Patterns of Service-Oriented Architectures

Uwe Zdun

Email: uwe.zdun@wu-wien.ac.at, zdun@acm.org

WWW: http://wi.wu-wien.ac.at/~uzdun

Department of Information Systems

Vienna University of Economics and Business Administration

Agenda

- Service-oriented architectures (SOA)
- Patterns and pattern languages
- Basic service architecture
- SOA layers and basic remoting architecture
- SOA variation points and adaptation
- SOA and business processes
- Integration of services and processes
- Composing SOAs

Service-oriented architectures (SOA)

Software architecture

There are many definitions of software architecture

For instance, following *Bass, Clements, and Kazman. Software Architecture in Practice, Addison-Wesley, 1997*:

- Every software has an architecture
- Architecture defines components and their interactions
- Interfaces (externally visible behavior) of each component are part of the architecture
- Interfaces allow components to interact with each other
- A system comprises many different kinds of components, but none of these is *the* architecture

Complexity and change

Software architecture is a metaphor that helps us to better cope with the challenges we see in today's software systems

These challenges are described by a number of so-called "Laws of Software Evolution" (Lehman and Belady, 1980). The two most prominent are:

- Law of continuing change
- Law of increasing complexity (entropy)

But: Software architectures are not easy to document, create, and maintain

$\Rightarrow$ Description of the architecture using quality attributes

$\Rightarrow$ Software architectural principles

Architectural quality attributes (1)

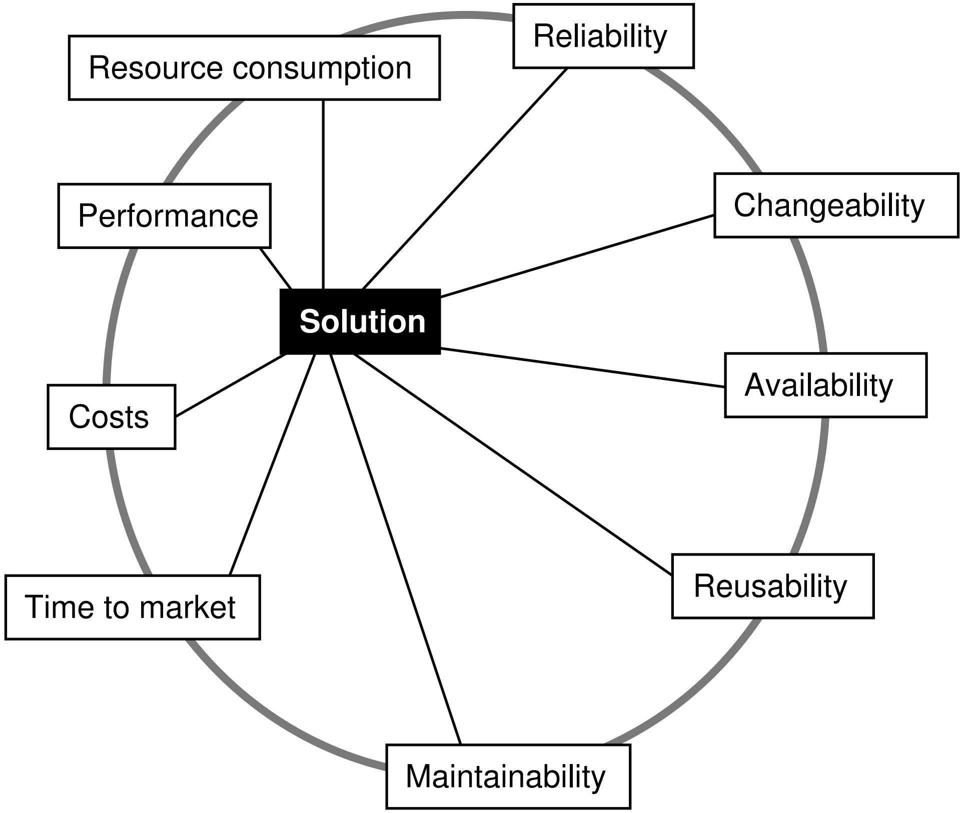
- Quality of an architecture $\sim$ essential attributes for the fulfillment of the requirements
- Factors that are important to make an architecture good or bad

$\longrightarrow$ ISO 9126 International Standard for the Evaluation of Software

$\longrightarrow$ System by Bass, Clement, and Kazman

  - *System quality attributes:* availability, reliability, maintainability, understandability, changeability, evolvability, testability, portability, efficiency, scalability, security, integrability, reusability, ...
  - *Business quality attributes:* time to market, costs, projected lifetime, targeted market, legacy system integration, roll-out schedule, ...
  - *Architecture quality attributes:* conceptual integrity, correctness, completeness, buildability, ...

Architectural quality attributes (2)

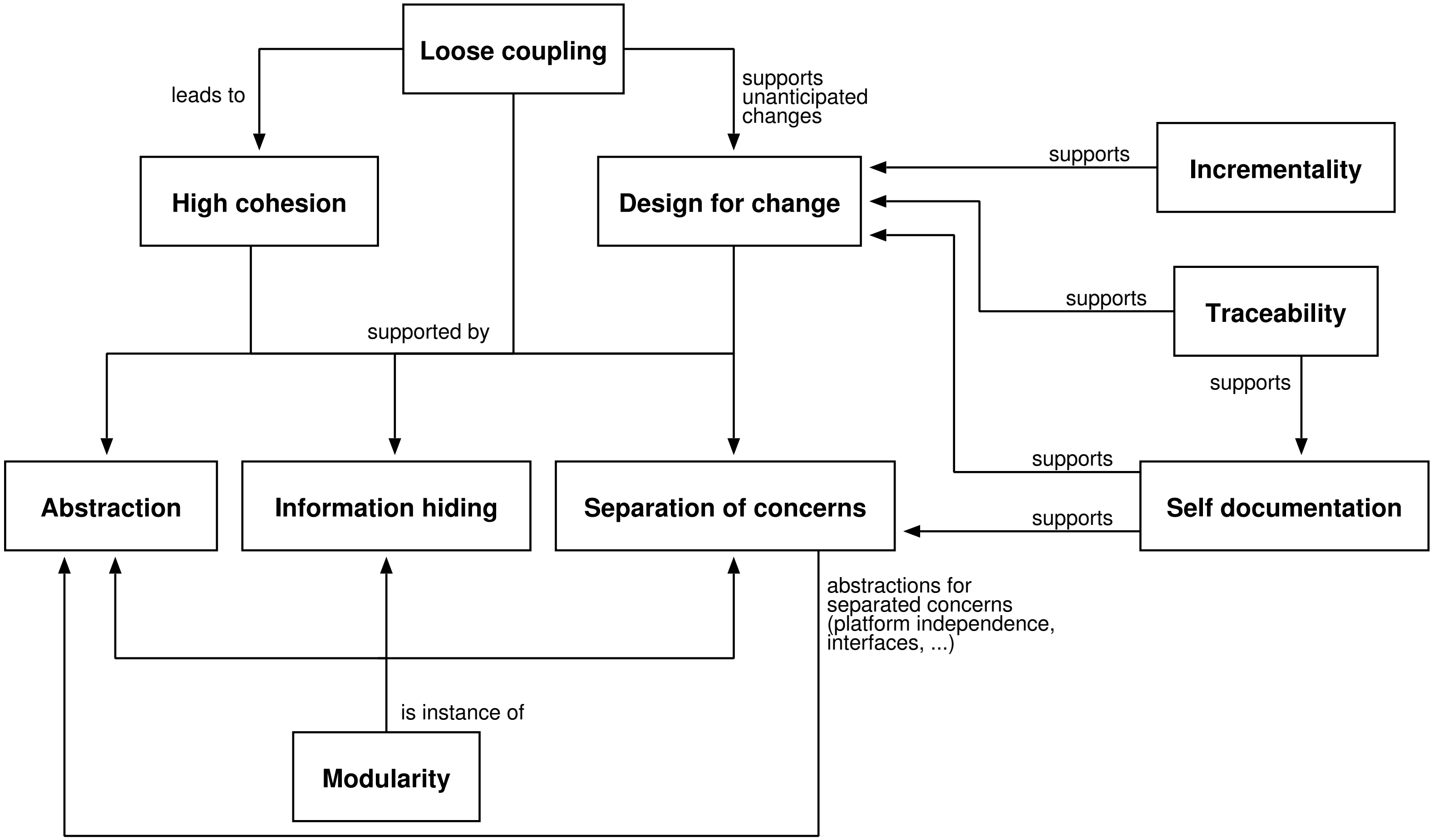


- Architects must find a proper balance between the quality attributes
- Many architectural choices influence the qualities of the architecture
- The quality attributes are generally hard to quantify objectively

Software architectural principles (1)

- More concrete guidelines than quality attributes are needed to make informed decisions about the architecture
- System of software architecture principles:
  - Principles have mainly be presented in the context of other fields than architecture (OO, software engineering, . . .)
  - Here: we explicitly focus on the architectural meaning of those principles
  - Result: system of principles with rich interdependencies
  - Loose coupling is central to building software architectures that can cope both with increasing complexity and continuing change

Software architectural principles (2)



Principles are not sufficient for building architectures because they provide only general guidelines. For creating and maintaining software architectures more concrete guidelines are necessary $\longrightarrow$ Software patterns, reference architectures, ...

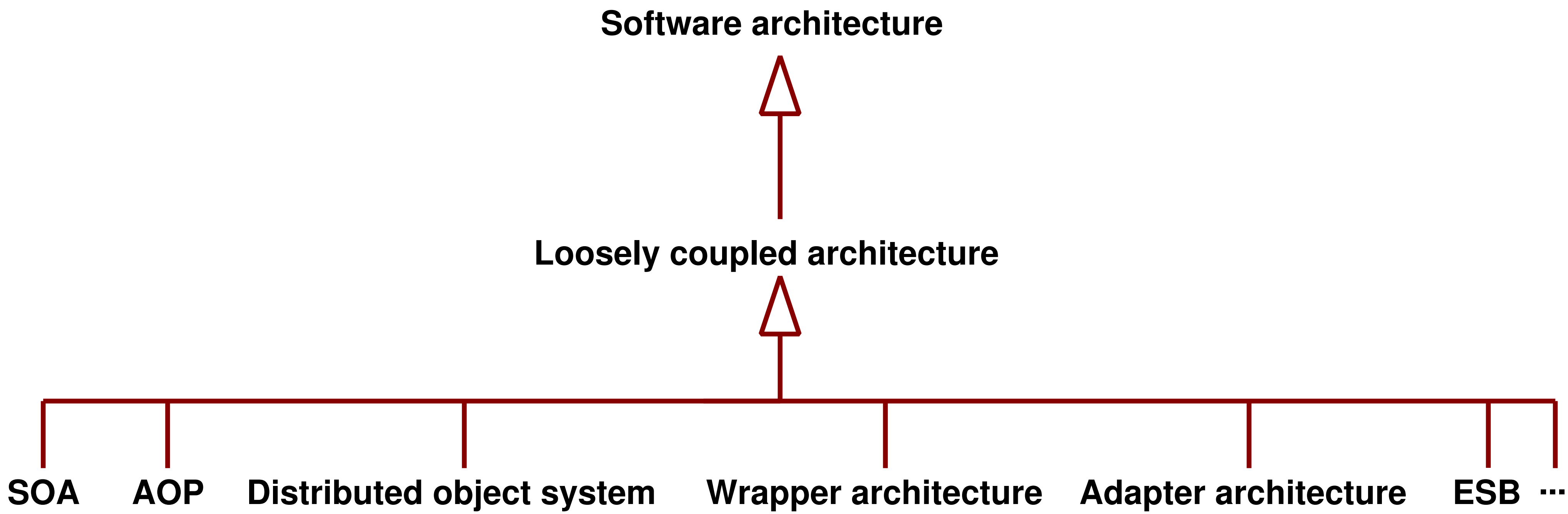
Loosely coupled architectures and SOA

Software architecture

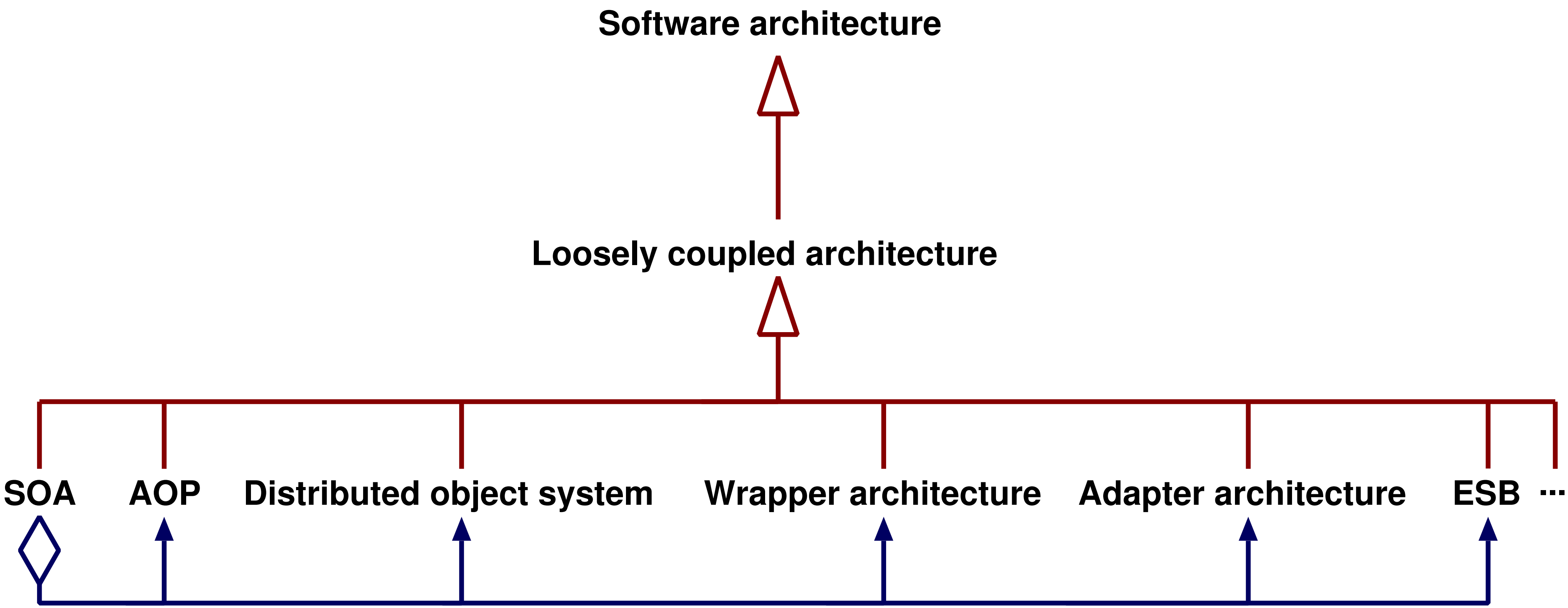
↑

Loosely coupled architecture

Loosely coupled architectures and SOA



Loosely coupled architectures and SOA



Service-oriented architectures (SOA)

- A service-oriented architecture (SOA) is essentially a collection of services that are able to communicate with each other
- Each service is the endpoint of a connection, which can be used to access the service and interconnect different services
- Communication among services can involve only simple invocations and data passing, or complex coordinated activities of two or more services
- In this sense, service-oriented architectures are nothing new

SOA: Properties

- Services offer public, invokable interfaces
- These interfaces are defined using interface description languages
- Each interaction is independent of each other interaction
- Many protocols are used and co-exist
- Platform-independent
- SOA is an architectural concept, not a specific technology:
  - Nothing new (many CORBA systems realize SOAs, for instance)
  - Not equal to Web services (just one technology to realize SOAs)

SOA: Goals

- SOAs are not well-defined at the moment and there is not much architectural guidance how to design a SOA
  - Many definitions and guides are focused on concrete technologies
  - Not on the essential elements of the architecture
- Providing an architectural framework for Service-Oriented Architectures (SOA)
- Survey of patterns relevant for building SOAs
- Towards a reference architecture based on software patterns

Patterns and pattern languages

Pattern definition

- Software patterns provide reusable solutions to recurring design problems in a specific context
- Pattern definition by Alexander: *A pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution.*
- Heavily simplified definition
- Alexander's definition is much longer. Summary by Jim Coplien:

  Each pattern is a three-part rule, which expresses a relation between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain software configuration which allows these forces to resolve themselves.

Elements of a pattern

- Name
- Context
- Problem
- Solution
- Forces
- Consequences
- Examples/Known Uses
- Pattern Relationships

Kinds of patterns

- Last couple of years: Patterns have become part of the mainstream of OO software development
- Different kinds of patterns:
  - Design patterns (GoF)
  - Software architecture patterns (POSA, POSA2, SEI)
  - Analysis patterns (Fowler, Hay)
  - Organizational patterns (Coplien)
  - Pedagogical patterns (PPP)
  - Many others
- Many of the patterns in this tutorial are architectural patterns

Software architecture patterns

- Problem: Quality attributes and principles are very generic and hard to use for concrete solving problems
- Goal: Concrete, but yet generic guidelines for documenting, creating, and maintaining SW architectures
- Solution: Software architecture patterns

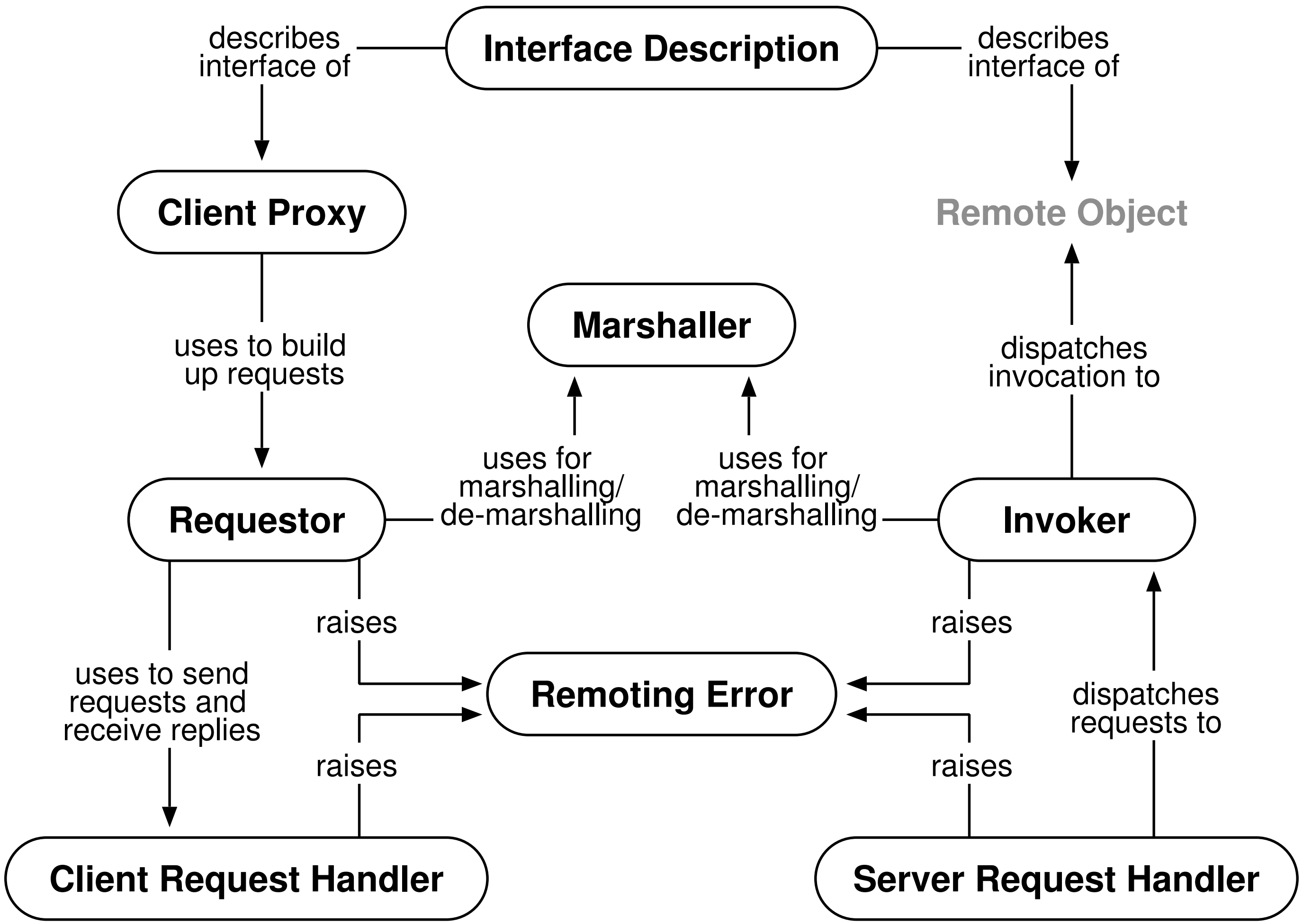
  $=$ Patterns working in the architectural realm

  $\longrightarrow$ Most patterns presented in this tutorial are architectural patterns

From patterns to pattern languages

- Single pattern $=$ one solution to a particular, recurring problem
- However: “Real problems” are more complex $\longrightarrow$ Pattern relationships:
  - Compound patterns
  - Family of patterns
  - Collection or system of patterns
  - Pattern languages
- Pattern languages:
  - Language-wide goal
  - Generative
  - Sequences $\longrightarrow$ has to be applied in a specific order
  - Pattern defines its place in the language $\longrightarrow$ context, resulting context

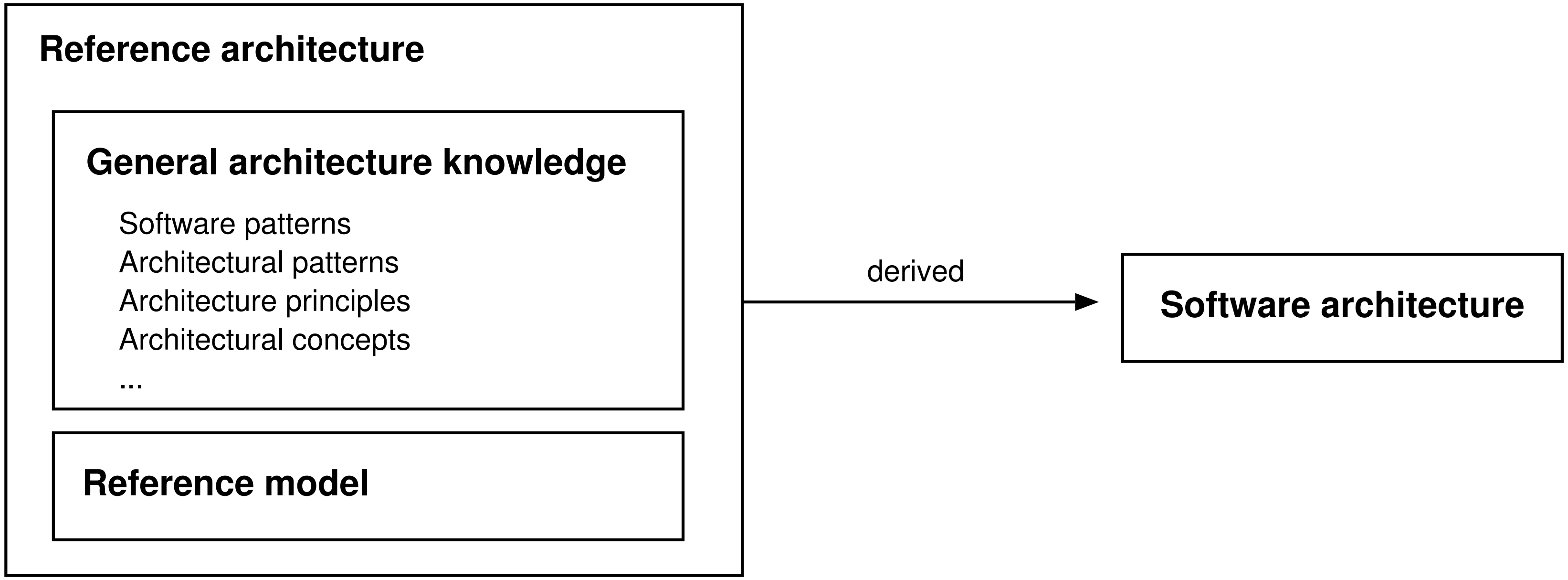
Example: Pattern language overview diagram – Basic remoting patterns



Patterns and SOA

- Pattern-based approach enables a broad, platform-independent view on SOAs that still contains all relevant details about the technical realization alternatives
- Many architectural patterns are important for SOAs:
  – General software architecture (POSA, SEI, . . .)
  – Pattern languages for remoting, messaging, resource management
  – Networked and concurrent objects
  – Object-oriented design
  – Component and language integration
  – e-business, process-driven architectures, business objects, and workflow systems
- Domain-specific combination of these patterns – in the SOA domain

Pattern-based reference architecture for SOAs

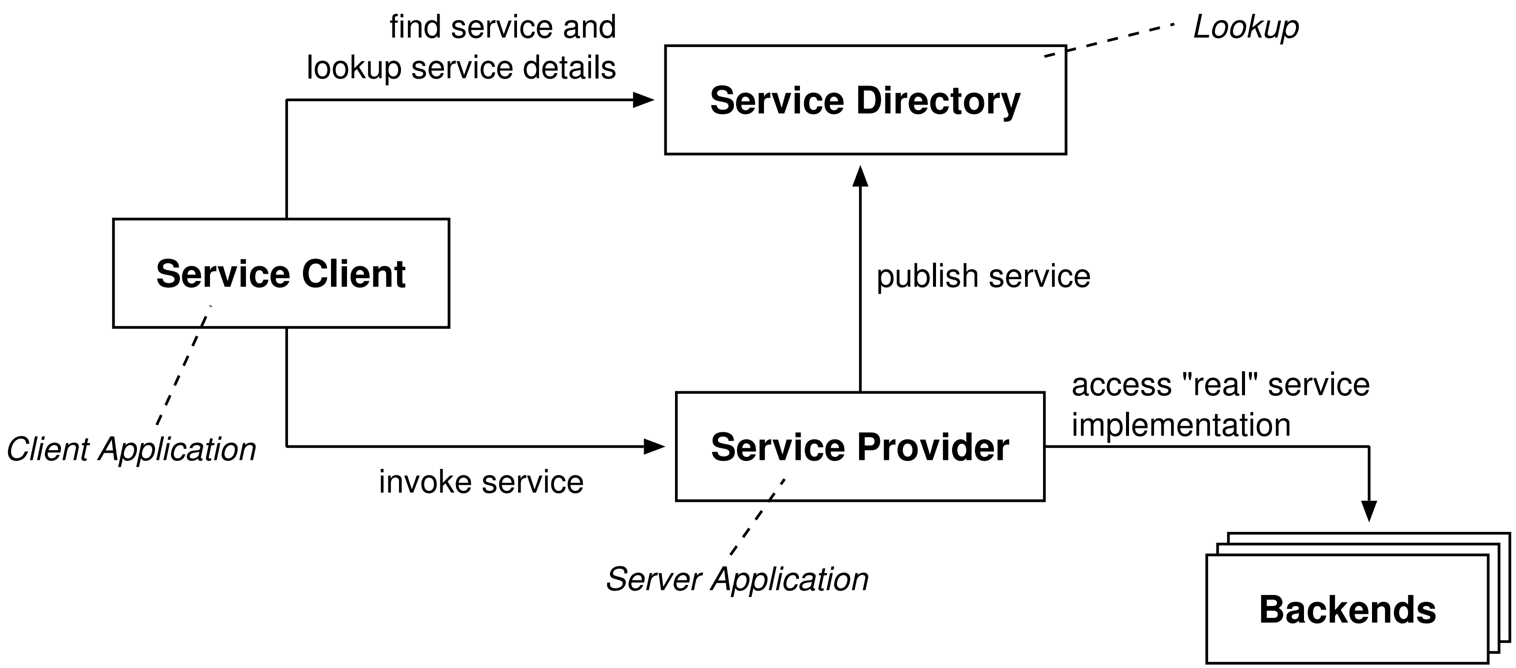


- Principles are used in a specific way
- Specific quality attributes are in focus
- SOAs are described in a technology-neutral fashion
- Nonetheless: Concrete guidelines are given

Basic service architecture

Basic SOA concept

- A service offers a remote interface with a well-defined INTERFACE DESCRIPTION
- The interface description contains all interface details about the service
- The service advertises itself at a central service, the LOOKUP service



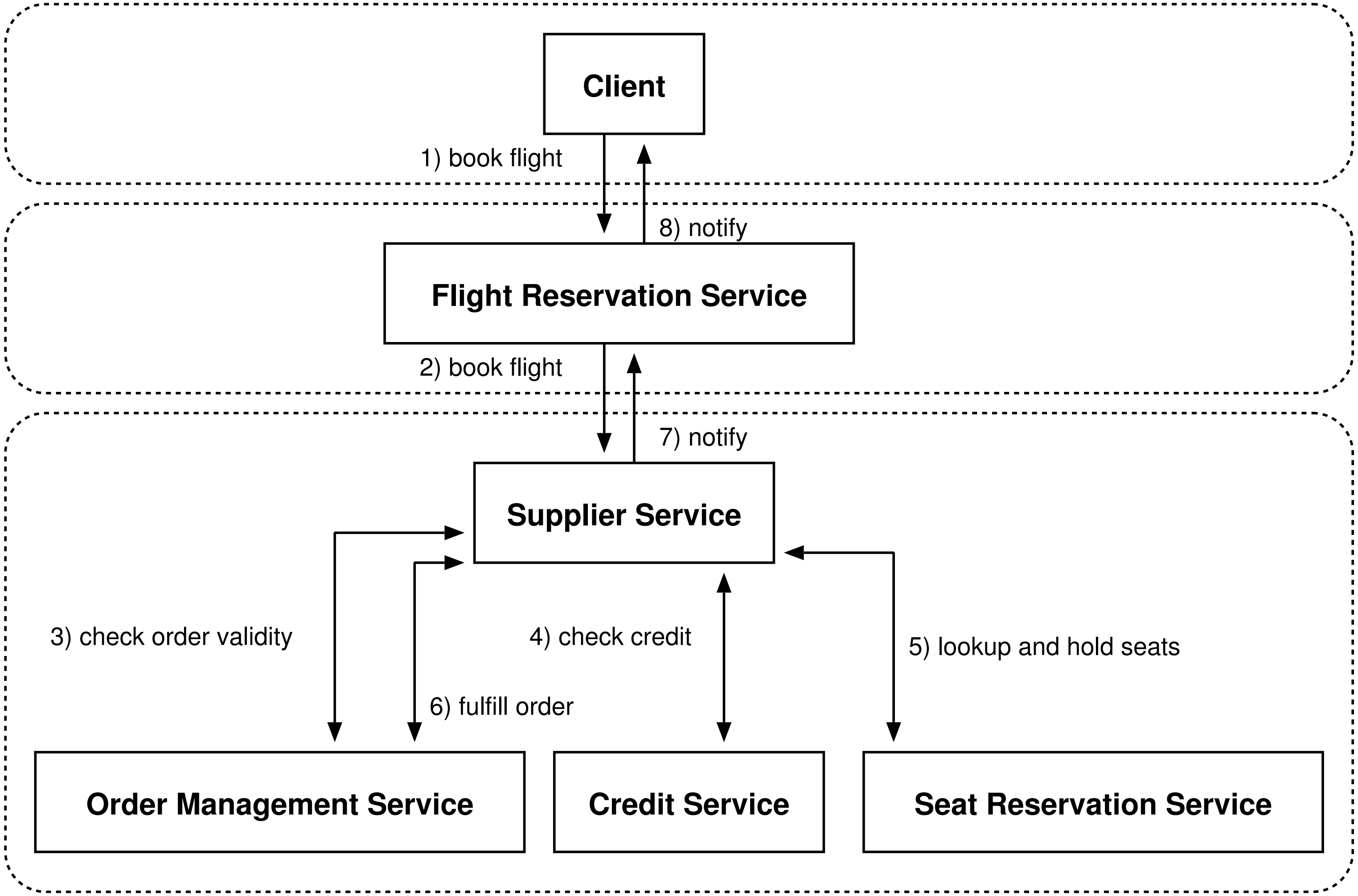
Lookup in SOAs

- Developers usually assign logical OBJECT IDS to identify services
- But they are only valid in the local server application context
- An ABSOLUTE OBJECT REFERENCE extends OBJECT IDS to include location information, such as host name and port
- The LOOKUP pattern plays a central role in a SOA:
  - Services are published in a service directory
  - Clients can lookup services
- LOOKUP is queried for properties (e.g. provided as key/value pairs) of the service and other details
- Usually clients can lookup the ABSOLUTE OBJECT REFERENCES of a service

Service composition

A service client is often itself a service provider, leading to the composition of services.

Example:



Service contracts

- Central idea: Services reflect a contract between the service provider and service clients
- Concept derives from the design-by-contract concept:
  - Originally developed for software modules
  - An approach to design in which each method has a contract with its callers regarding preconditions, postconditions and invariants
- Service contracts define the interaction between service client and service provider
- Intention: a service needs to be specified a step further than simple remote interactions, such as RPC-based invocations in a middleware

Elements of service contracts

- Communication protocols
- Message types, operations, operation parameters, and exceptions
- Message formats, encodings, and payload protocols
- Pre- and post-conditions, sequencing requirements, side-effects, etc.
- Operational behavior, legal obligations, service-level agreements, etc.
- Directory service
- ...

Note: Not all of these contract elements can be expressed with today's Web services implementations easily

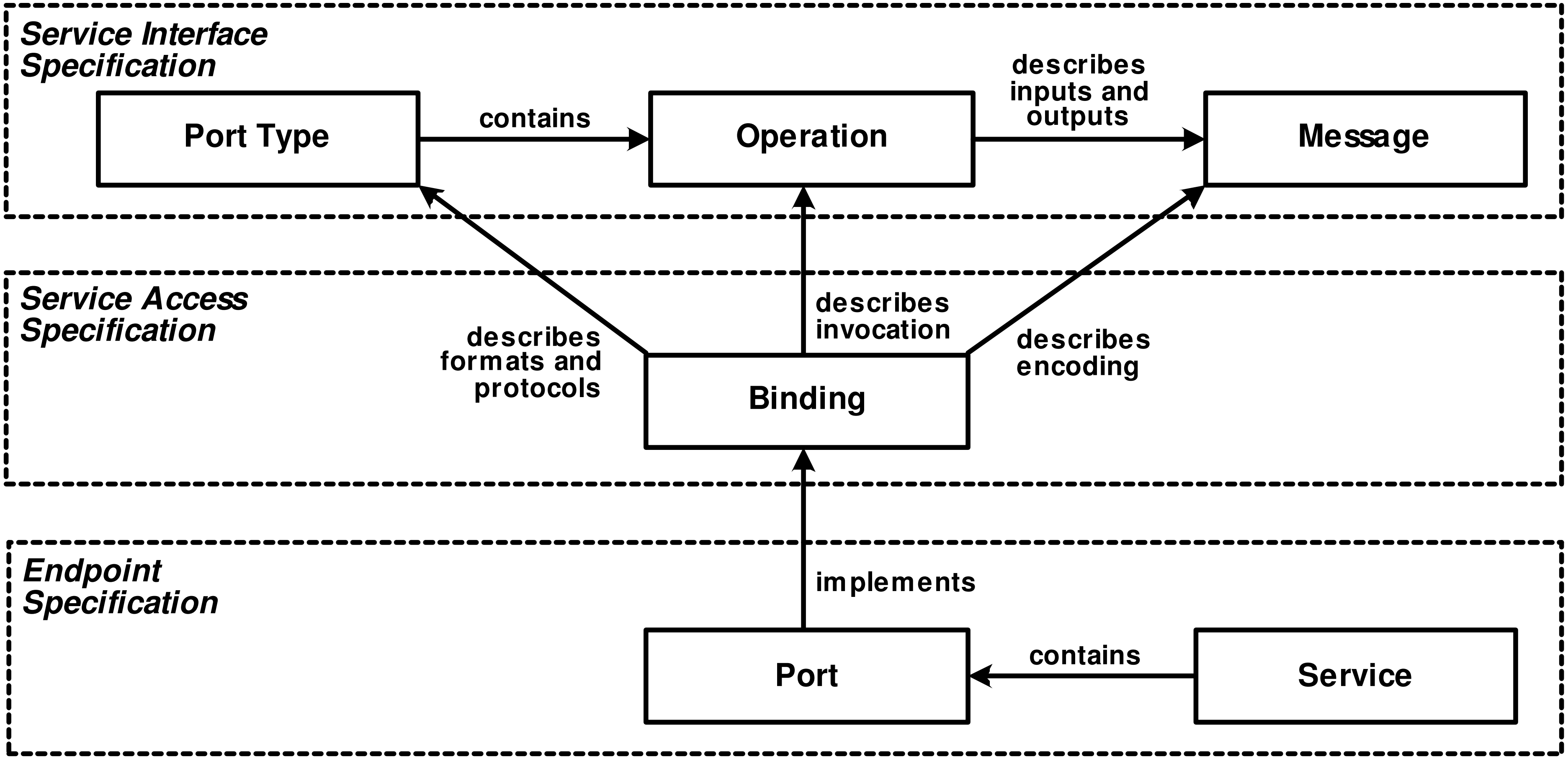
Expressing the elements of the service contract

- A service contract is realized by a mixture of explicit and implicit specifications
- Implicit, non-electronic specifications are inconvenient for the technical specification elements though (e.g. ABSOLUTE OBJECT REFERENCES distributed by hand)
- Some elements are often specified only implicitly or non-electronically:
  - Documentation of the services behaviour and its implied semantics
  - Business agreements
  - Quality of service (QoS) guarantees
  - Legal obligations
  - ...
- Might be needed in electronic form, e.g. to verify or monitor the quality of the service (e.g. using the pattern QOS OBSERVER)

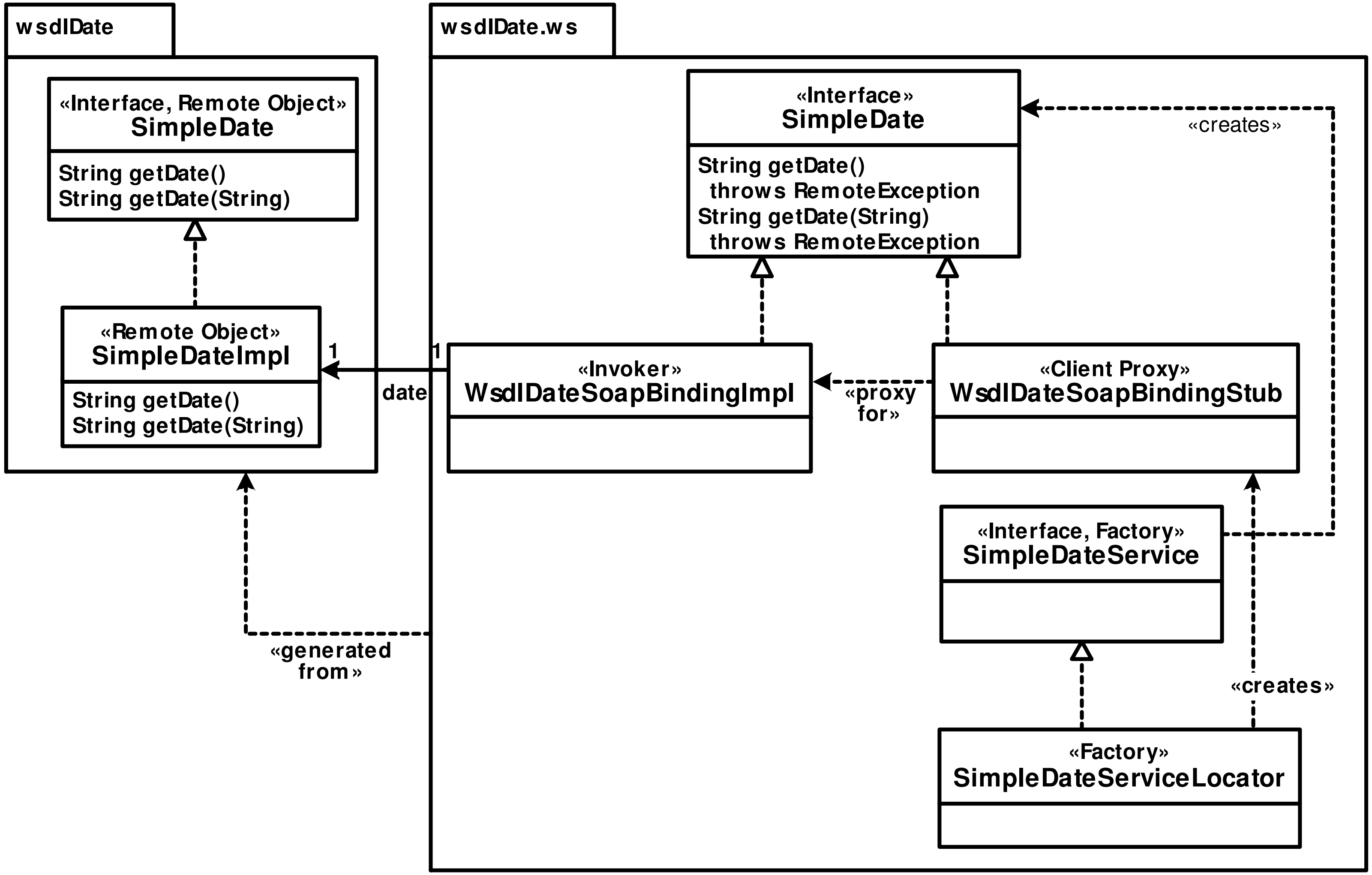
The role of Interface Descriptions in service contracts

- Communication channels and messages are usually described with INTERFACE DESCRIPTIONS
- INTERFACE DESCRIPTION pattern:
  - Describes the interface of remote objects (services)
  - Serves as the contract between CLIENT PROXY and INVOKER
  - Used to enable code generation or runtime configuration techniques
- The INTERFACE DESCRIPTION of a SOA needs to be more sophisticated than the INTERFACE DESCRIPTIONS of (OO-)RPC distributed object middleware, however
- Needs to be able to describe a wide variety of message types, formats, encodings, payload, communication protocols, etc.

Example: WSDL

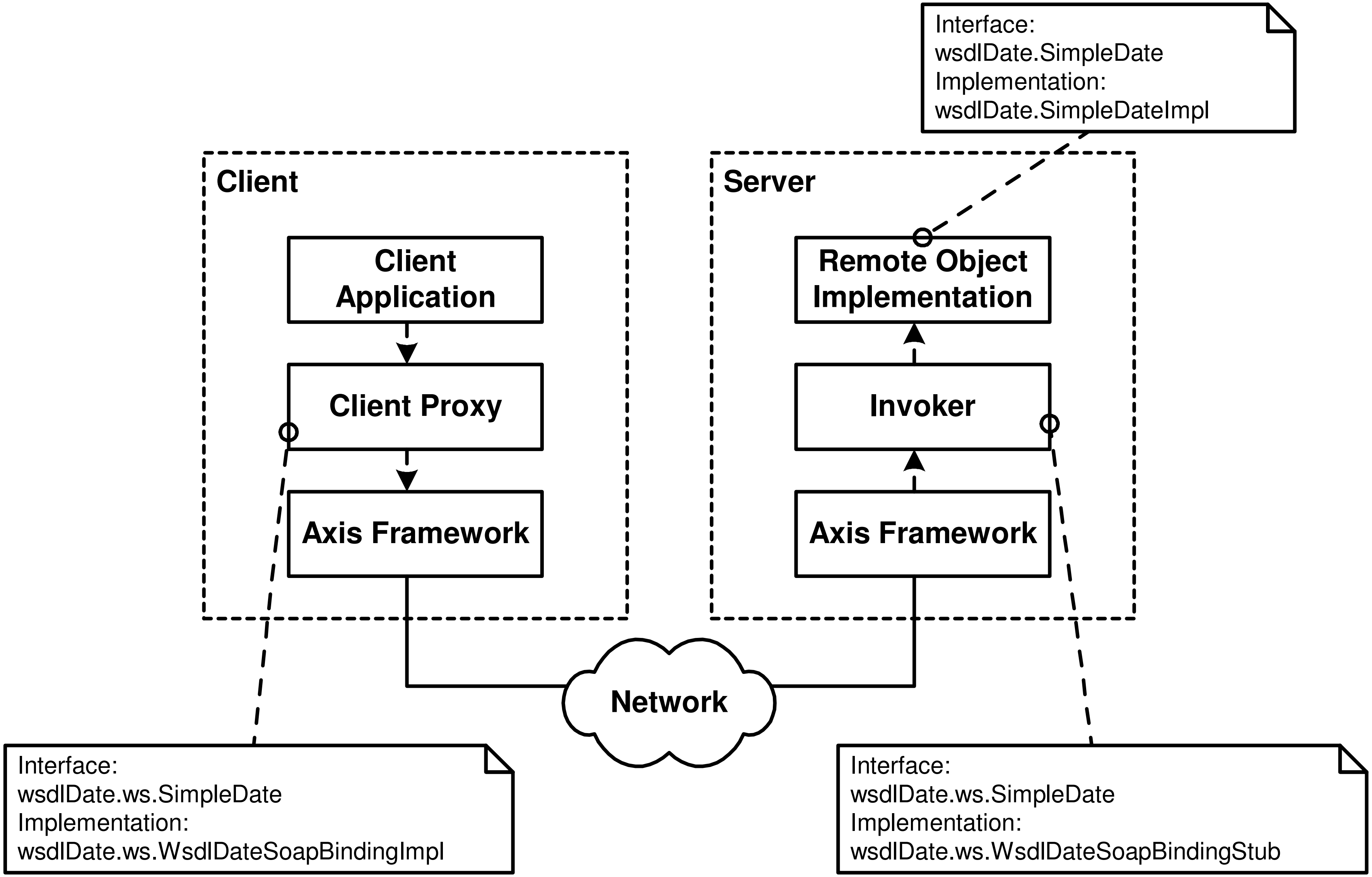


Example: WSDL generation in Axis (1)



From the WSDL INTERFACE DESCRIPTION we generate the CLIENT PROXY and INVOKER code

Example: WSDL generation in Axis (2)



Lookup and service contracts

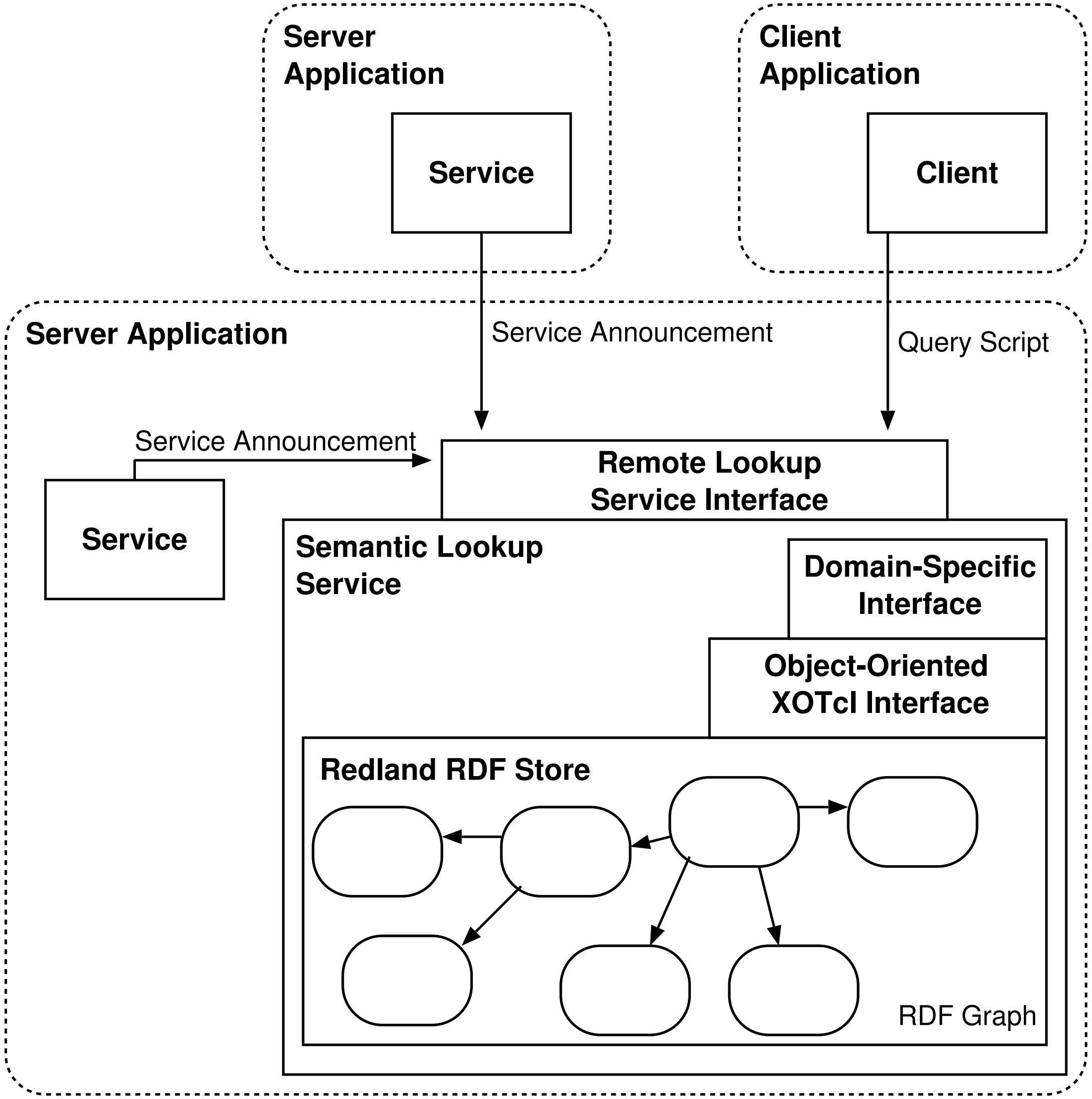
A SOA lookup service might offer not only the ABSOLUTE OBJECT REFERENCES, but also elements of the service contract:

- INTERFACE DESCRIPTION of the service
- a location where the INTERFACE DESCRIPTION can be downloaded
- other metadata about the service (e.g. described using domain-specific schemas or ontologies, as for instance industry-specific XML schemas like OFX or MISMO)
- elements of the service contract, such as operational behaviour, legal obligations, and service-level agreements

Example: Lookup of Web Services with UDDI

- Many possible ways to realize lookup with Web Services
- UDDI is an automated directory service that allows one to register services and lookup services
- All UDDI specifications use XML to define data structures
- An UDDI registry includes four types of documents:
  - A business entity is a UDDI record for a service provider
  - A business service represents one or more deployed Web Services
  - A technical model (tModel) can be associated with business entities or services
  - A binding template binds the access point of a Web Service and its tModel
- UDDI allows a service provider to register information about itself and the services it provides

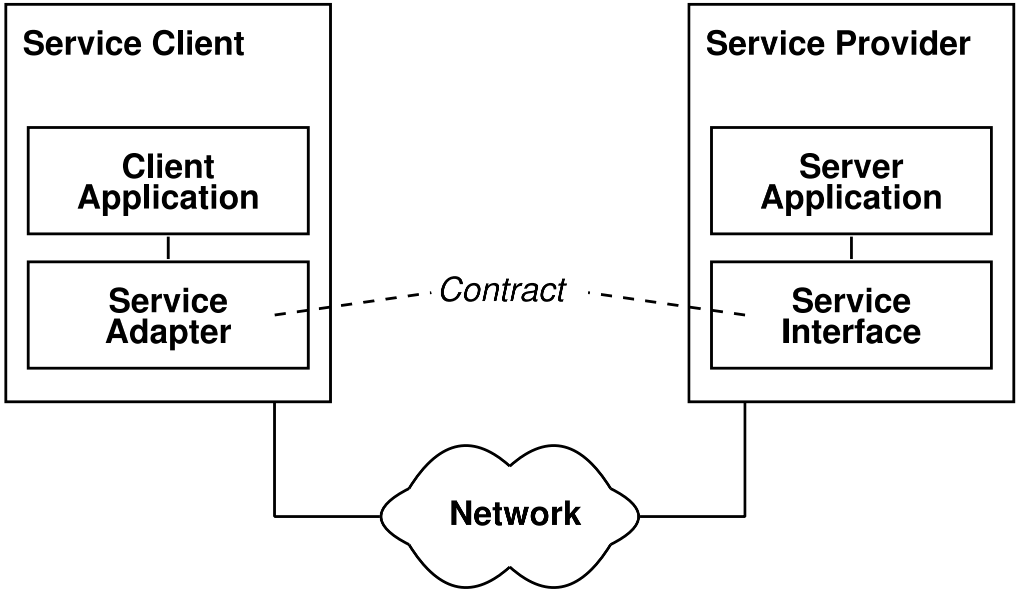
Example: Semantic lookup service



Service interface and service adapter (1)

- If SOA is used within larger client and server applications for integration purposes, then it is advisable to introduce:
  - a service interface to the server application and
  - a service adapter on the client side
- Both are separated from the rest of the application and encapsulate all communication issues
- This way the client and server applications are isolated from changes in the service contract or the SOA in general
- Note: Service interface and adapter encapsulate service contracts

Service interface and service adapter (2)

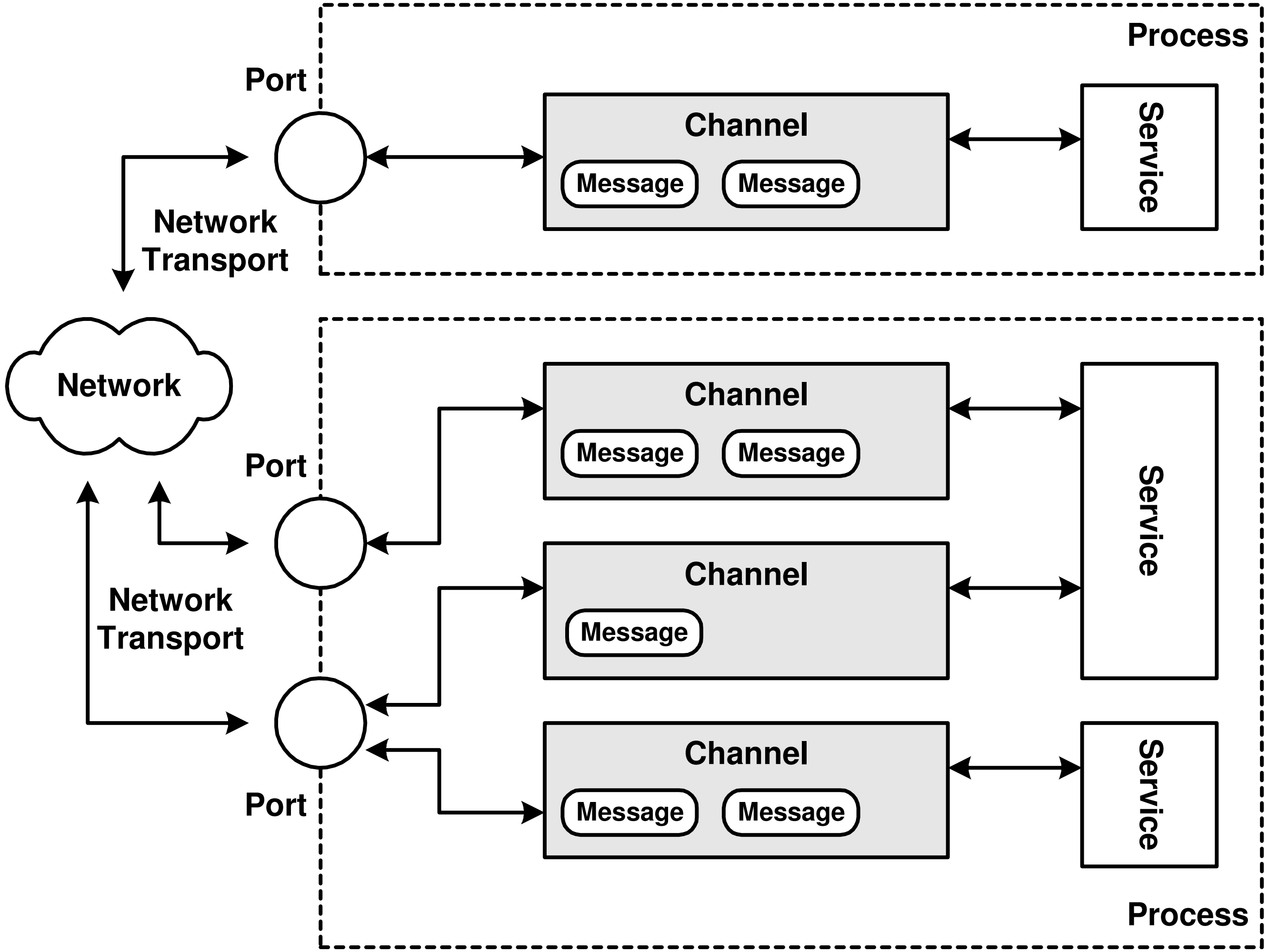


- Service adapter can be realized using the PROXY pattern
- I.e.: service adapter is a remote proxy to the service interface
- Service interface wraps the server application, following the COMPONENT WRAPPER pattern

Service interface and service adapter (3)

- Important task is handling synchronization issues:
  - Services are sometimes message-oriented, sometimes they are RPC-oriented
  - For realizing messages, sometimes reliable messaging protocols are used, sometimes unreliable asynchronous RPC is used
- Client and server applications might support many different service adapters and service interfaces, supporting different models
  - On client side, invocation asynchrony patterns or messaging patterns can be used
  - Service interface on server side must receive asynchronous messages, perform the invocation (and perhaps wait synchronously for the result), and then send a reply message to the client

Example: Indigo ports and channels



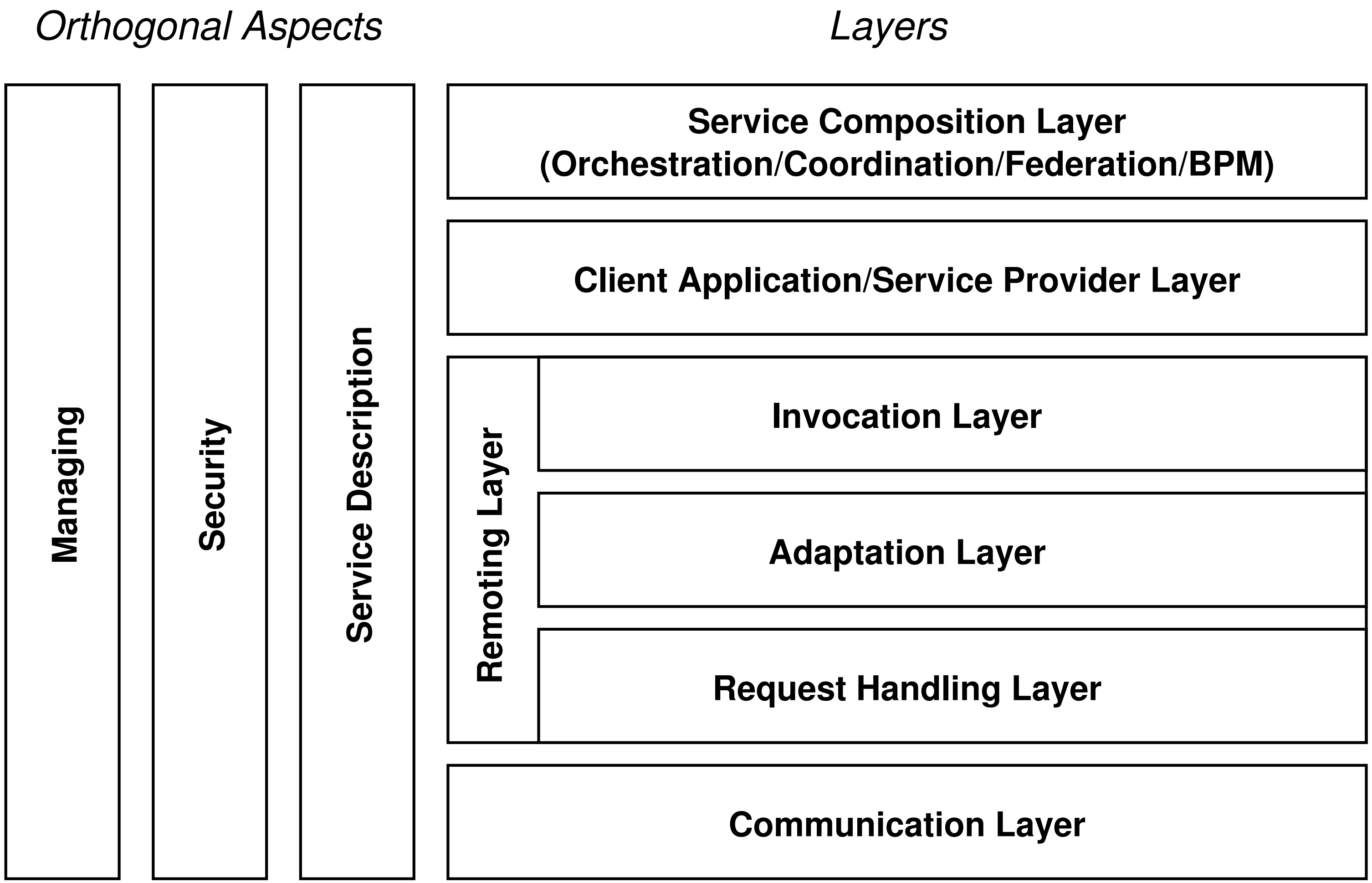
Any process can send/receive messages → ports/channels are used for realizing service interfaces und service adapters

SOA layers and basic remoting architecture

A look inside the message processing architecture of a SOA

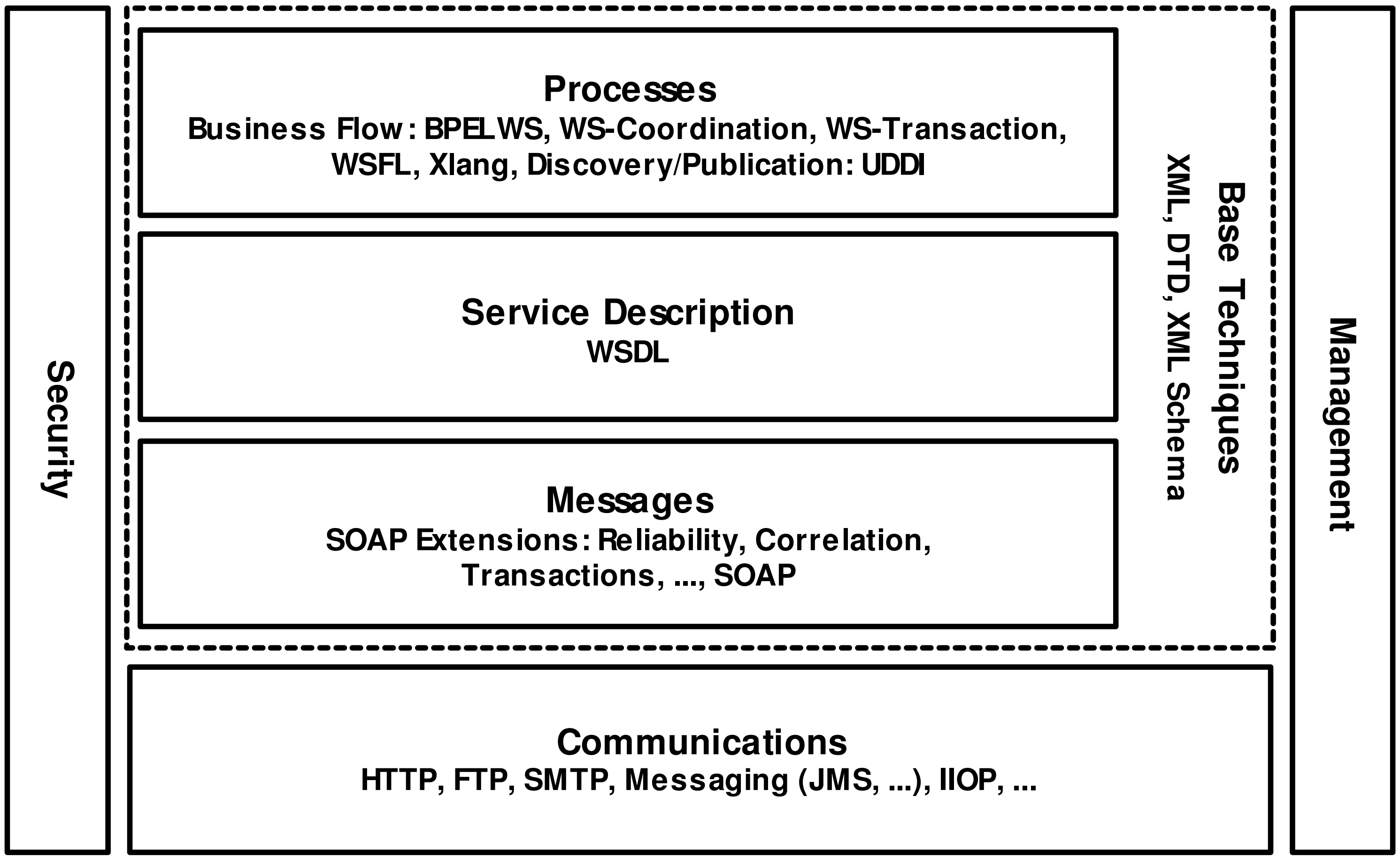
- SOAs generally have a highly symmetrical architecture on client side and server side
- Architecture follows the LAYERS pattern:
  - *Service composition* (optional): composition of services, service orchestration, service coordination, service federation, business process management (BPM)
  - *Client applications* and *service providers*
  - *Remoting*: middleware functionalities of a SOA (for instance a Web services framework), follows a BROKER architecture
  - *Communication*: defines the basic message flow and manages the operating system resources, such as connections, handles, or threads
- In addition, there are a number of orthogonal extension tasks, such as: management functionalities for services, security of services, description of services

SOA layers



Orthogonal Aspects
Layers
Managing
Security
Service Description
Service Composition Layer
(Orchestration/Coordination/Federation/BPM)
Client Application/Service Provider Layer
Remoting Layer
Invocation Layer
Adaptation Layer
Request Handling Layer
Communication Layer

Example: Typical Web Services stack



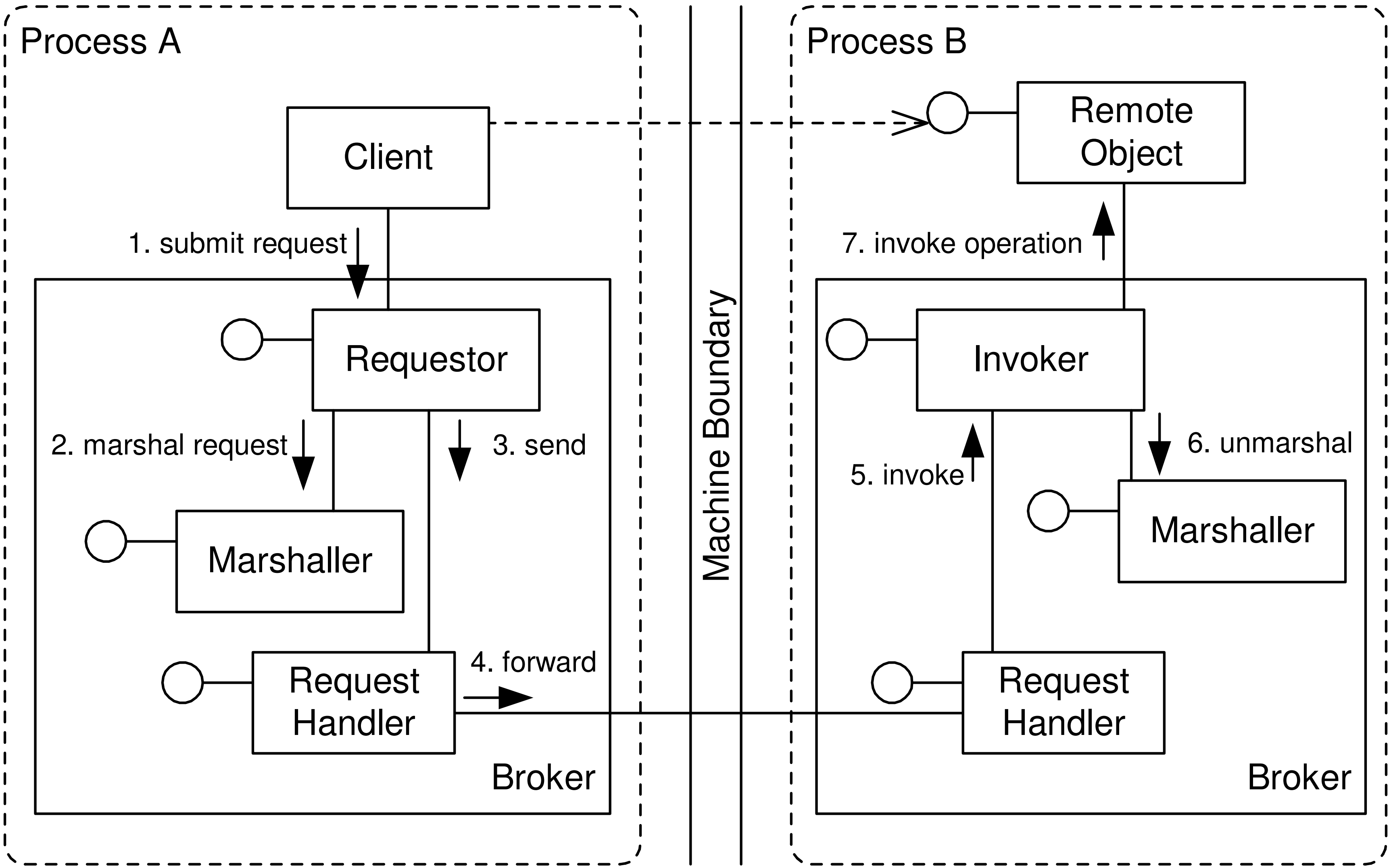
Remoting layer

- A BROKER hides and mediates all communication between the objects or components of a system
- Remoting patterns detail the BROKER architecture
- The remoting layer consists itself of three layers:
  - invocation
  - adaptation
  - request handling

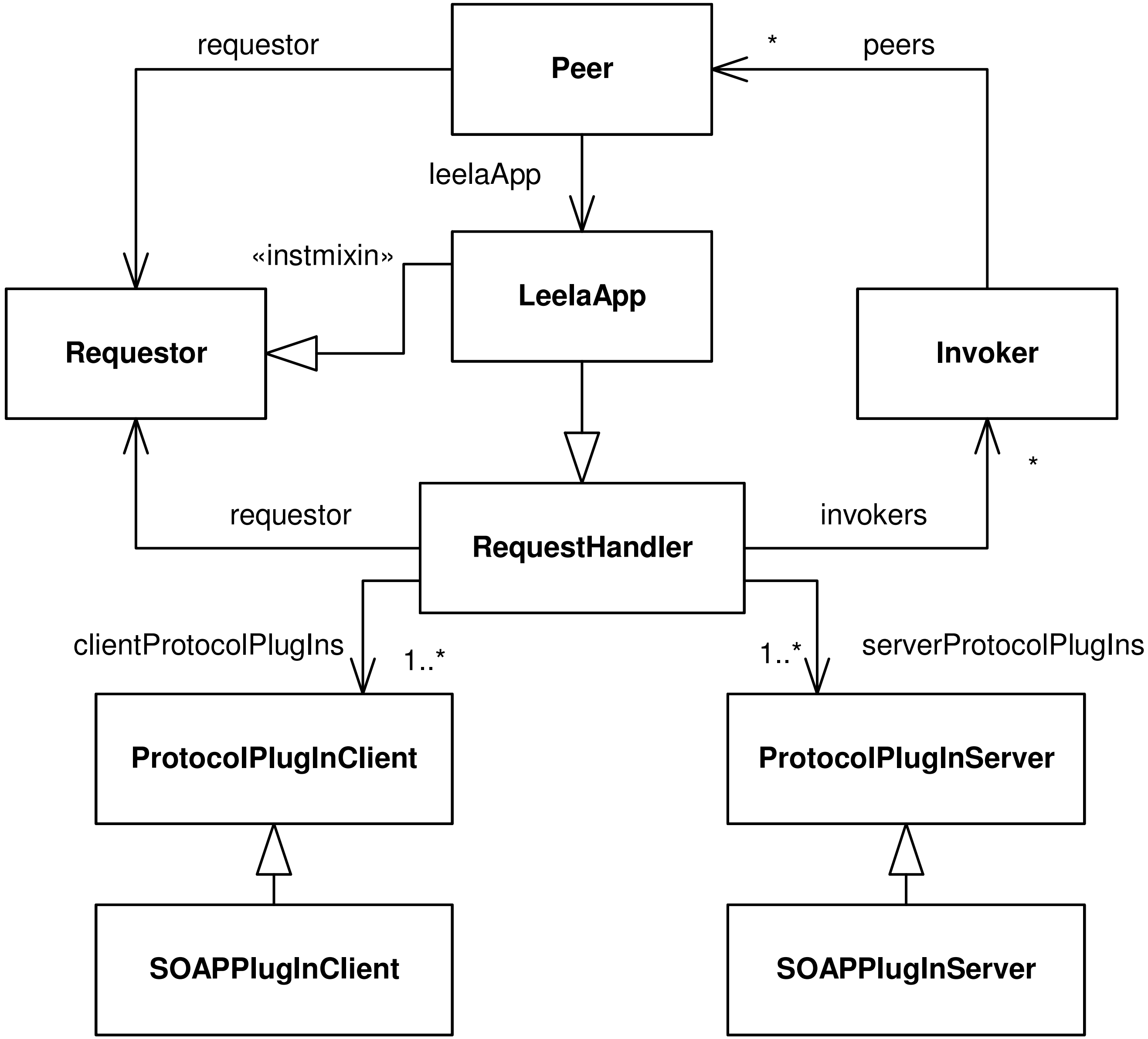
Basic remoting patterns

- REQUESTOR constructs invocation on client side
- CLIENT PROXY supports the same interface as the remote object,translates the local invocation into parameters for the REQUESTOR, and triggers the invocation
- INVOKER accepts invocations on server side and performs the invocation on the targeted remote object
- SERVER REQUEST HANDLER deals with all communication issues of a server application
- CLIENT REQUEST HANDLER handles network connections for all requestors within a client
- MARSHALLER is responsible for marshaling/demarshaling invocations on client and server side

Basic remoting patterns

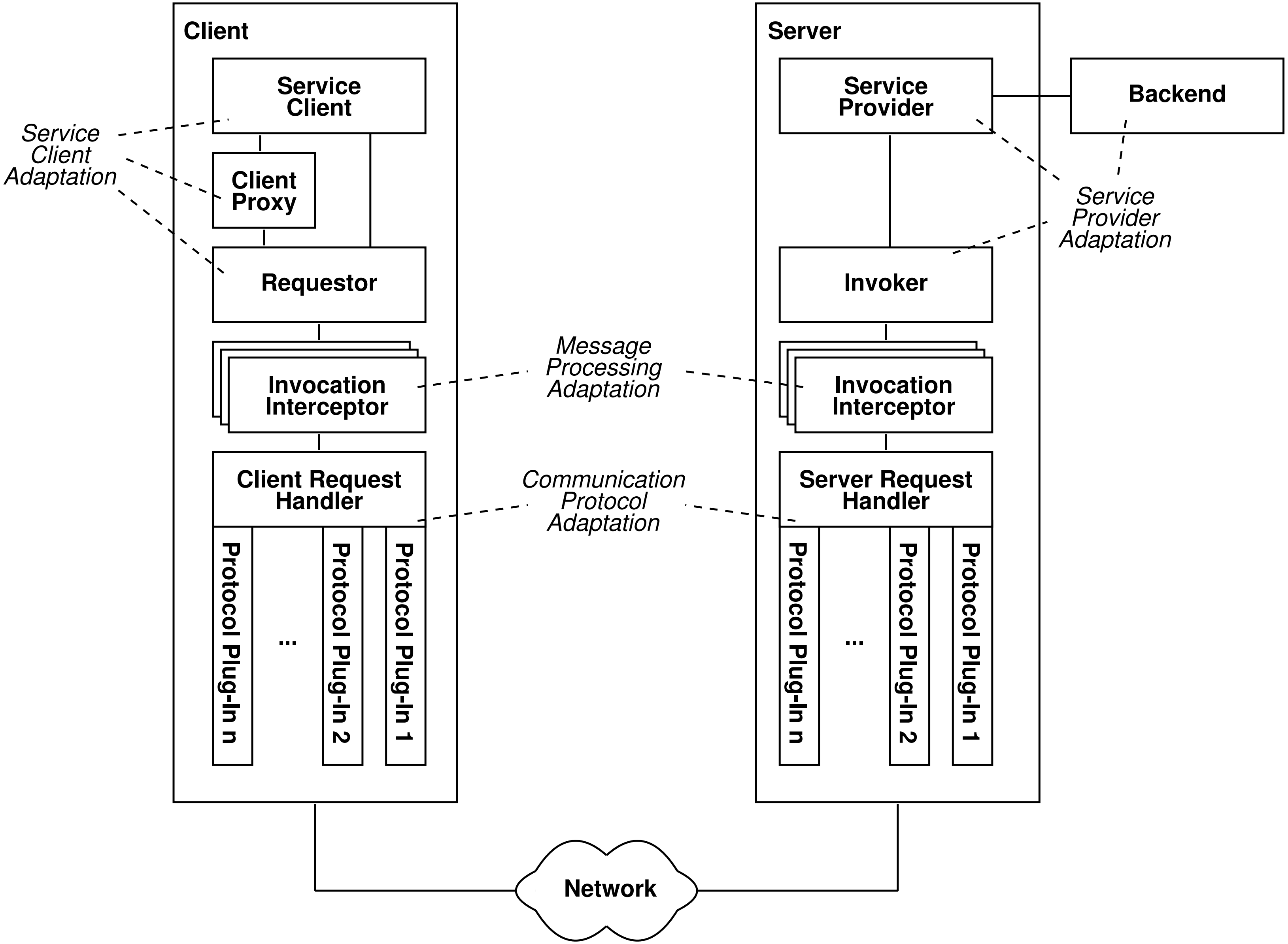


Example: Structure of the Leela communication framework



SOA variation points and adaptation

SOA variation points: Overview



Communication protocol adaptation

- A SOA allows for a number of communication protocols to be used
- And: different styles of communication, such as synchronous RPC, asynchronous RPC, messaging, publish/subscribe, etc.
- Thus, on the communication layer, we require a high flexibility regarding the protocols used
- Variation at the communication layer is usually handled via PROTOCOL PLUG-INS:
  - PROTOCOL PLUG-INS extend the CLIENT REQUEST HANDLER and SERVER REQUEST HANDLER with support for multiple, exchangeable communication protocols
  - They provide a common interface to allow them to be configured from the higher layers

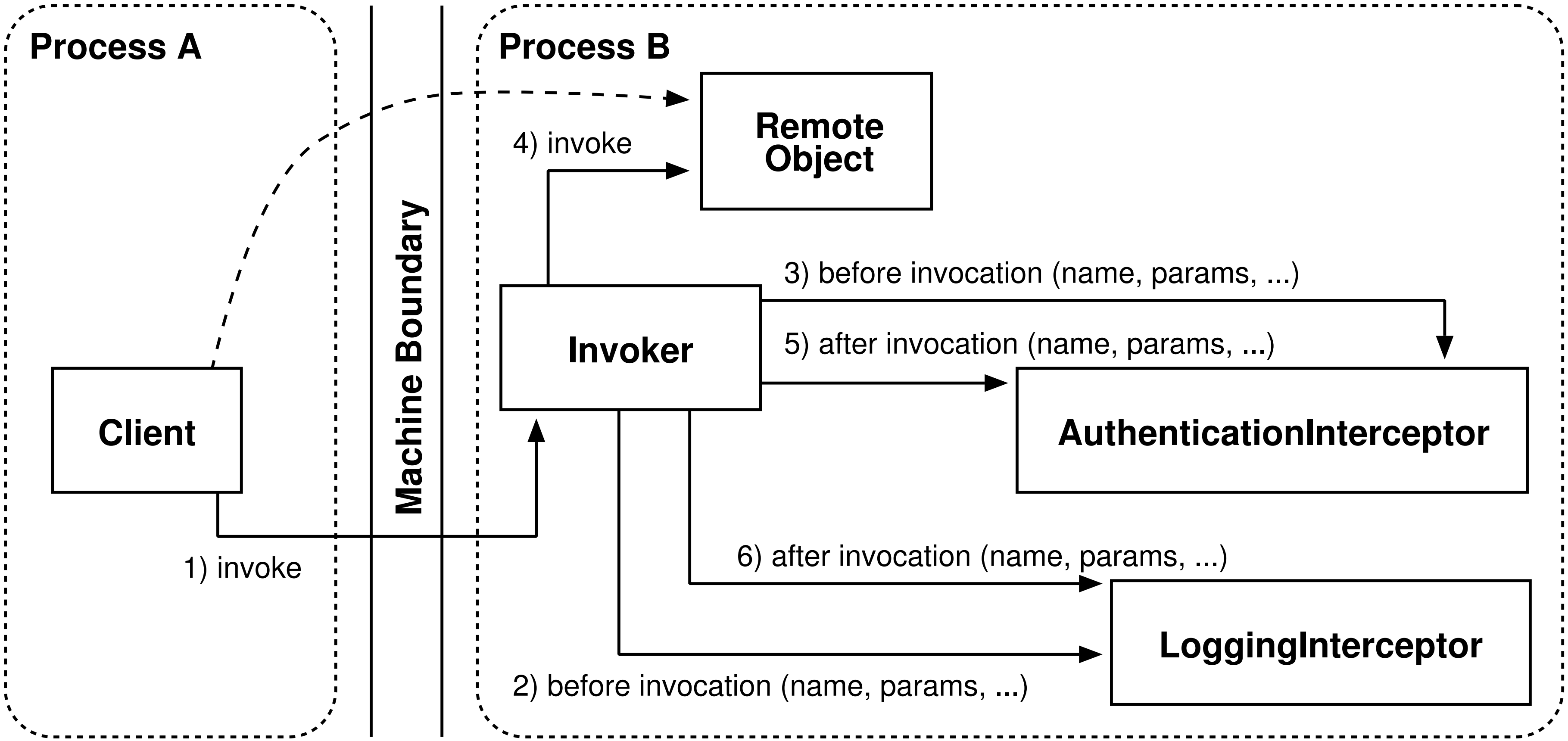
Example: Protocol integration in Web Services frameworks

- Heterogeneity of communication protocols of Web Service frameworks
  - Most Web Service frameworks provide for some extensibility at this layer
  - Slightly different REQUEST HANDLER/PROTOCOL PLUG-IN architectures
- In the default case HTTP is used as a communication protocol
- SOAP also allows for other communication protocols
- For instance: Axis supports PROTOCOL PLUG-INS for HTTP, Java Messaging Service (JMS), SMTP, and local Java invocations
- Protocol plug-ins are responsible for implementing a message queue, if needed (e.g. JMS-based messaging)

Adaptation of message processing

- Adapting the message processing is necessary . . .
  - to handle various control tasks, like management and logging
  - to handle pervasive tasks, like security
  - to deal with multiple payload formats with different marshalling rules
- These tasks need to be flexibly configurable
- Often realized using the INVOCATION INTERCEPTOR pattern:
  - INVOCATION INTERCEPTORS are automatically triggered before and after request and reply messages pass the INVOKER or REQUESTOR
  - The INTERCEPTOR intercepts the message at these spots and can add services to the invocation
- Usually: same INVOCATION INTERCEPTOR architecture on client and server side

Example: Invocation Interceptor on server side



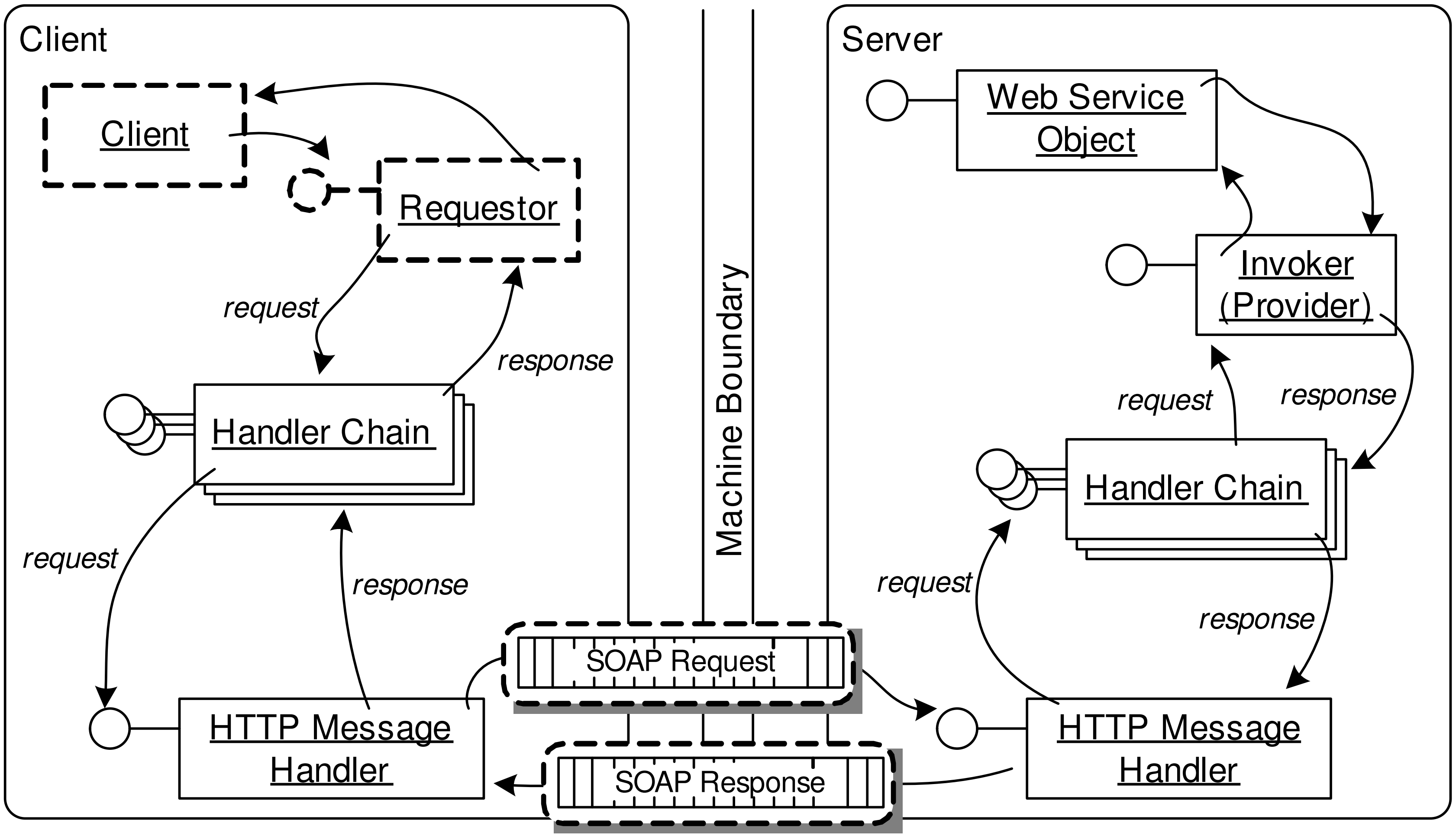
Invocation contexts

- For many tasks, we need to pass additional information between client and server
  - E.g.: For an authentication interceptor on the server side we require additional information to be supplied by the client side: the security credentials (such as user name and password)
  - These can be provided by an INVOCATION INTERCEPTOR on client side
  - However, how to transport this information from client to server?
- This is the task of the pattern INVOCATION CONTEXT:
  - The INVOCATION CONTEXT bundles contextual information in an extensible data structure
  - It is transferred between client and remote object with every remote invocation

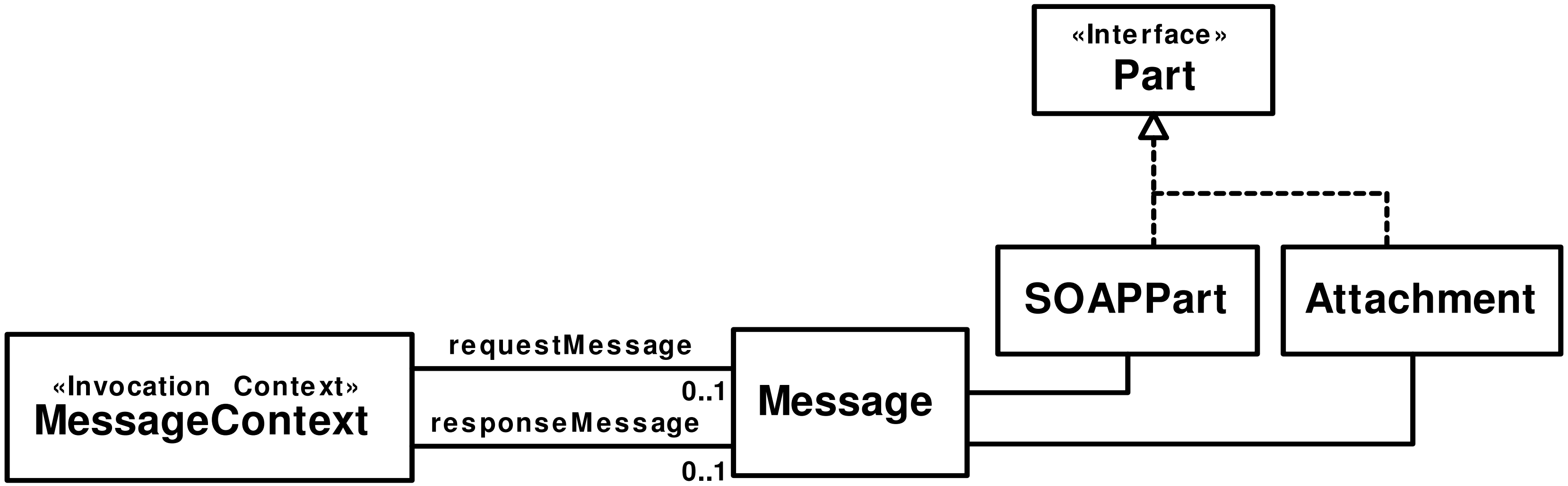
Example: Adaptation of message processing in Apache Axis

- Concerns on client and server side:
  - there are many different, orthogonal tasks to be performed for a message,
  - there is a symmetry of the tasks to be performed for request and response,
  - similar problems occur on client side and server side, and
  - the invocation scheme and add-ons have to be flexibly extensible.
- Solution: Combination of the patterns: REQUESTOR, INVOKER, INVOCATION INTERCEPTOR, CLIENT/SERVER REQUEST HANDLER, INVOCATION CONTEXT.
- INTERCEPTORS are implemented as COMMANDS are ordered in a chain

Example: Apache Axis's message processing architecture



Example: Apache Axis's message context structure



Example: Log handler in Apache Axis

```
public class LogHandler extends BasicHandler {
  ...
  public void invoke(MessageContext msgContext)
    throws AxisFault {
    ...
    if (msgContext.getPastPivot() == false) {
      start = System.currentTimeMillis();
    } else {
      logMessages(msgContext);
    }
    ...
  }
  ...
}
```

Example: Handler configuration with deployment descriptors in Apache Axis

```
<handler
  name="logger"
  type="java:org.apache.axis.handlers.LogHandler"/>
...
<chain name="myChain"/>
  <handler type="logger"/>
  <handler type="authentication"/>
</chain>
...
<service name="DateService" provider="java:RPC">
  ...
  <requestFlow>
    <handler type="myChain"/>
  </requestFlow>
</service>
```

Service provider adaptation (1)

- Service provider $=$ remote object realizing the service
- Often the service provider does not realize the service functionality solely, but instead uses one or more backends
- When a SOA is used for integration tasks, it should support multiple backend types

$\longrightarrow$ Only the service interfaces are exposed and service internals are hidden from the service client

$\longrightarrow$ Integration of any kind of backend with one common service provider model

Service provider adaptation (2)

- Service provider adaptation needs to be supported by:
  - Remote objects realizing the service and
  - INVOKER that is used for invoking them
- Common realization:
  - One INVOKER type for each backend type
  - Make INVOKERS flexibly exchangeable (e.g. using deployment descriptors)

Service provider adaptation (3)

INVOKERS used in this way realize the pattern COMPONENT WRAPPER:

- COMPONENT WRAPPER: wrap an external component using a first-class object of the programming language
- Use of COMPONENT WRAPPERS gives the application a central, white-box access point to the component
- Component access can be customized without interfering with the client or the component implementation
- Because all components are integrated in the same way, a variation point for white-box extension by component's clients is provided for each component in a system

Example: Apache Axis providers

- Almost all Web Services frameworks provide some dynamic form of deployment
- In Axis, a provider actually invokes the Web Services (a pluggable INVOKER)
- Many different providers are implemented in Axis, including those for Java, CORBA, EJB, JMS, RMI, ...
- Configured using the deployment descriptor; e.g. to select the RPC provider:

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
   xmlns:java="
     http://xml.apache.org/axis/wsdd/providers/java">
  <service name="DateService" provider="java:RPC">
    <parameter name="className"
     value="simpleDateService.DateService"/>
    <parameter name="allowedMethods" value="getDate"/>
  </service>
</deployment>
```

Service provider adaptation requires lifecycle & resource management

- Service providers and invokers need to be tightly integrated with the LIFECYCLE MANAGER: Central place for lifecycle management in the SOA
- INVOKER selects the best-suited lifecycle strategy pattern for the service
  - STATIC INSTANCES: live from application startup to its termination
  - PER-REQUEST INSTANCES: live only as long as a single invocation, advisable for most systems that access a backend
  - CLIENT-DEPENDENT INSTANCES: when session state needs to be maintained between invocations; the CLIENT DEPENDENT INSTANCE must implement a session model and a LEASING model compatible with the model of the backend
- The LIFECYCLE MANAGER should also handle resource management tasks, such as POOLING or LAZY ACQUISITION

Example: Lifecycle handling in Axis

Axis supports the following lifecycle patterns using a scope option chosen in the deployment descriptor

- PER-REQUEST INSTANCE: default, request scope
- STATIC INSTANCE: application scope
- CLIENT-DEPENDENT INSTANCE: session scope
  - Sessions are supported either by HTTP cookies or by - communication protocol independent - SOAP headers
  - Each session object has a timeout (which can be set to a certain amount of milliseconds). After the timeout expires, the session is invalidated. A method touch can be invoked on the session object, which re-news the lease.

Service client adaptation (1)

- Service clients should also be adaptable
- Goals are different than on the server side:
  - independence of service realization
  - loose coupling
- Service client adaptation is mainly reached by LOOKUP of services and well-defined INTERFACE DESCRIPTIONS

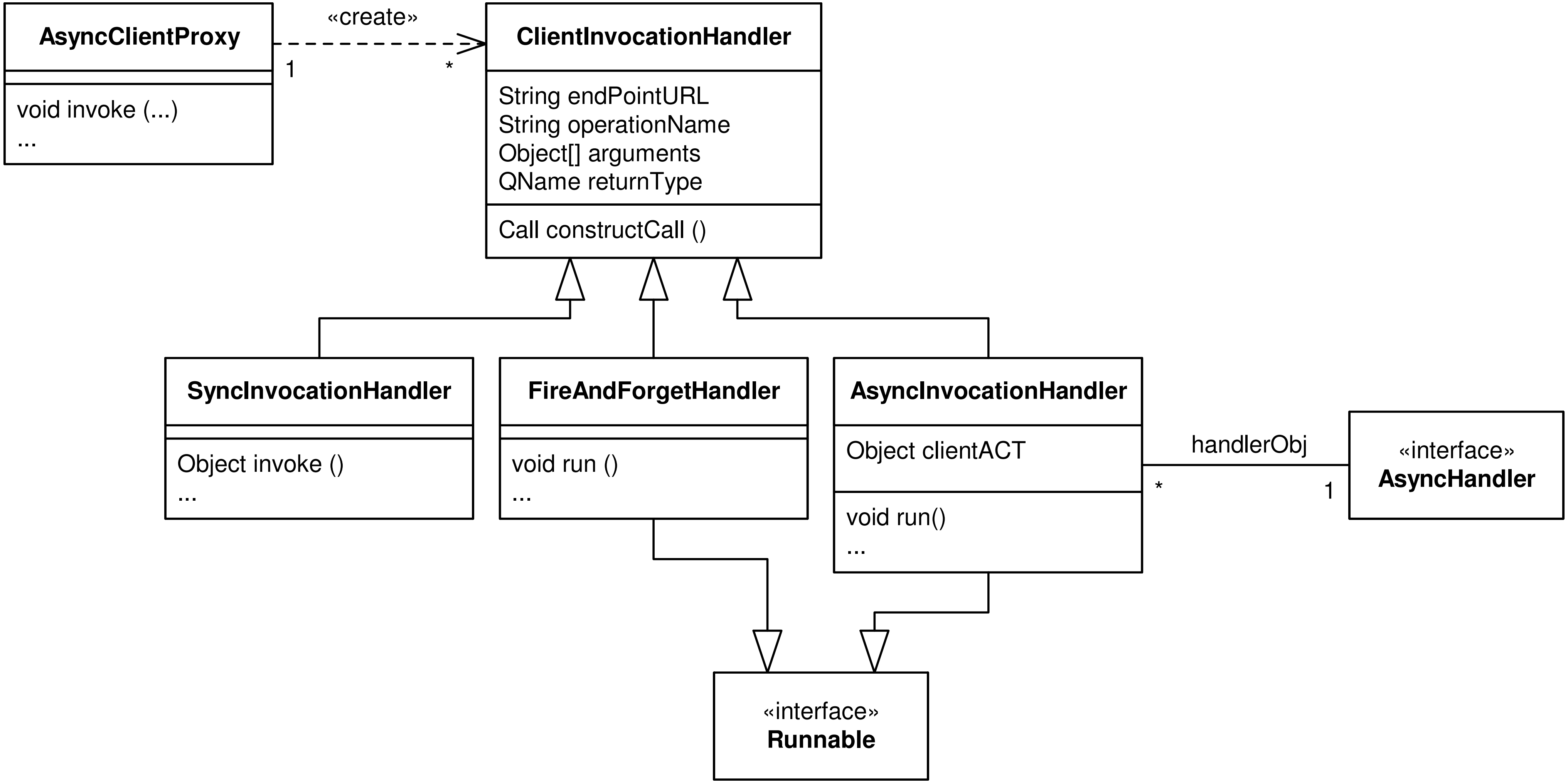
Service client adaptation (2)

- Other aspects of service client adaptation:
  - Flexible (e.g. on-the-fly) generation of CLIENT PROXIES
  - Direct use of REQUESTORS to construct invocations on-the-fly
- Client must be adapted to how the result is sent back (if there is any)
  - Synchronous blocking
  - Client invocation asynchrony patterns: FIRE AND FORGET, SYNC WITH SERVER, POLL OBJECT, and RESULT CALLBACK

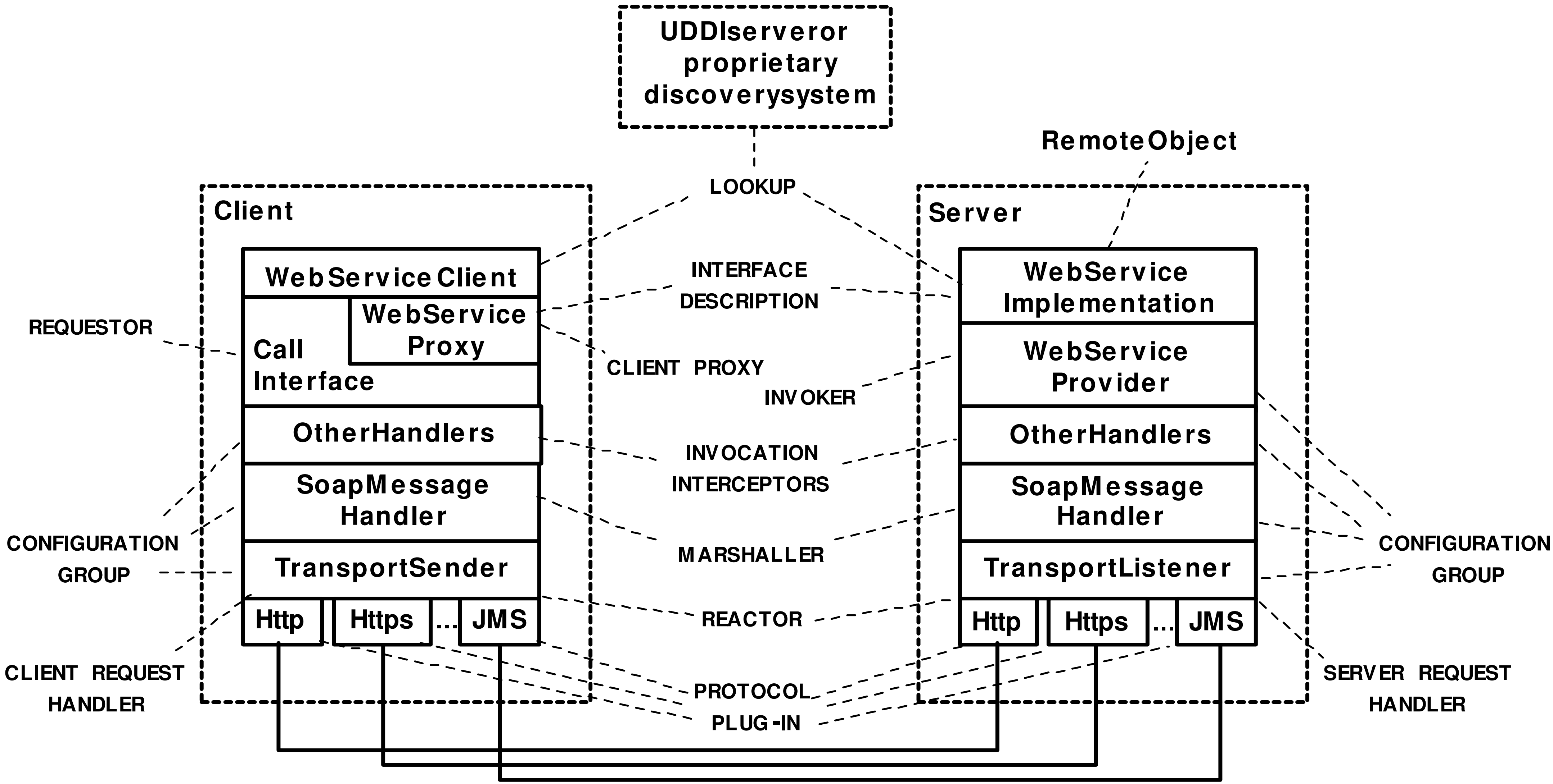
Example: Service client adaptation and client-side asynchrony

- Axis does not support client-side asynchrony patterns without using a messaging protocol
- SAIWS: Asynchrony layer on top of synchronous invocation layer provided by Axis, `http://saiws.sourceforge.net`
- Two kinds of REQUESTORS:
  - one for synchronous invocations
  - one for asynchronous invocations

Example: SAIWS – Invocation handlers



Overview: Remoting Patterns in typical Web Services architectures



SOA and business processes

Business process management

- Process Engineering aims at optimizing the business processes of an organization
  - Business processes need to be implemented quickly
  - Cope with a dynamic business environment
- Latest definitions of the term Business Process Management (BPM) illustrate that workflow technology brings together the formerly separate worlds of . . .
  - organizational design
  - technical design

  Business Process Management implies, on a technical level, the design of technological platforms that allow organizational flexibility

BPM and flexible technology platforms

- Design of flexible technology platforms for BPM is already strongly demanded by many industries
  - Time to react on organizational change requirements is becoming shorter and shorter
  - IT of an organization is the key enabling factor
  - Organizationally inflexible technology implies cost-intense implementation of organizational changes
- Many enterprises are shifting to process-oriented organizations
- IT platforms have to consider this process approach conceptually

→ It is important to address the link between business processes and SOA

Process engineering & SOA

- At the top layer of the SOA architecture: introduce the decoupling of process control logic by a service orchestration layer

$\longrightarrow$ Process-driven concept for SOA

  – Decoupling process logic implies another level of organizational flexibility

  – Perspectives of technical architecture and organizational architecture merge via the process paradigm

A high-level pattern perspective (1)

- Most abstract pattern perspective: several patterns that follow a process-oriented approach:
  - MANAGED COLLABORATION
  - MANAGED PUBLIC PROCESSES
  - MANAGED PUBLIC AND PRIVATE PROCESSES
  - EXPOSED BUSINESS SERVICES
- Mapped to SOA these patterns address variations of service orchestration within an enterprise or across enterprise boundaries
- However: they represent design guidelines at a high level where principle collaborative decisions are made at the business level
  - Explain what collaborative patterns are appropriate for a certain business problem
  - Help finding appropriate patterns of service collaboration

A high-level pattern perspective (2)

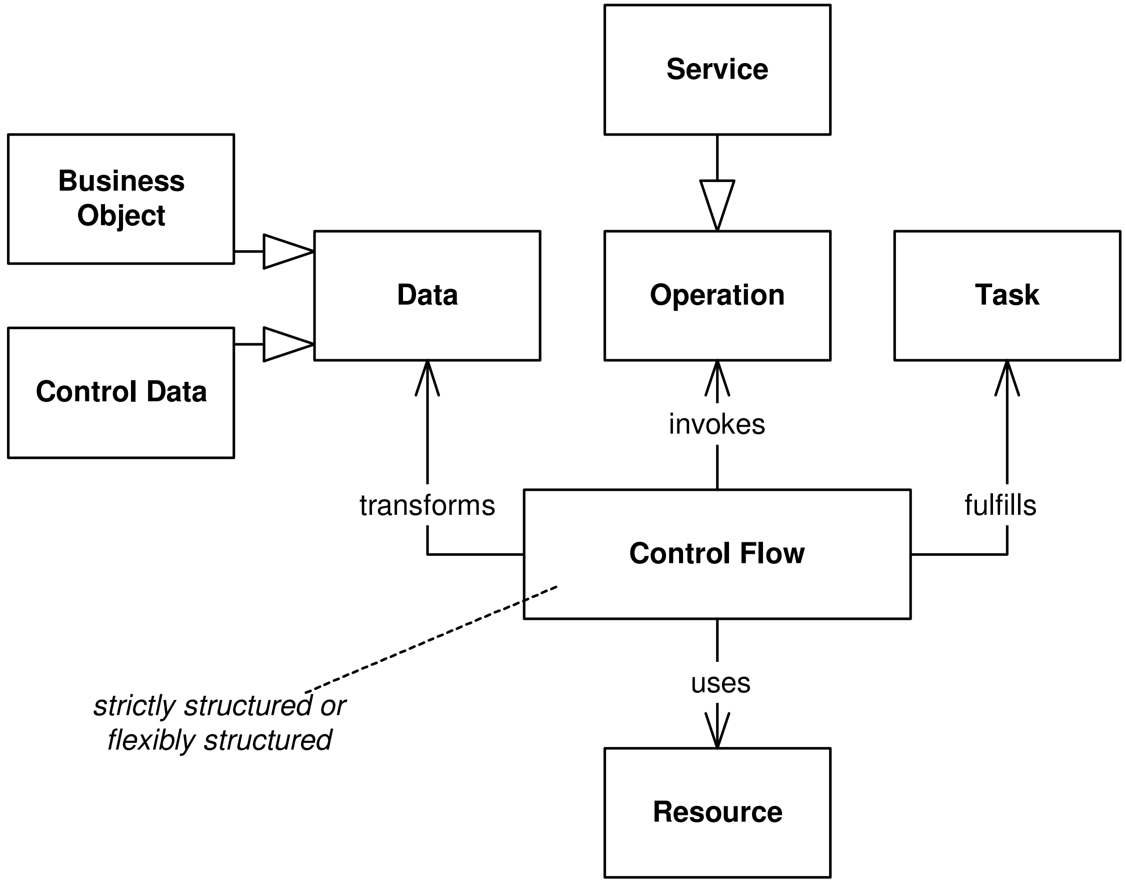
- Concerning integration of SOA and business processes there are several important integration patterns, such as:
  - ROUTER
  - BROKER
  - MANAGED PROCESS
- General patterns that are, in combination, suitable for bridging the two views of SOA and business processes
- On the following slide we will elaborate on this in more detail

Integration of services and processes

Integrating services and processes

- Fundamentally, a process-aware information system is shaped by 5 perspectives:
  - data (or information)
  - resource (or organization)
  - control flow (or process)
  - task (or function)
  - operation (or application)
- Basic Mapping to the SOA approach:
  - services are a specialization of the general operation perspective
  - process control flow orchestrates the services via different process steps
  - operations executed by tasks in a control flow correspond to service invocations

Overview: Link between SOA and workflow processes



Mapping the data perspective

- In the data perspective distinguish between:
  - process control data
  - business objects that are transformed via the process flow
- Example:
  - Business object: a customer order that is being processed via a process flow
  - The actual processing of that order is controlled by control data that depicts the routing rules, for instance
  - Each process step can be interpreted as a certain state of the business object
- The SOA's service orchestration has to deal with control data and business objects being transformed and passed from one orchestration step to the next one

Patterns for the data perspective (1)

Business objects:

- Business objects are manipulated via the process steps (represented by services)
- Business objects following the ENTITY pattern represent entities in a REPOSITORY
- In the REPOSITORY: business objects depict a CANONICAL DATA MODEL for storing process relevant business data

Process control data

- Many process engines struggle with changes to control data at runtime

$\longrightarrow$ GENERIC PROCESS CONTROL STRUCTURE pattern: design of a control data structure that is unlikely to change

Patterns for the data perspective (2)

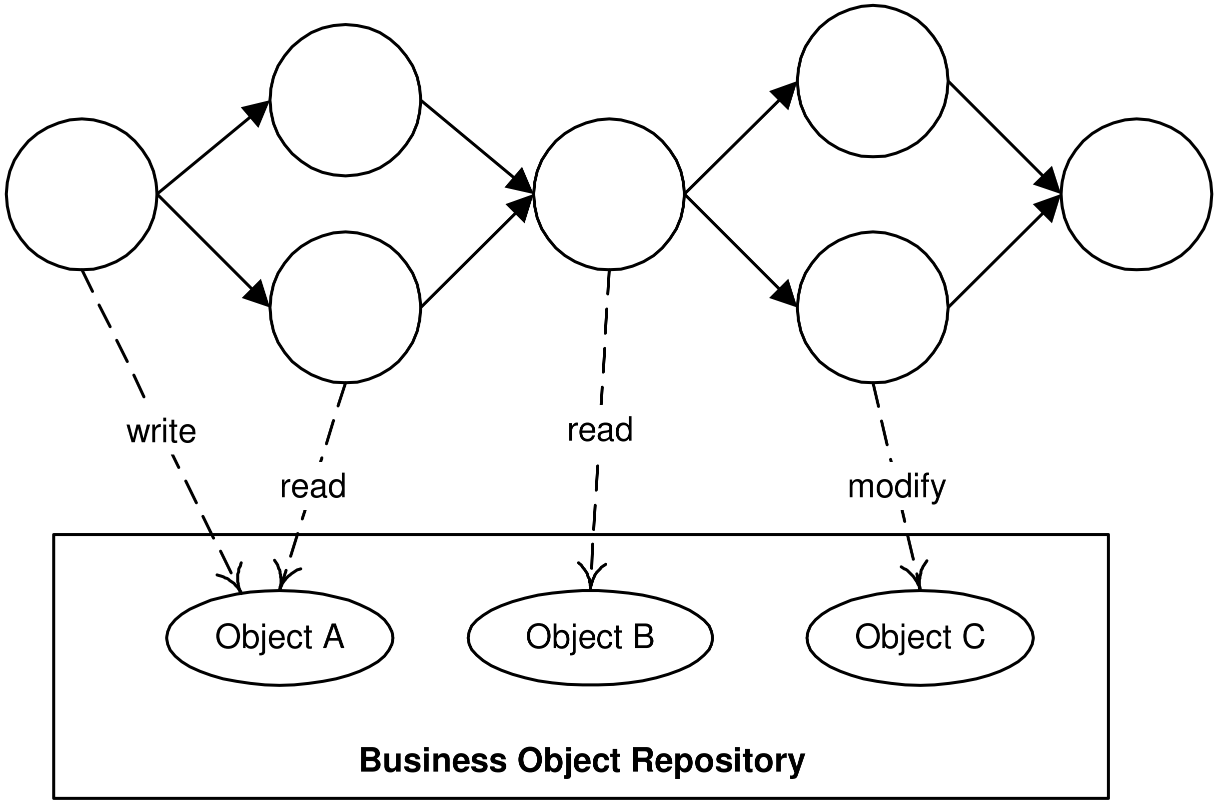
Integrating business objects and process control data:

- Business objects can concurrently be modified by different process instances

$\longrightarrow$ BUSINESS OBJECT REFERENCES must be part of the control data

- Pointers to business objects in a REPOSITORY and the concrete business objects can thus be accessed concurrently via these references

Business objects being accessed via process steps

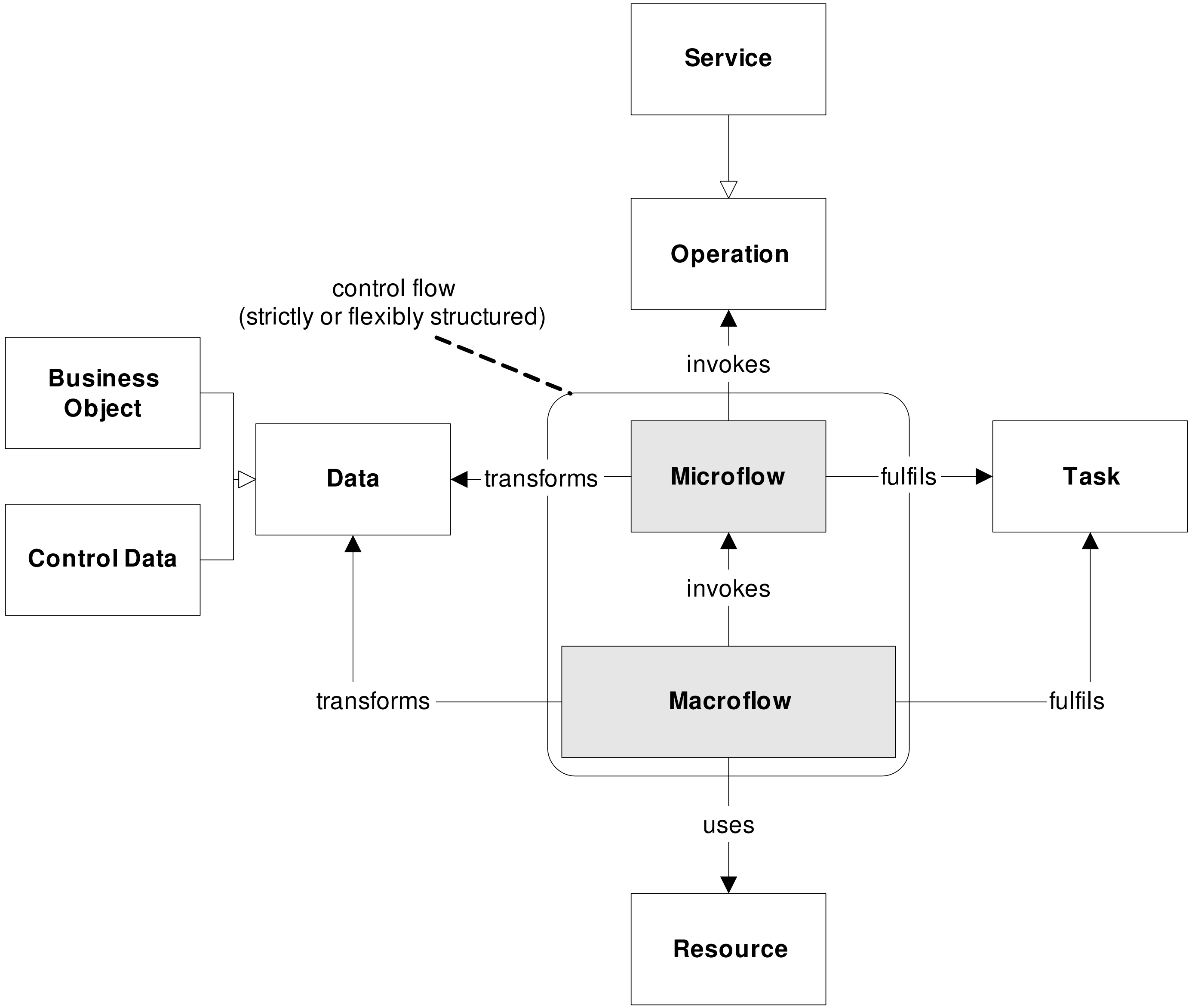


Mapping the control flow perspective

- Control flow perspective is captured by a process engine
- In order to create the link between an activity of a process and a service, integration logic is required (represented by a process flow)
- We distinguish between two general types of process flow:
  - *Macroflow* representing the higher-level business process
  - *Microflow* addressing the process flow within a macroflow activity
- Note: this is a conceptual decision in order to be able to design process steps at the right level of granularity
  - Macroflow $\sim$ long running business process level
  - Microflow $\sim$ short running, more technical level

$\longrightarrow$ Important for separating the business problems from the technical problems

Adding macroflow and microflow to the mapping



Control flow design

Control flow design (at microflow and macroflow level) usually follows (some of) the workflow patterns:

Basic Control Flow Patterns
- Sequence
- Parallel Split
- Synchronization
- Exclusive Choice

Advanced Branching and Synchronization Patterns
- Multi-choice
- Synchronizing merge
- Multi-merge
- Discriminator

Structural Patterns
- Arbitrary cycles
- Implicit termination

Cancellation Patterns
- Cancel activity
- Cancel case

State Based Patterns
- Deferred choice
- Interleaved parallel routing
- Milestone

Patterns Involving Multiple Instances
- Multiple instances without synchronization
- Multiple instances with a priori design time knowledge
- Multiple instances with a priori design runtime knowledge
- Multiple instances without a priori design runtime knowledge

Further architectural control flow patterns

- ACTIVITY INTERRUPT: interrupting the processing of an activity without the loss of data
- PROCESS INTERRUPT TRANSITION: terminating a process in a controlled way
- PROCESS BASED ERROR MANAGEMENT: managing errors returned by an invoked service via the process flow

Patterns for modeling the microflow

- At the microflow level, we must:
  - route requests of service invocations sent by a process-step to the right endpoint
  - route the corresponding responses backwards
  - perform data transformation
- Technical solutions:
  - The request for service invocation sent by the process-step must be routed to the right endpoint, which is done by a BROKER
  - In message-oriented communication between a process engine and a service, various messaging patterns are used: MESSAGE ROUTER, MESSAGE TRANSLATOR, and their specializations like CONTENT-BASED ROUTER, DYNAMIC ROUTER, ENVELOPE WRAPPER, CONTENT ENRICHER

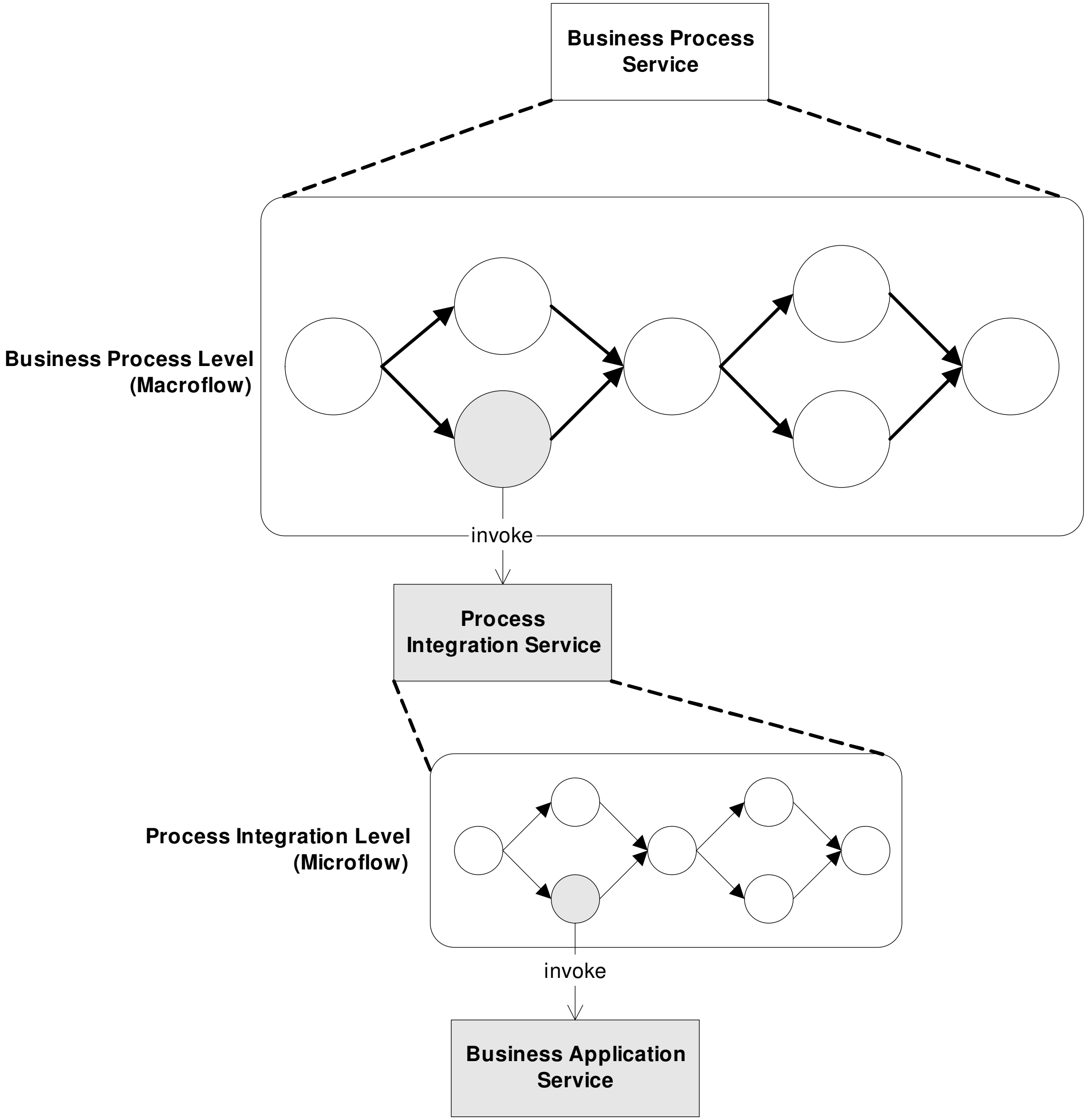
Process service levels (1)

- A process flow orchestrates the service invocations
- A business process may be exposed itself as a service
  - A process has a well defined service INTERFACE DESCRIPTION

→ The result are several levels of service invocation:

  - *business process service* – a business process being exposed as a service
  - *process integration service* – a process integration logic at the microflow level
  - *business application service* – a service that is offering functionality of a business application

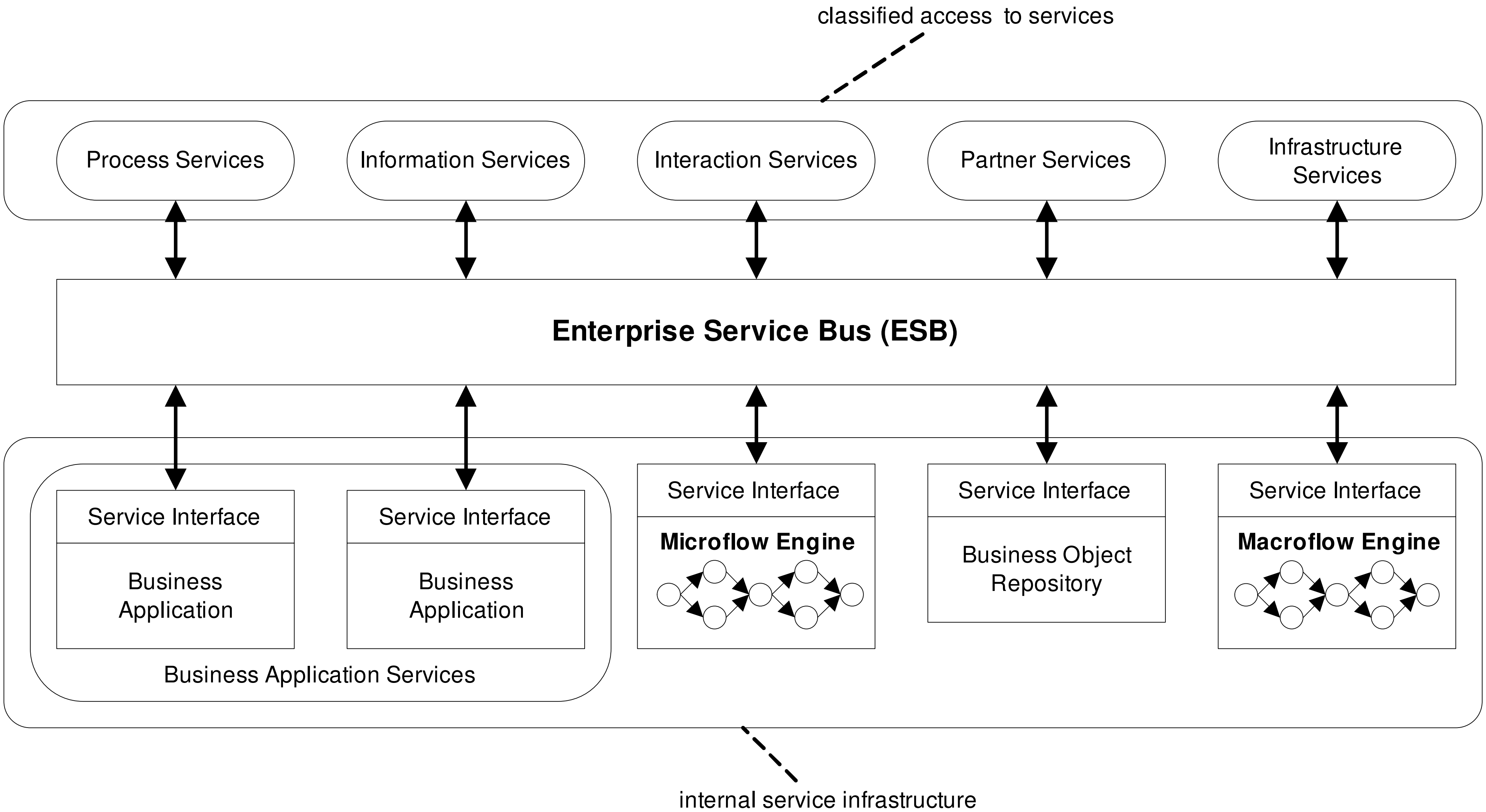
Process service levels (2)



Enterprise Service Bus (1)

- ENTERPRISE SERVICE BUS (ESB):
  - Architectural pattern that integrates concepts of SOA, EAI, and workflow management
  - Based on MESSAGE BUS pattern
- Various components connect to a service bus via their service interfaces
- Service adapters are used to connect those components to the bus
- Service bus handles service requests
- Represents a message-based ROUTER and/or BROKER

Enterprise Service Bus (2)



Enterprise Service Bus (3)

Service requests are routed to appropriate components connected to the bus, where services are invoked $\longrightarrow$ ESB can act as a:

- CONTENT-BASED ROUTER
- MESSAGE FILTER
- DYNAMIC ROUTER
- AGGREGATOR
- MESSAGE BROKER
- ...or other message routing patterns

Enterprise Service Bus (4)

- Message transformation patterns are applied by the bus to integrate different service interfaces:
  - NORMALIZER
  - ENVELOPE WRAPPER
  - CONTENT ENRICHER
- Often a repository of business objects is connected to the service bus

Process integration services (1)

Process integration services $=$ connection between a macroflow activity of a business process and a service interface in the backend

- Backend services can be developed and enhanced independently over time
- The result of the service invocation may be stored in a business object
- Control data, based on the service result, containing the BUSINESS OBJECT REFERENCE, must be passed to the calling macroflow activity

$\longrightarrow$ Integration logic is required to establish the communication between the backend service and the macroflow activity

- routing
- data transformation

$\longrightarrow$ Again, done using message routing patterns

Process integration services (2)

- Business objects relevant to microflows and macroflows form the CANONICAL DATA MODEL for storing process relevant business data
- Flexible concept for process integration services that can be adapted according to changing workload

Process integration services for process engines (1)

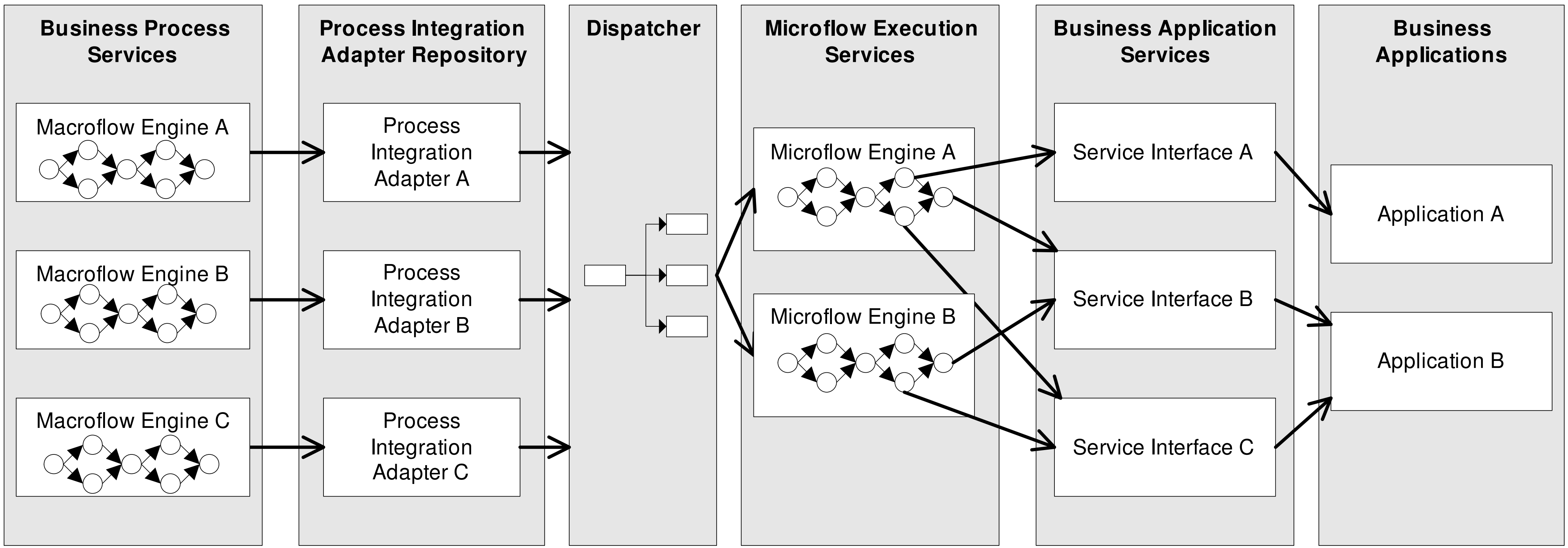
In larger architectures there might be several process engines involved for microflows and macroflows that need to be connected

- Introduce a process integration service ADAPTER for each macroflow engine
- A MESSAGE DISPATCHER distributes process integration service requests to different microflow engines
- Integration logic is executed by the PROCESS MANAGER of the microflow engine
- The PROCESS MANAGER coordinates the integration steps and invokes the business services in the backend

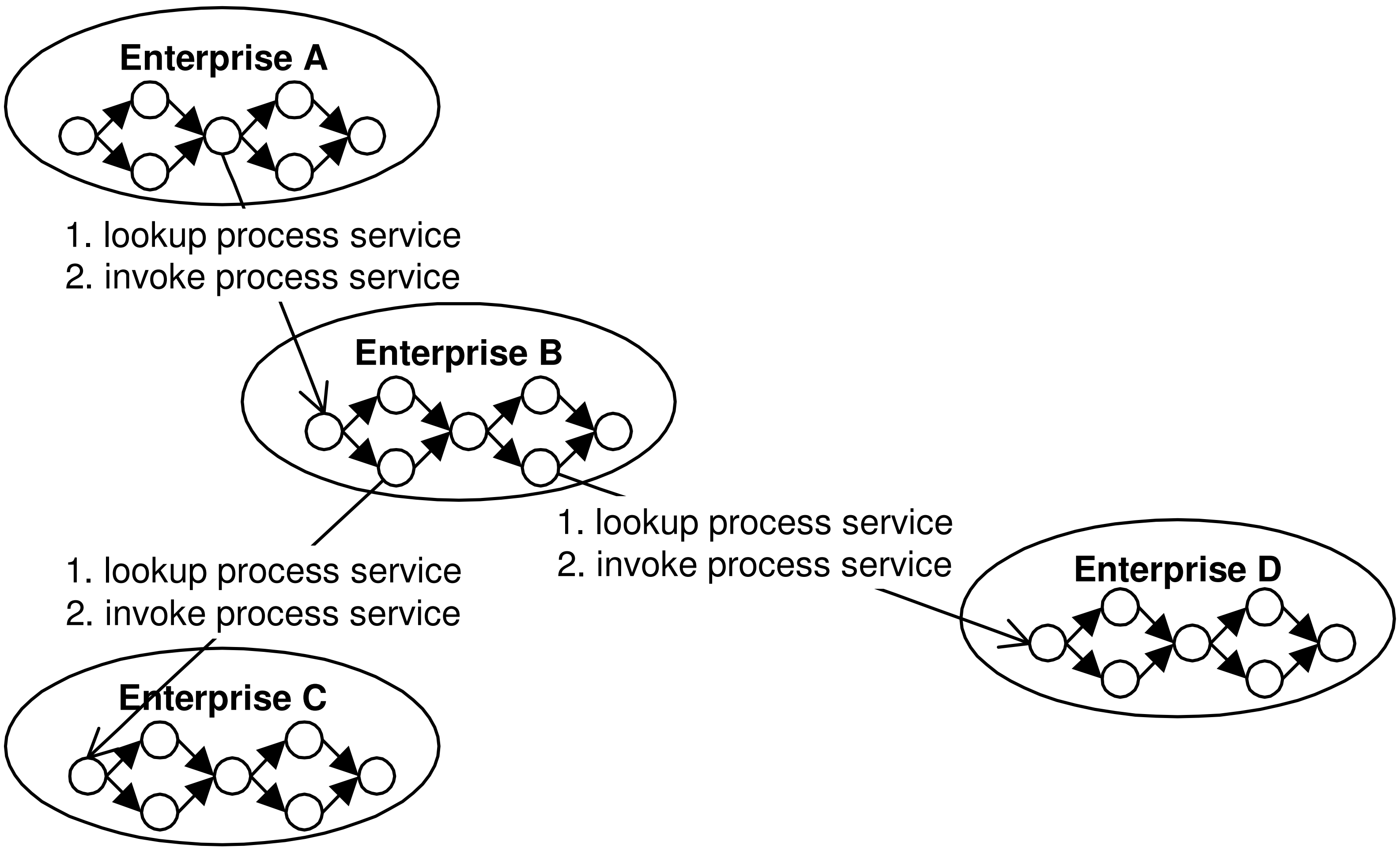
Process integration services for process engines (2)

- A REPOSITORY of process integration adapters contains all available adapters
- REPOSITORY and MESSAGE DISPATCHER are CONFIGURABLE COMPONENTS $\longrightarrow$ thus administration and configuration is possible during runtime
- The request and responses are related to a specific macroflow activity by an ASYNCHRONOUS COMPLETION TOKEN (or CORRELATION IDENTIFIER)
- Often there is only one macroflow and microflow engine $\longrightarrow$ dispatcher(s) might be superfluous

Macroflow/microflow engine integration



Cross-organizational processes



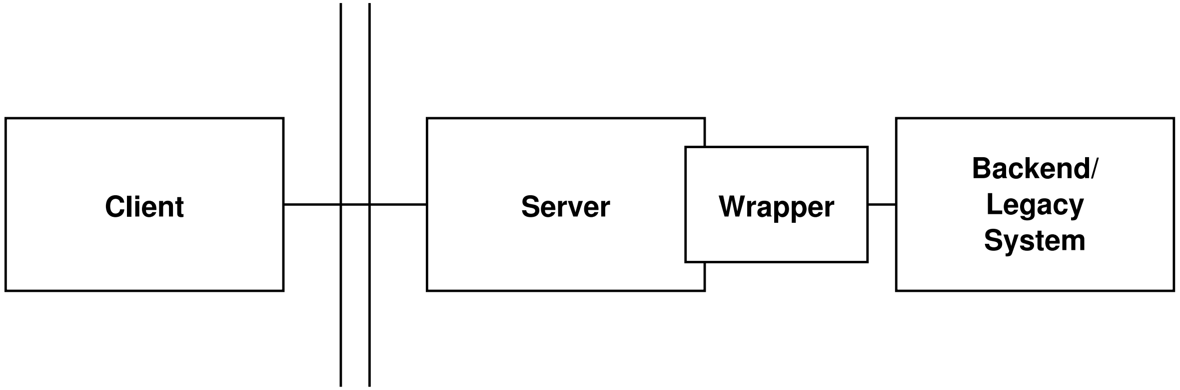
Composing SOAs

Composing multiple SOAs

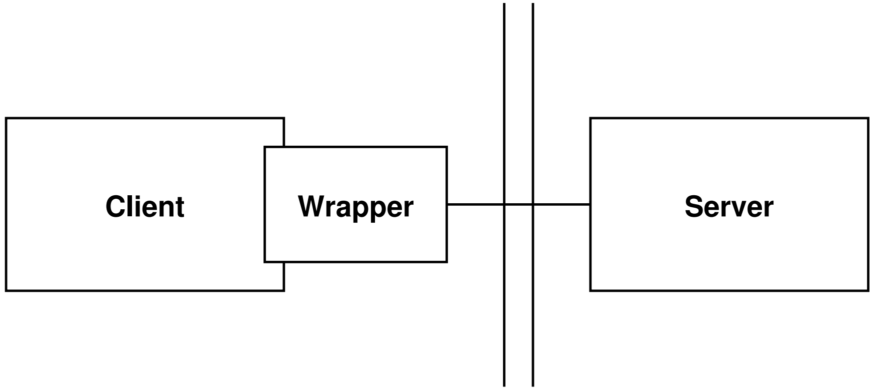
- In the enterprise scope, often multiple SOAs and other (distributed) systems need to be composed to work together
- Internally follow a similar approach as process integration services discussed before:
  - Wrap another system just like the wrapping of backends
  - The backend does not need to be a legacy system or another non-SOA participant
  - The backend can be another service as well
  - This kind of service composition is a distributed variant of the pattern COMPONENT WRAPPER

Wrapping SOAs

- Distributed COMPONENT WRAPPER:

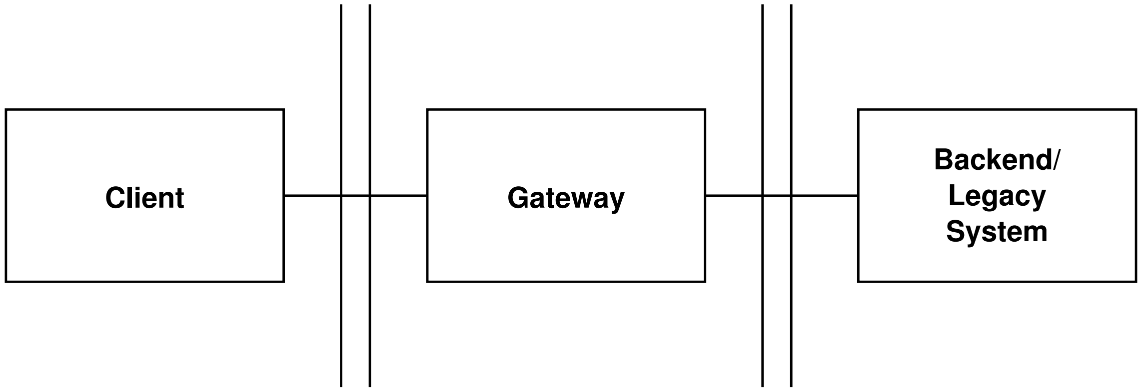


- In case the server cannot be adapted, the wrapper needs to be provided in the client to adapt to an interface provided by a server:



Gateways

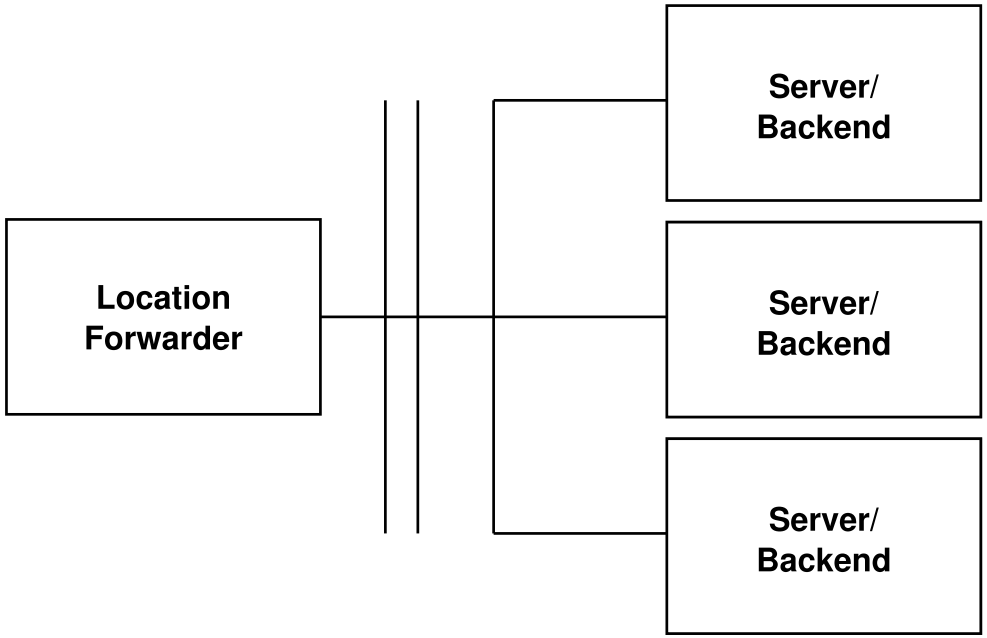
- Alternative to client-based or server-based wrappers: gateways
- Gateway $=$ intermediary component, outside of client and server
  - Translates non-SOA invocations into SOA messages, and vice versa
  - Also used for extra tasks, such as routing, mapping RPC invocations to asynchronous messages (queuing up invocations), mapping asynchronous messages to RPC invocations (de-queuing invocations), temporarily storing messages, logging, etc.



Location forwarder (1)

- LOCATION FORWARDER pattern: forward invocations to a remote object in another server application
  - E.g.: remote objects that the INVOKER cannot resolve locally
  - LOCATION FORWARDER looks up the actual location of the remote object based on its OBJECT ID
  - Result: ABSOLUTE OBJECT REFERENCE of another remote object
- LOCATION FORWARDER has two options:
  - Send the client an update notification about the new location
  - Transparently forward the invocation to the new location.
- Can be used as part of a SOA service to connect to other services or backends
- Alternatively, it can be used on a gateway, e.g. to realize routing or fault tolerance measures

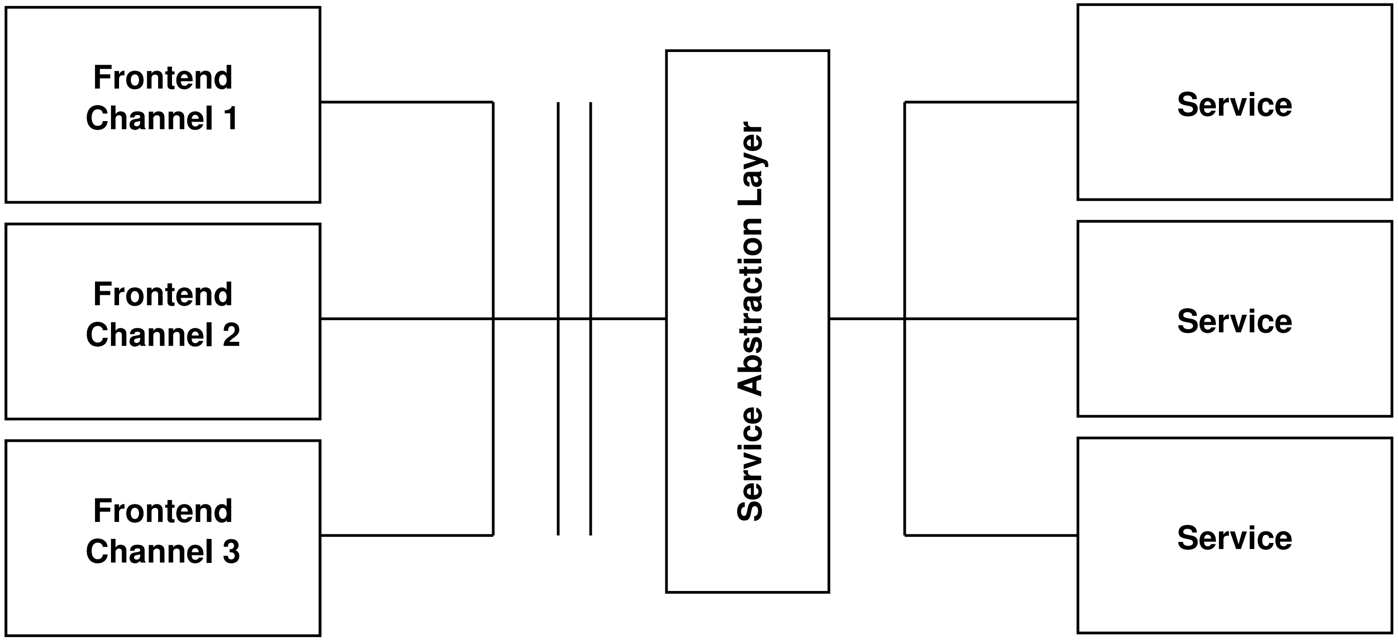
Location forwarder (2)



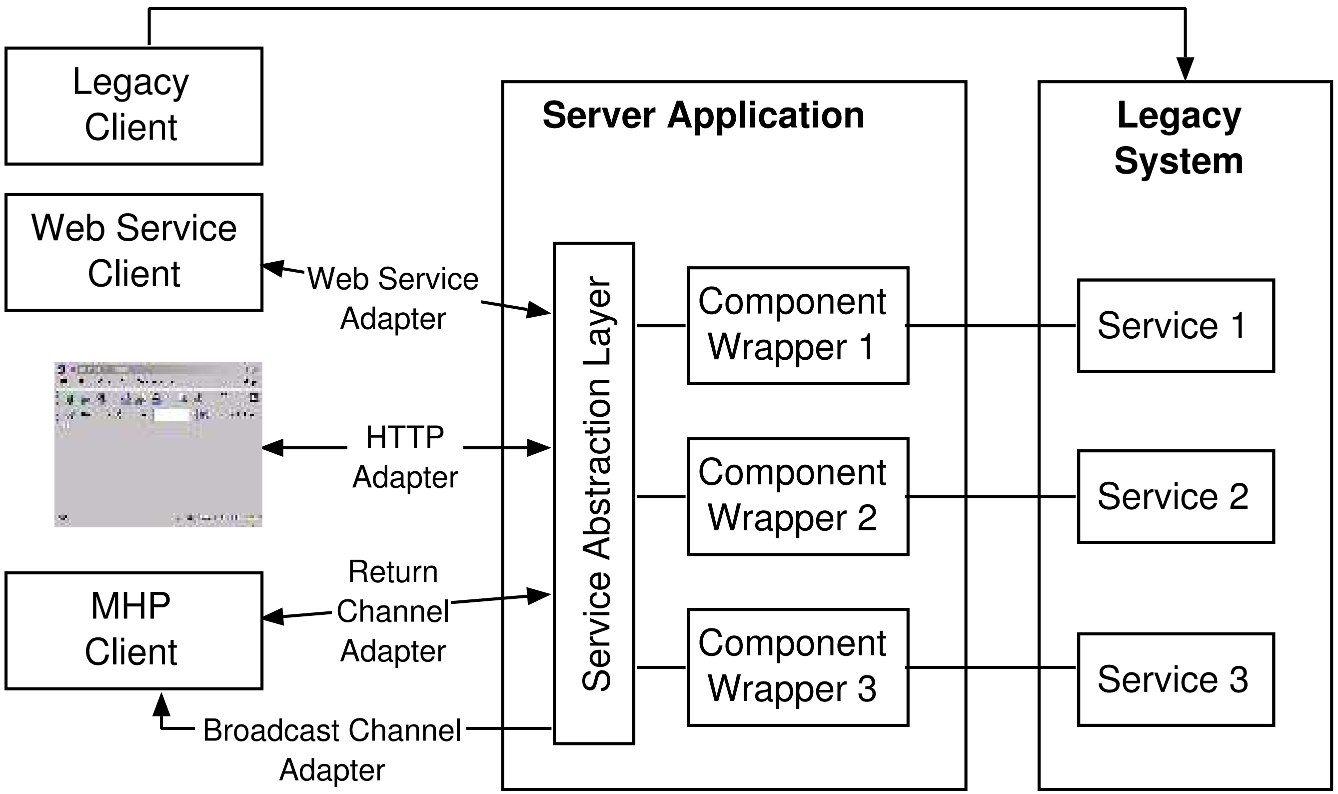
Service abstraction layer (1)

- Sometimes a number of different frontends need to access one service
- Special variant: each of the frontends is a different channel, such as:
  - Web services invocation channel
  - Web presentations channel
  - CORBA channel
  - proprietary protocol channel
  - ...
- Introduce a SERVICE ABSTRACTION LAYER: extra layer to the application logic tier containing the logic to receive and delegate requests.

Service abstraction layer (2)



Example: Composing HTTP, MHP, Web Service, and Legacy Channels



Conclusions

- Better understanding of service-oriented architectures by mapping them to the conceptual space of patterns from various domains
- Patters are successful solutions that have proven their value in numerous architectures
- We surveyed and explained the “timeless” concepts in SOAs, apart from technology details
  - Technically detailed but yet technology-neutral approach
  - Informally described the cornerstones of a SOA reference architecture
- Because patterns are solution guidelines, the patterns are also useful as SOA design guidelines

Further Reading

- Paper summarizing some of the tutorial contents: Uwe Zdun, Carsten Hentrich, Wil van der Aalst: A Survey of Patterns for Service-Oriented Architectures, to appear in: International Journal of Internet Protocol Technology, Inderscience

- M. Voelter, M. Kircher, and U. Zdun. Remoting Patterns. Pattern Series. John Wiley and Sons, 2004.

- G. Hohpe and B. Woolf. Enterprise Integration Patterns. Addison-Wesley, 2003.

- C. Hentrich. Six patterns for process-driven architectures. In Proceedings of the 9th Conference on Pattern Languages of Programs (EuroPLoP 2004), 2004.

Further Reading

- W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Advanced workflow patterns. In 7th International Conference on Cooperative Information Systems (CoopIS 2000), volume 1901 of Lecture Notes in Computer Science, pages 18–29. Springer-Verlag, Berlin, 2000.
- W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow Patterns. BETA Working Paper Series, WP 47, 2000.
- W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow Patterns. Distributed and Parallel Databases, 14:5–51, 2003.
- E. Evans. Domain-Driven Design - Tackling Complexity in the Heart of Software. Addison-Wesley, 2004.

Further Reading

- O. Vogel. Service abstraction layer. In Proceedings of EuroPlop 2001, Irsee, Germany, July 2001.
- U. Zdun. Some patterns of component and language integration. In Proceedings of EuroPlop 2004, Irsee, Germany, July 2004.
- F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. Pattern-oriented Software Architecture - A System of Patterns. J. Wiley and Sons Ltd., 1996.
- E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

Further Reading

- M. Kircher and P. Jain. Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource Management. J. Wiley and Sons Ltd., 2004.
- D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. Patterns for Concurrent and Distributed Objects. Pattern-Oriented Software Architecture. J.Wiley and Sons Ltd., 2000.
- J. Adams, S. Koushik, G. Vasuveda, and G. Calambos. Patterns for e-Business - A Strategy for Reuse. IBM Press, 2001.