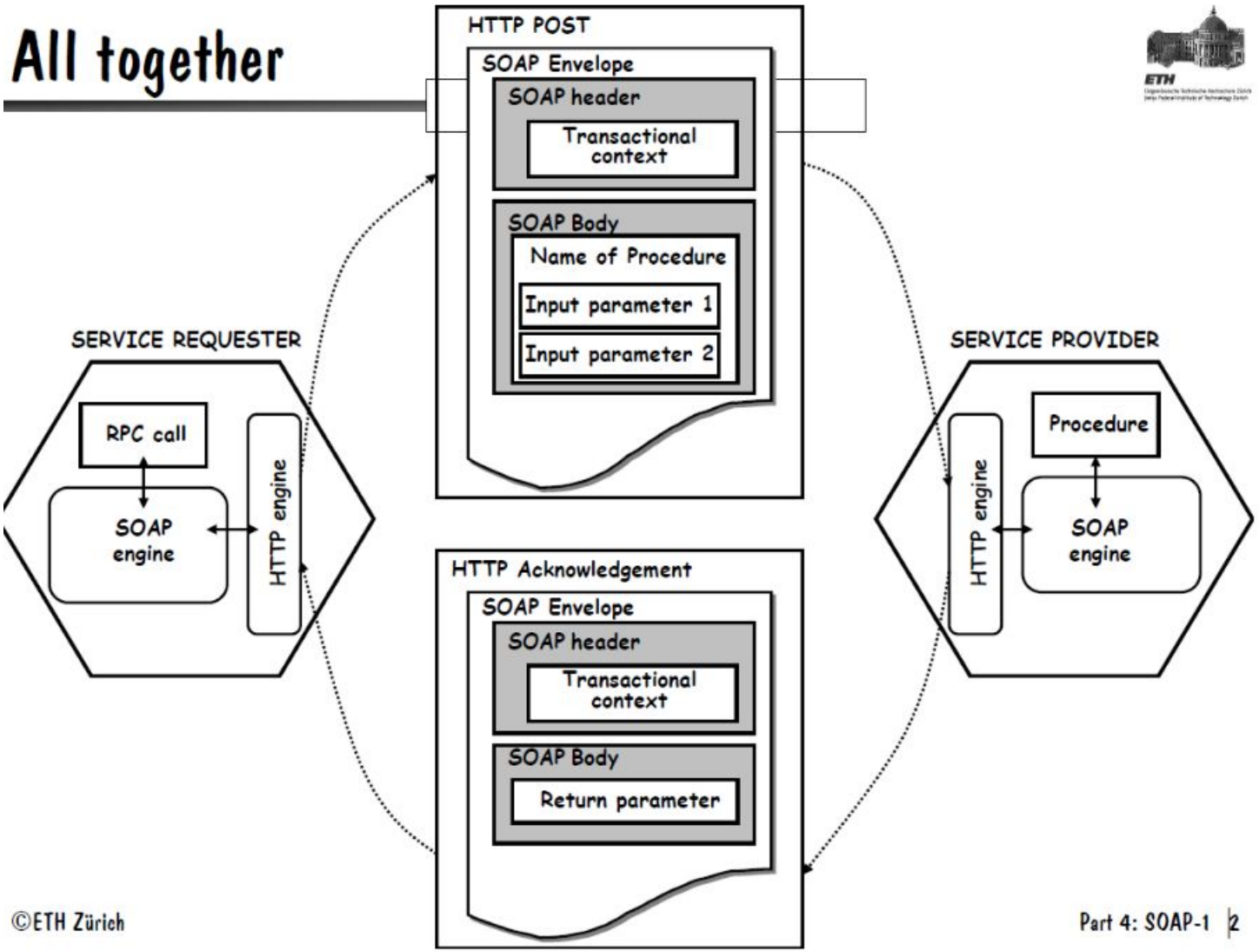
Creating a Web Service

- Online services can be used by external organizations and systems.
- Services communicate through message passing. Therefore, we need standardized means of sending messages across networks.
- Most common: REST and SOAP.
  - We already discussed REST.
  - Now, for SOAP.

Simple Object Access Protocol (SOAP)

- Lightweight protocol used for exchange of messages in a decentralized, distributed environment.
  - Used to perform Remote Procedure Calls.
- By default, uses XML as payload message format and HTTP or SMTP as transport.
- Facilitates interoperability in a platform-independent manner.
  - XML and HTTP are open standard, running on all operating systems.

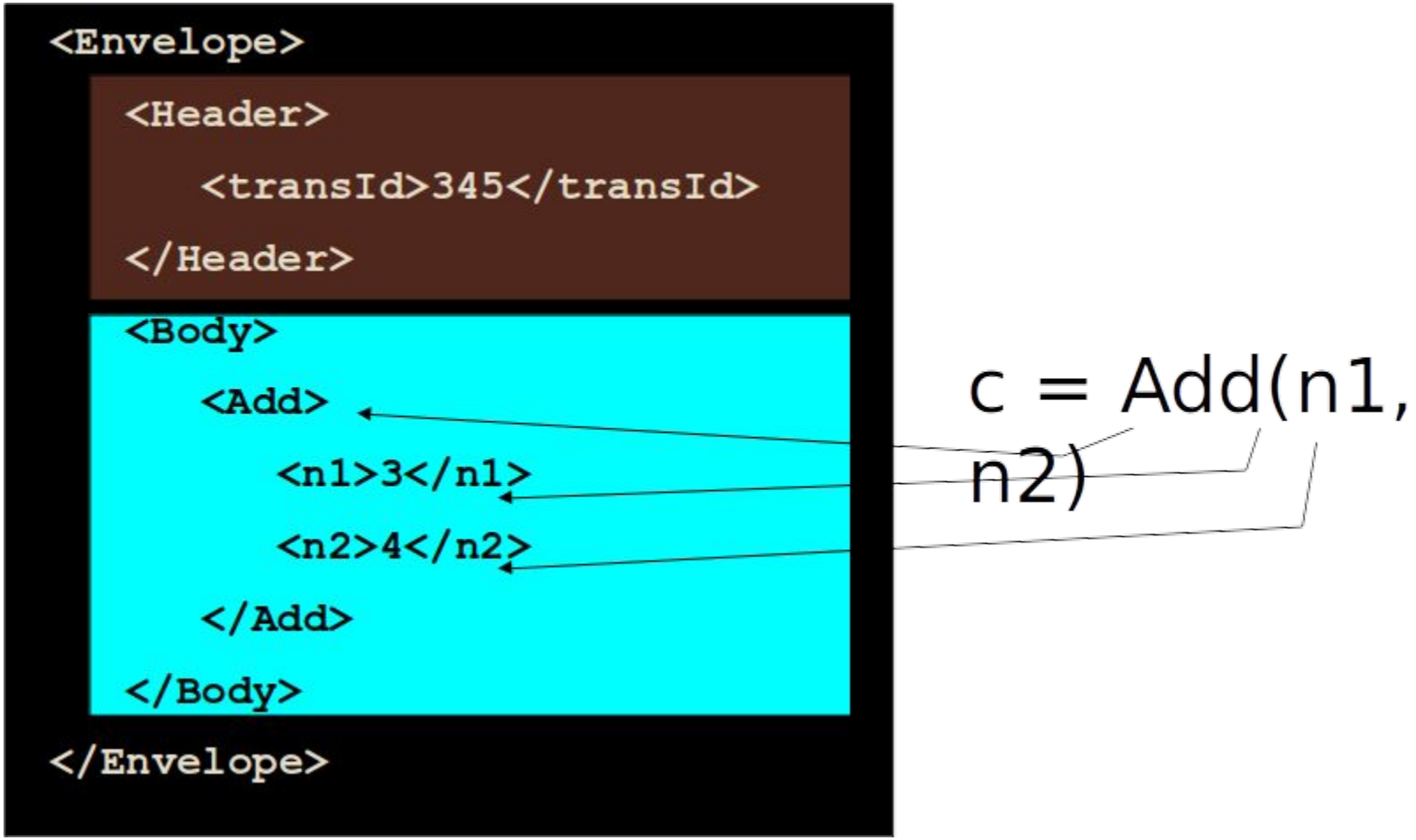
SOAP Ecosystem



SOAP Elements

- Envelope (mandatory)
  - Top element of the XML document.
  - Defines message structure and how to process it.
- Header (optional)
  - Determines how a recipient of a SOAP message should process the message.
  - Adds features to the SOAP message such as authentication, transaction management, payment, message routes, etc.
- Body (mandatory)
  - Includes information for the recipient of the message
  - Typical use is for RPC calls and error reporting.

Simple Example



Slide from: www.cs.columbia.edu/~knarig/SOAP.ppt

SOAP Request

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Header>
    <t:transId xmlns:t="http://a.com/trans">345</t:transId>
  </SOAP-ENV:Header>
<SOAP-ENV:Body>
    <m:Add xmlns:m="http://a.com/Calculator">
      <n1>3</n1>
      <n2>4</n2>
    </m:Add>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Slide from: www.cs.columbia.edu/~knarig/SOAP.ppt

SOAP Request

Scopes the message to the SOAP namespace describing the SOAP envelope

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV=“http://schemas.xmlsoap.org/soap/envelope/”
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/”>

…

</SOAP-ENV:Envelope>
```

Establishes the type of encoding that is used within the message (the different data types supported)

Slide from: www.cs.columbia.edu/~knarig/SOAP.ppt

SOAP Request

Qualifies the transaction ID

```
<SOAP-ENV:Header>
  <t:transId xmlns:t="http://a.com/trans">345</t:transId>
</SOAP-ENV:Header>
```

Defines the procedure to call

```
<SOAP-ENV:Body>
    <m:Add xmlns:m="http://a.com/Calculator">
      <n1>3</n1>
      <n2>4</n2>
    </m:Add>
  </SOAP-ENV:Body>
```

Slide from: www.cs.columbia.edu/~knarig/SOAP.ppt

SOAP Response

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Header>
    <t:transId xmlns:t="http://a.com/trans">345</t:transId>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <m:AddResponse xmlns:m="http://a.com/Calculator">
      <result>7</result>
    </m:AddResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

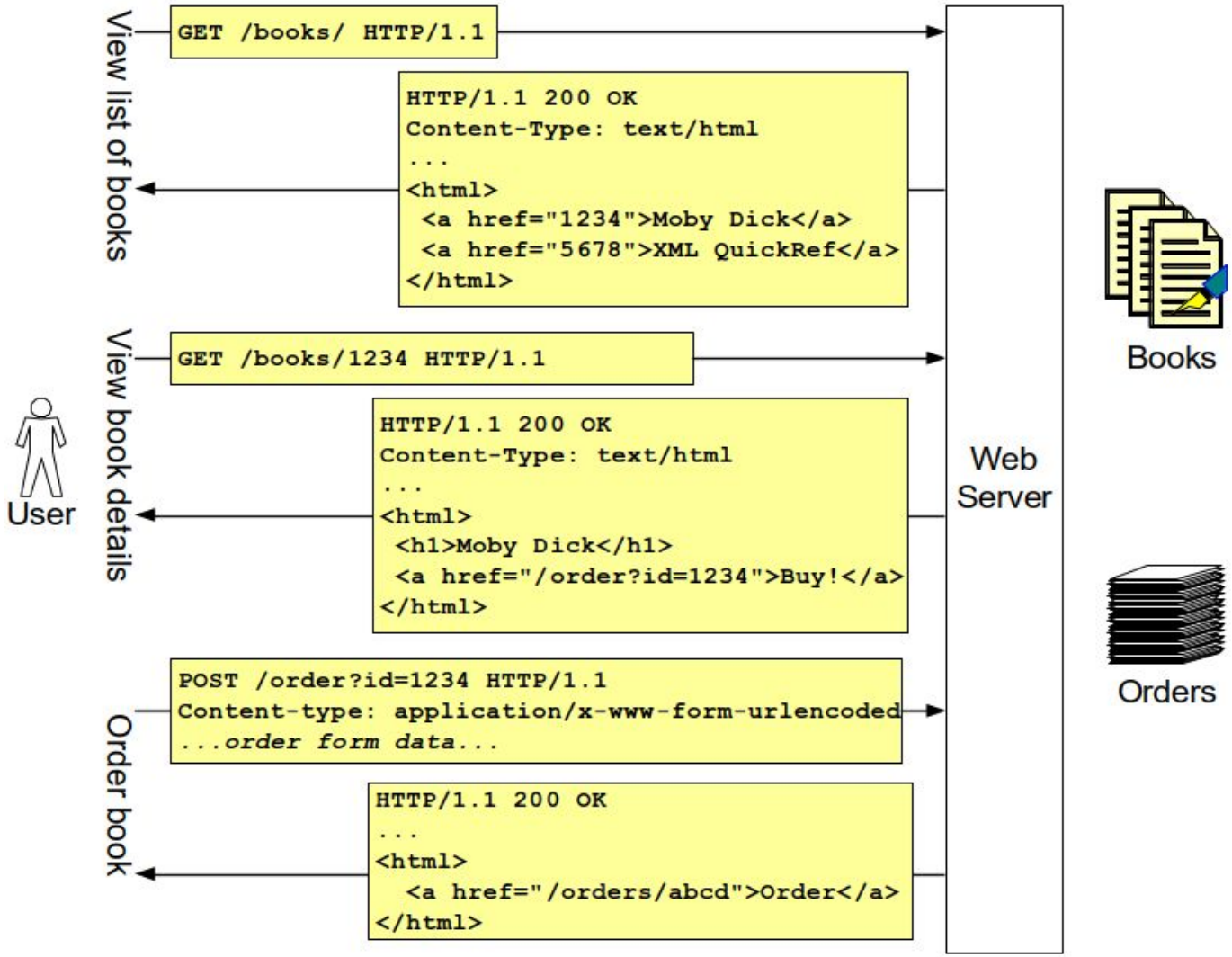
Slide from: www.cs.columbia.edu/~knarig/SOAP.ppt

SOAP Encoding

- Based on a simple type system that has common features with programming languages and databases.
- Types are either simple (scalar) or a composite of several parts.
- An XML schema which is consistent with this type system can be constructed.
  - Use of schemas is encouraged but NOT required.

Slide from: www.cs.columbia.edu/~knarig/SOAP.ppt

REST Bookstore



Slide from DevelopMentor: REST and SOAP

SOAP Bookstore

Slide from DevelopMentor: REST and SOAP

SOAP from a REST Viewpoint: Addressing

- REST architectures utilize the existing web addressing model:
  - Standard URI schemes subsume protocols (http, ftp)
  - Standardized distributed naming authorities (DNS).
  - Standardized way of discovering, referring to resources (URIs).
- SOAP applications define their own addressing schemes
  - Web service entrypoints have URIs.
  - Resources have custom, service-specific addresses.
  - No standardized way of discovering, referring to resources.

Slide from DevelopMentor: REST and SOAP

SOAP from a REST Viewpoint: Addressing Issues in SOAP

- Intermediaries (proxies, caching) cannot operate solely on URI.
- Simple URI-based technologies (XSLT, XInclude) hampered.
- Integrating disparate applications requires custom logic.
- "Deep linking" into applications not generally possible.

Slide from DevelopMentor: REST and SOAP

SOAP from a REST Viewpoint: Generic Interfaces

- REST emphasizes standardized, generic operations:
  - HTTP provides PUT, GET, POST, DELETE.
  - Allows for uniform manipulation of URI-identified resources.
- SOAP does not provide for generic operations:
  - Each application defines its own set of operations
  - Creates need for description, discovery mechanisms
  - Knowledge of semantics of operation is out-of-band.

Slide from DevelopMentor: REST and SOAP

SOAP from a REST Viewpoint: Generic Interface Issues

- Clients need knowledge of description, discovery mechanisms.
- Clients need foreknowledge of specific service semantics.
- Generic clients not universally feasible (local standardization).

Slide from DevelopMentor: REST and SOAP

SOAP from a REST Viewpoint: State Management

- REST apps have explicit state transitions:
  - Servers & intermediaries are inherently stateless.
  - Resources contain data, links to valid state transitions.
  - Clients maintain state, traverse links in generic manner.
- SOAP apps have implicit state transitions:
  - Servers & intermediaries may (should!) be stateless.
  - Messages contain only data (not valid state transitions).
  - Clients maintain state, require knowledge of state machine.

Slide from DevelopMentor: REST and SOAP

SOAP from a REST Viewpoint: State Management Issues

- Clients need foreknowledge of service's state machine.
- Generic clients not universally feasible (local standardization).
- Limits independent evolution of client/server state machine.
- State machine description needed for automated discovery.

Slide from DevelopMentor: REST and SOAP

REST from a SOAP Viewpoint

- SOAP & related technologies have broad industry support.
- Client & server toolkits are widely deployed.
  - Tool support on client & server matters.
- SOAP headers provide a widely adopted extensibility model
  - Despite presence of HTTP extension mechanisms.
- SOAP can be bound to non-HTTP transports
  - Important for richer XML messaging in the future.
- SOAP 1.2 can be used in a RESTful manner
  - "Can't we all just get along?"

Key Points

- Services are small programs that do “a single job” and encapsulate their own data.
  - Services can be reused endlessly.
  - Changes to services should not affect the rest of the system.
- Service-oriented architectures create systems from a collection of services.
  - Services “talk” by exchanging messages.
  - Often performed using REST or SOAP.
    - SOAP offers richer implementations, but lacks standardization of REST.

Next Time

- Machine Learning for Software Architects
  - Guest speaker - Dr. Jamshidi
  - (This will be on the final, so don’t skip!)

- Practice Final
  - On site, without answers.
  - We will go over on December 6
- Homework:
  - Project, Part 4 - Due on December 6
  - Assignment 3 - Due on December 9