Advanced Topics of Software Engineering (ASE)

Chapter 3. Software architectures and their trade-offs

Prof. Dr. Florian Matthes, Prof. Dr. Alexander Pretschner

Chair of Software Engineering for Business Information Systems (sebis)
Faculty of Informatics
Technische Universität München
wwwmatthes.in.tum.de

Software architectures and their trade-offs

Message-oriented middleware

- The class of middleware applications that support message-based interoperability is called **message-oriented middleware** (MOM).
- The distinction between client and service provider is blurred – at least from the perspective of the middleware:
  - To the MOM all parties look alike; i.e., they ***send and receive messages***,
  - The difference between clients and service providers is purely conceptual and can only be determined by humans who are aware of the semantics of the messages and of the message exchange.

["Web services - concepts, architectures and applications." Alonso, G.et al. (2004)]

Message construction (1)

- In MOM, all clients and service providers communicate by exchanging **messages**.
- A message is a structured data set, typically characterized by a **type** and a set of <name, value> pairs that constitute the message **parameters**.
- The language for defining message types varies with the messaging platform.

```
Message quoteRequest {
  QuoteReferenceNumber : Integer
  Customer : String
  Item : String
  Quantity : Integer
  RequestedDeliveryDate : Timestamp
  DeliveryAddress : String
}
```

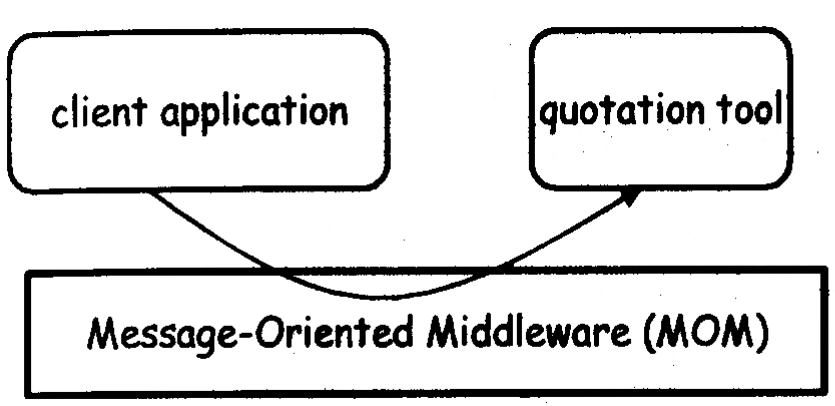
- Clients and service providers have to agree on a set of message types.

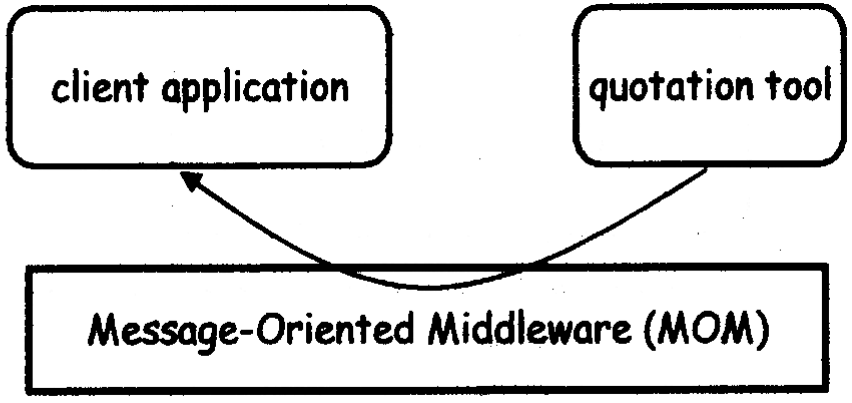
["Web services - concepts, architectures and applications." Alonso, G.et al. (2004)]

Message construction (2)

```
Message : quoteRequest {
 QuoteReferenceNumber: 325
 Customer: Acme,INC
 Item:#115 (Ball-point pen, blue)
 Quantity: 1200
 RequestedDeliveryDate: Mar 16,2003
 DeliveryAddress: Palo Alto, CA
}
```

```
Message: quote {
 QuoteReferenceNumber: 325
 ExpectedDeliveryDate: Mar 12, 2003
 Price:1200$
}
```

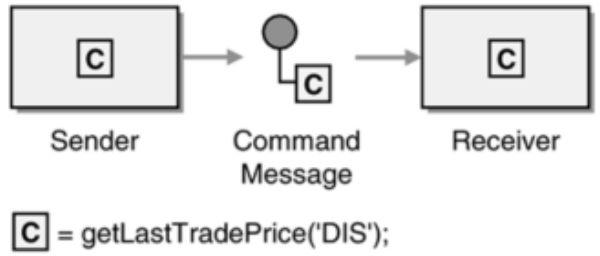




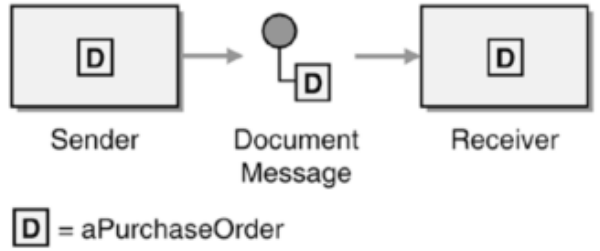
["Web services - concepts, architectures and applications." Alonso, G.et al. (2004)]

Message intent

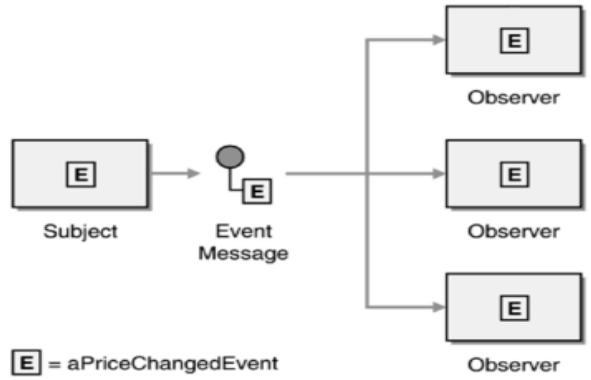
- **Command message**
  - The sender tells the receiver what code to run
  - Typically over point-to-point connection



- **Document message**
  - The sender transmits one of its data structures to the receiver
  - The sender is passing the data to the receiver but not specifying what the receiver should necessarily do with it



- **Event message**
  - Notifying the receiver of a change in the sender
  - The sender is not telling the receiver how to react, just providing a notification



["Enterprise integration patterns: Designing, building, and deploying messaging solutions." Hohpe, G. and Woolf, B. (2004)]

Message-oriented middleware - example

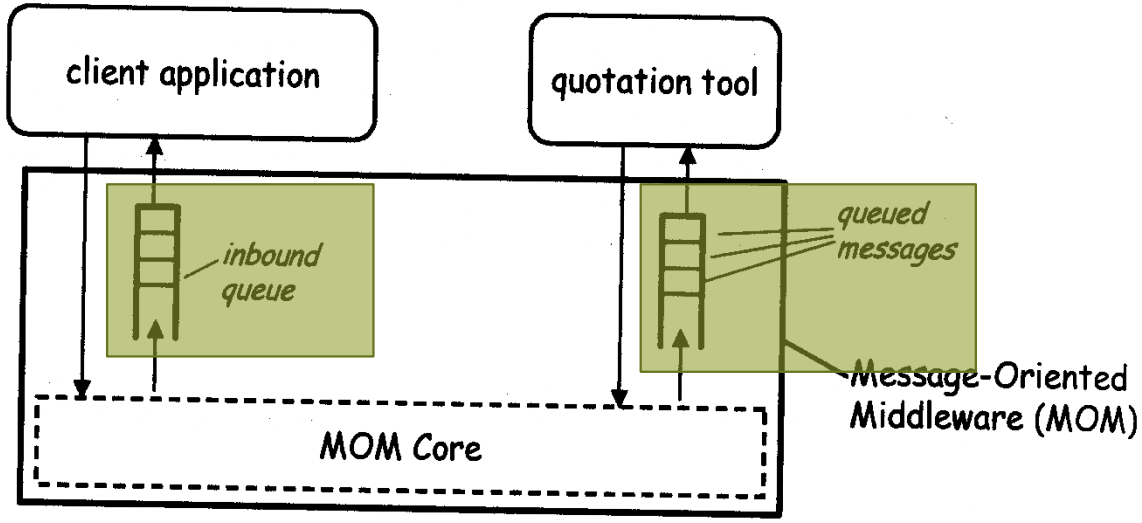
Java message service

- Java industry-standard API: Java Message Service (JMS)
- In JMS, a message is characterized by
  - A header, which includes metadata such as the message type, expiration date, priority, …
  - A body, which includes the actual application-specific information that needs to be exchanged.
- Addressing is performed through queues (discussed in the next slides):
  - Senders (receivers) first bind to a queue, i.e., identify the queue to (from) which they want to send messages (receive messages from), based on the queue name,
  - Then they can start sending (retrieving) messages to (from) the queue.
- JMS is simply an API and not a platform.
- JMS can be implemented as a stand-alone system or as a module within an application server.

["Web services - concepts, architectures and applications." Alonso, G.et al. (2004)]

Message queues

- One of the most important abstractions based on the message-oriented middleware (MOM) is that of **message queuing**.
- In a message queuing model, messages sent by MOM clients are placed into a queue, typically identified by a name, and possibly bound to a specific intended recipient.
- Whenever the recipient is ready to process a new message, it invokes the suitable MOM function to retrieve the first message in the queue.

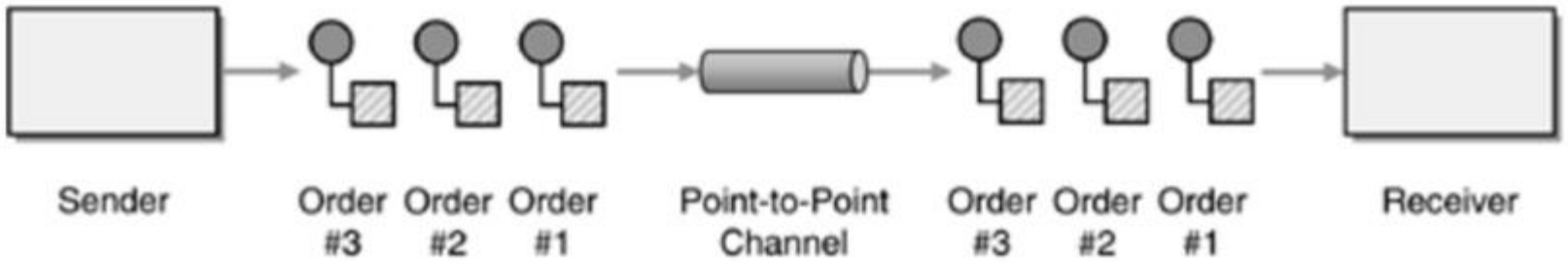


["Web services - concepts, architectures and applications." Alonso, G.et al. (2004)]

Message queues

Point-to-point pattern

- A point-to-point pattern ensures that only one receiver consumes any given message.




```
Queue queue = // obtain the queue via JNDI
QueueConnectionFactory f = // factory via JNDI
QueueConnection connection = f.createQueueConnection();
QueueSession session = connection.createQueueSession(true,
Session.AUTO_ACKNOWLEDGE);
QueueSender sender = session.createSender(queue);

Message message = session.createTextMessage("The contents of
the message.");

sender.send(message);
```

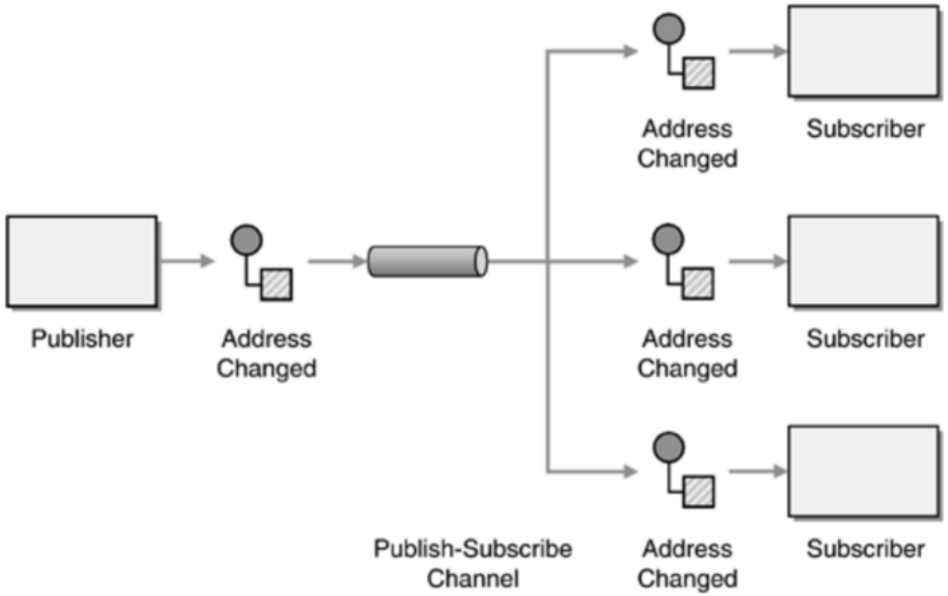
```
Queue queue = // obtain the queue via JNDI
QueueConnectionFactory f = // factory via JNDI
QueueConnection connection = f.createQueueConnection();
QueueSession session = connection.createQueueSession(true,
Session.AUTO_ACKNOWLEDGE);
QueueReceiver receiver = session.createReceiver(queue);

TextMessage message = (TextMessage) receiver.receive();
String contents = message.getText();
```

[“Enterprise integration patterns: Designing, building, and deploying messaging solutions.” Hohpe, G. and Woolf, B. (2004)]

Message queues

Publish-subscribe pattern

- Subscribers (data sinks) - subscribe to specific topics/events
- Publishers (data sources) – advertise and publish topics/events
- Middleware takes care of routing messages from source to sink



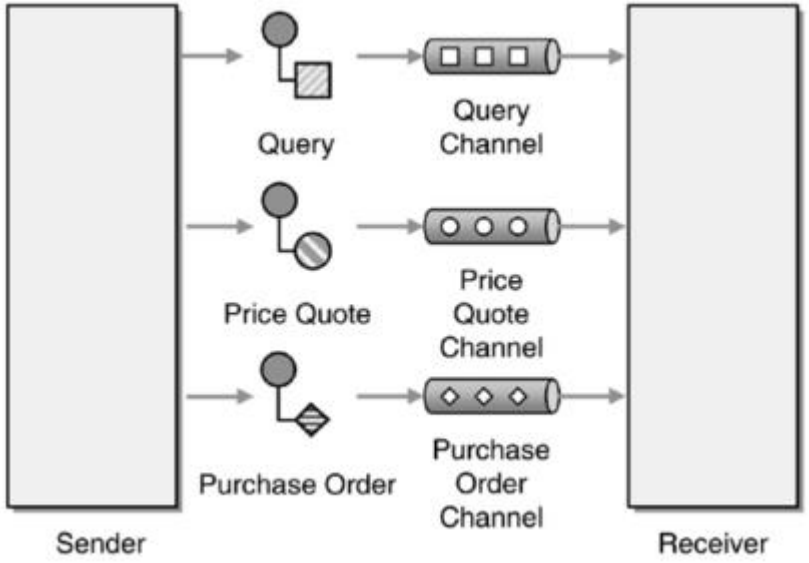
In JMS, use TopicPublisher and TopicSubscriber to send and receive messages

["Enterprise integration patterns: Designing, building, and deploying messaging solutions." Hohpe, G. and Woolf, B. (2004)]

Message queues

Datatype pattern

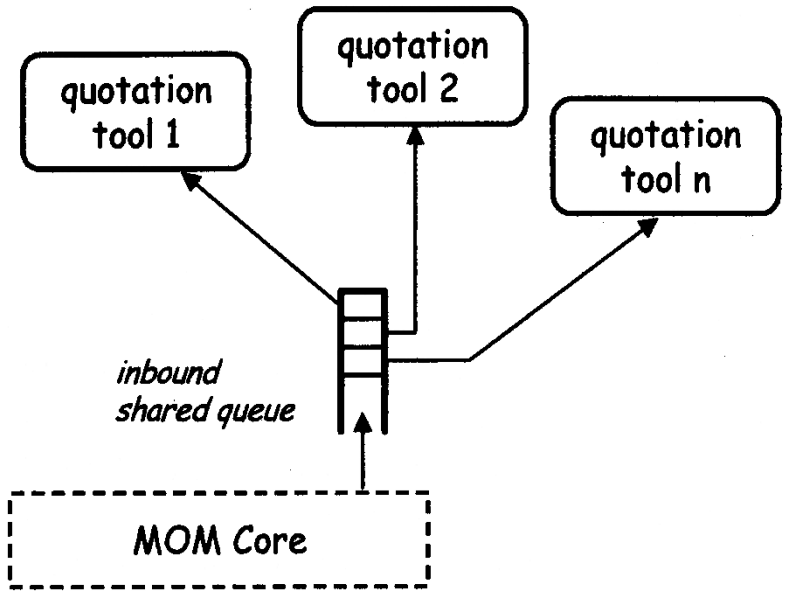
- Use separate datatype queues for each datatype



[“Enterprise integration patterns: Designing, building, and deploying messaging solutions.” Hohpe, G. and Woolf, B. (2004)]

Shared queues

- Shared queues can be used to distribute load among multiple applications that provide the same service.
- The MOM system controls access to the queue,
  ensuring that a message is delivered to **only one** application.



["Web services - concepts, architectures and applications." Alonso, G.et al. (2004)]

Benefits of message queues

- It gives recipients control of **when to process** messages.
- Recipients do not have to be continuously listening for messages and process them right away, but can instead retrieve a new message only when they can or need to process it.
- Queuing is **more robust to failures** with respect to RPC or object brokers, as recipients do not need to be up and running when the message is sent.
- If an application is down or unable to receive messages, these will be stored in the application's queue (maintained by the MOM).
- Queued messages may have an associated **expiration date** or interval.
- Queues can be **shared** among multiple applications.

Interacting with a message queuing system

- Queuing systems provide an API that can be invoked to send messages or to wait for and receive messages.
- Sending a message is typically a non-blocking operation.
- Receiving a message is instead often a blocking operation, where the receiver listens for messages and processes them as they arrive, typically by activating a new dedicated thread, while the main thread goes back to listen for the next message.
- Non-blocking alternative: provide a callback function that is invoked each time a message arrives.

[“Web services - concepts, architectures and applications.” Alonso, G.et al. (2004)]

Why / why not use messaging

- Remote communication
- Decoupling of systems
- Platform/application integration
- Asynchronous communication protocol (send-and-forget)
- Solves the issue of throttling (thanks to message queues)

- Scenarios where applications need synchronous solutions
- Message sequence issues (message channels do not guarantee when a message will be delivered)

[“Enterprise integration patterns: Designing, building, and deploying messaging solutions.” Hohpe, G. and Woolf, B. (2004)]

Software architectures and their trade-offs

From middleware to application integration

The use of middleware led to a further proliferation of services.

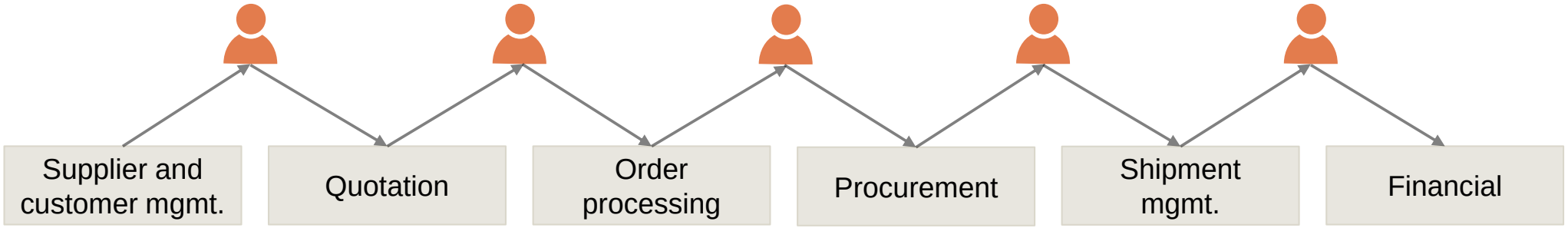
- Integration is now not only the integration of resource managers or servers, but also the integration of services.
- While for servers there has been a significant effort to standardize the interfaces of particular types of servers (e.g., databases), the same cannot be said of generic services.
- There was almost no infrastructure available that could help **integrate services provided by different middlewares**.
- Middleware was originally intended as a way to integrate servers that reside in the resource management layer.

→ Enterprise Application Integration (**EAI**) appeared in response to this.

- EAI includes as building blocks the application logic layers of different middleware systems.

["Web services - concepts, architectures and applications." Alonso, G.et al. (2004)]

Example of application integration

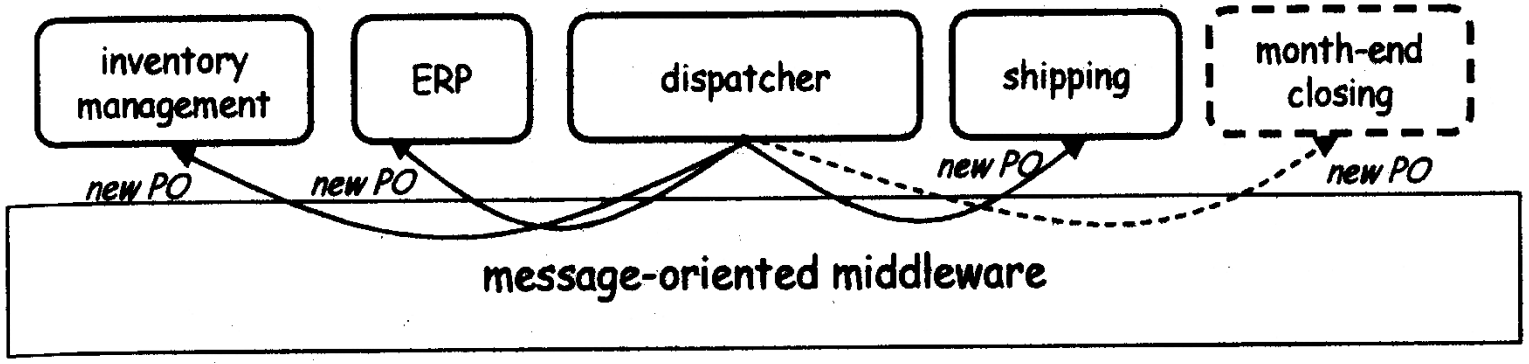


- Different operating systems
- Different interfaces (non-)/transactional, standard IDL/proprietary, …
- Different data formats
- Different security requirements (authentication)
- Different middleware

[“Web services - concepts, architectures and applications.” Alonso, G.et al. (2004)]

EAI middleware: message brokers (1)

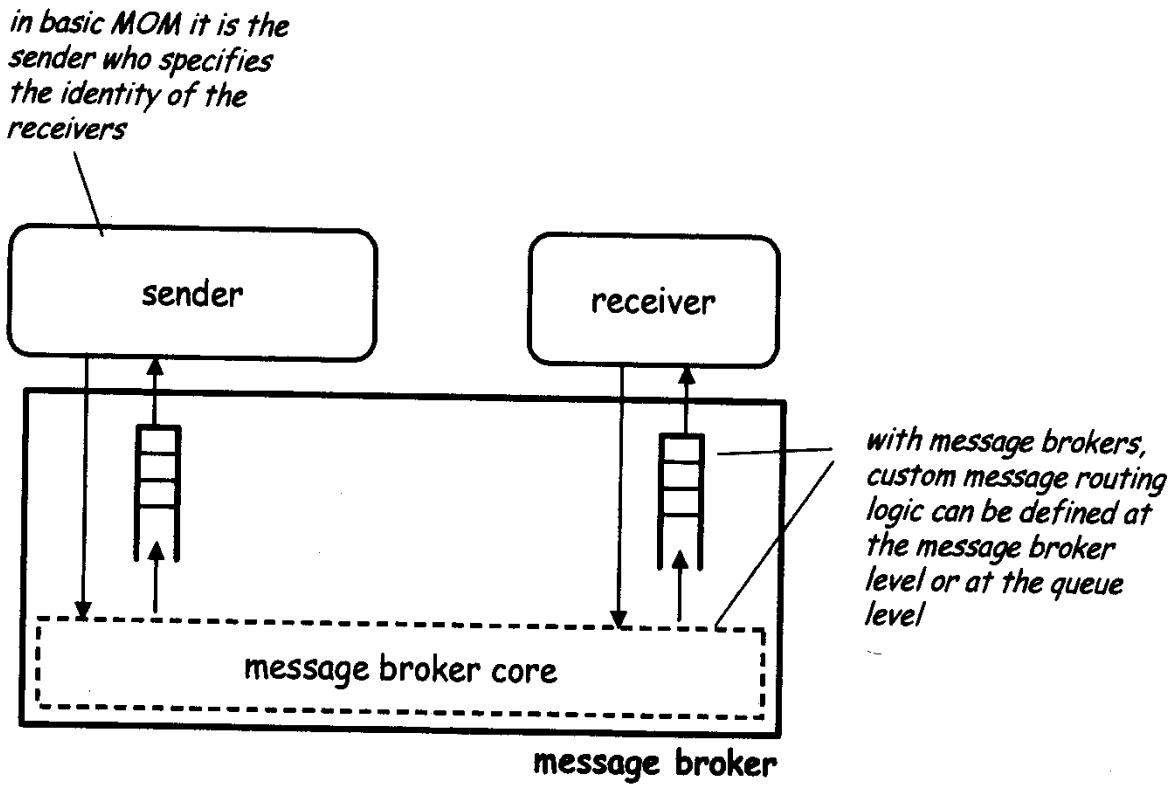
- Traditional RPC-based and MOM systems create **point-to-point** links between applications
  - They are rather static and inflexible with regard to the selection of queues to which messages are delivered.



- Message brokers **act as a broker** among system entities, thereby creating a (logical or physical) "**hub and spoke**" communication infrastructure for integrating applications.

["Web services - concepts, architectures and applications." Alonso, G.et al. (2004)]

EAI middleware: message brokers (2)



["Web services - concepts, architectures and applications." Alonso, G.et al. (2004)]

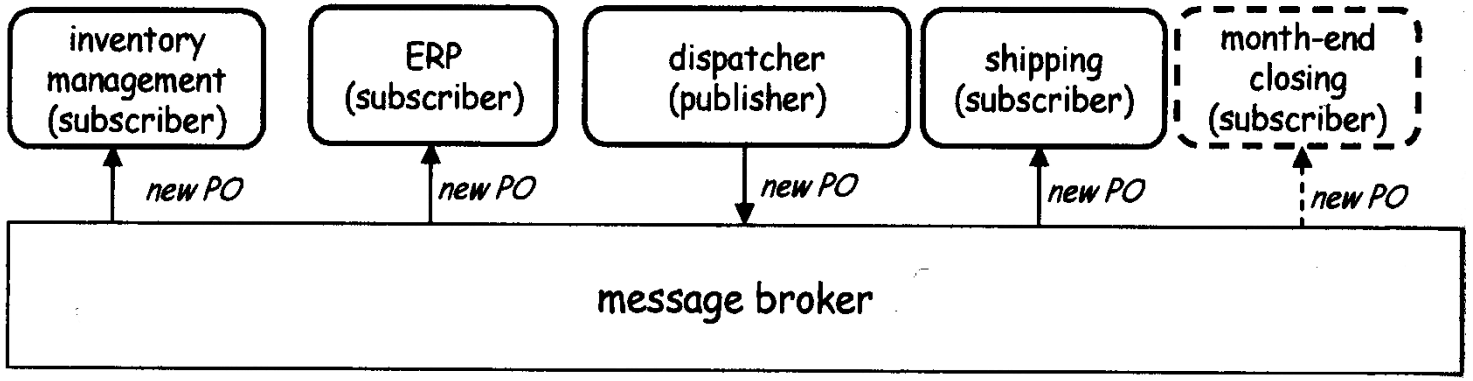
Extending basic MOM

- Message brokers factor the message routing logic out of the senders and place it into the middleware.
- There is now a **single place** where we need to make changes when the routing logic for messages needs to be modified.
- **Routing logic**
  - Can be based on the sender's identity, on the message type, or on the message content,
  - Is typically defined in a rule-based language.
- Message brokers decouple senders and receivers.
- Since message brokers require the communication to go through a middle layer, even more application-specific functionality can be implemented there, e.g., *content transformation* rules.

["Web services - concepts, architectures and applications." Alonso, G.et al. (2004)]

The publish/subscribe interaction model

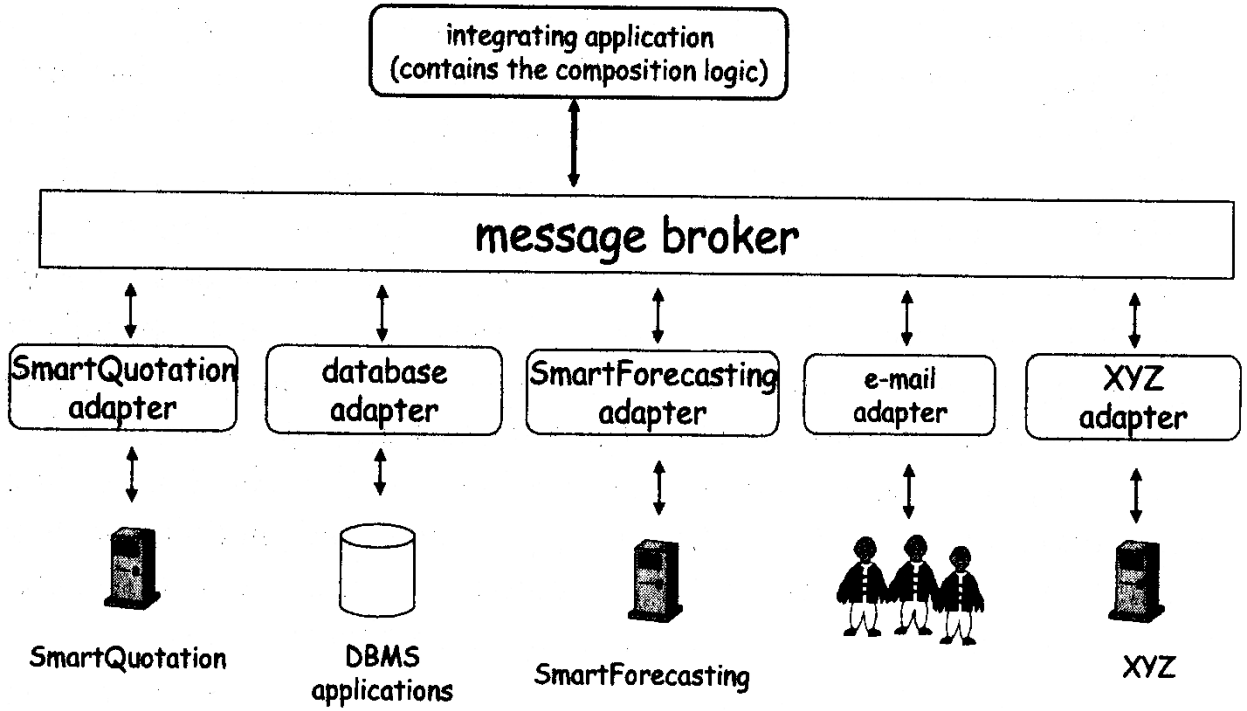
- Applications simply **publish** the message to the middleware system (**publishers**)
- If an application is interested in receiving messages of a given type, then it must **subscribe** with the publish/subscribe middleware (**subscribers**)
- Whenever a publisher sends a message of a given type, the middleware retrieves the list of all applications that subscribed to messages of that type, and delivers a copy of the message to each of them.



["Web services - concepts, architectures and applications." Alonso, G.et al. (2004)]

EAI with a message broker

- Adapters map heterogeneous data formats, interfaces, and protocols into a common model and format.
- A different adapter is needed for each type of application that needs to be integrated.



["Web services - concepts, architectures and applications." Alonso, G.et al. (2004)]

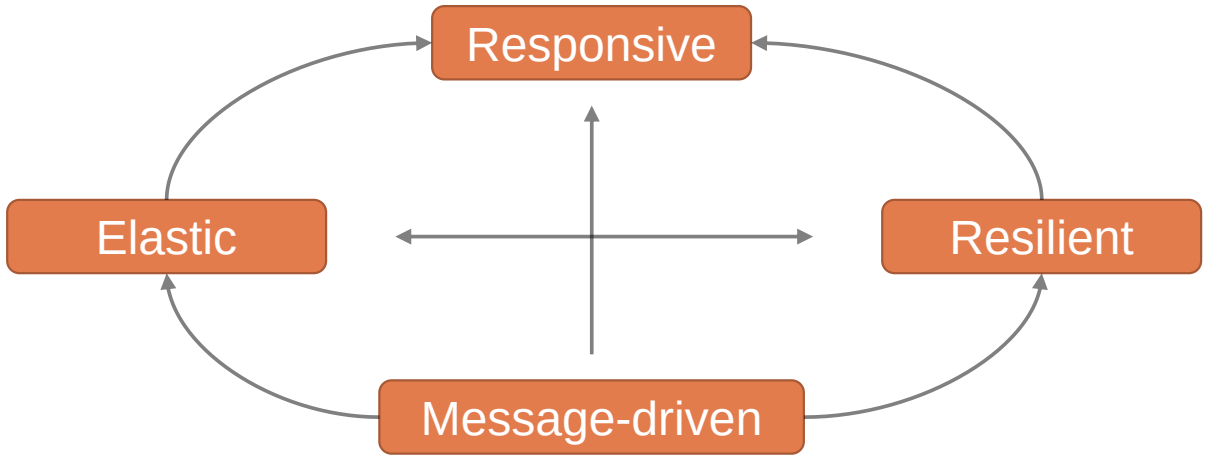
Challenges - message brokers

- **Scalability** - with thousands of clients, connections, and streams of messages
- **Security** - as the number of SSL certificates (for client authentication) increase, managing them becomes a challenge
- **Monitoring** and **debugging** within a broker becomes expensive over time
- **Maintenance** of message brokers – message broker cannot be updated without affecting the clients

Reactive systems

Reactive applications employ an architecture that allows you to build systems that are **responsive**, **resilient**, **elastic**, and **message-driven** and that are capable of producing a **real-time** feel.

- React to users and other application components
- React to failure
- React to load
- React to messages



["The reactive manifesto v2.0" (2014)]

Reactive systems

- **Responsive**
  - Focus on providing rapid and consistent response times, establishing reliable upper bounds so reactive systems deliver a consistent quality of service
- **Resilient**
  - The system stays responsive in the face of failure (replication, containment, isolation, delegation)
- **Elastic** (scale on demand)
  - The system stays responsive under varying workload
- **Message-driven**
  - Asynchronous message-passing between components
  - Ensures loose coupling, isolation, and location transparency
  - Message categories
    - Command message
    - Document message
    - Event message

["The reactive manifesto v2.0" (2014)]

Actor model

A computational model that embodies:

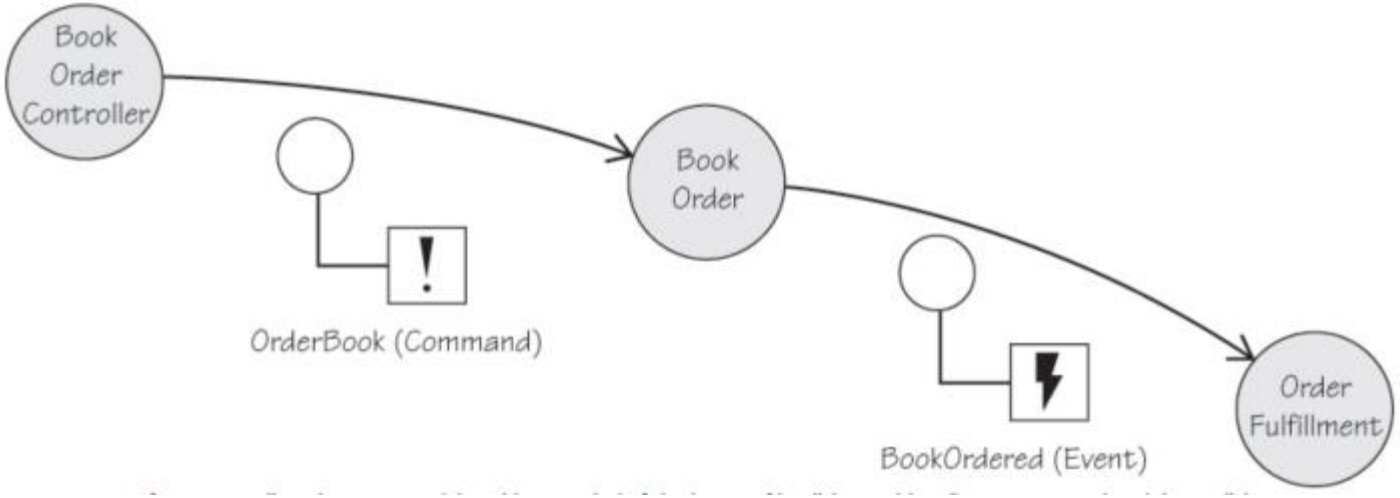
- Processing
- Storage
- Communication

Supports three axioms - when an actor receives a message it can:

- Make local decisions
- Create new actors
- Send messages to actors it knows
- Designate how it should handle the next message it receives

The actor model represents objects and their interactions, resembling human organizations and built upon the laws of physics.

[“Principles of reactive programming.” Kuhn R.]



What is an Actor?

- An actor
  - is an object with identity
  - has a behavior
  - only interacts using *asynchronous* message passing

Remember, no shared memory! Exchange of data only via *asynchronous* messages.

["A universal modular ACTOR formalism for artificial intelligence." Hewitt C., Bishop P., Steiger R. (1973)]

Actor library toolkit

- Actor model can be implemented directly in a programming language (ActorScript, Smalltalk, ..) or
  - Use an actor library toolkit –
    - Akka (with Scala/Java)
    - Dostero (.NET Actor model)

- Examples in the next slides - Scala with Akka

The actor trait

```
trait Actor extends AnyRef {
  def receive: Receive
  def sender: ActorRef
  implicit val self: ActorRef
  implicit val context: ActorContext
  ...
}
```

```
type Receive = PartialFunction[Any, Unit]
```

- An actor has a well-defined (non-cyclic) life-cycle.
  - *RUNNING* (created and started actor) – can receive messages
  - *SHUTDOWN* (when “stop” is invoked) – can’t do anything

- The Actor’s own ActorRef is available as **self**, the current message’s sender as **sender()** and the ActorContext as **context**. The only abstract method is **receive** which shall return the initial behavior of the actor as a partial function.
- The behavior can be changed using *context.become* and *context.unbecome* (see ActorContext)

[Akka documentation]

A simple actor example

Implementing a reactive counter in three steps

```
class Counter extends Actor {
  var count = 0
  def receive = {
    case “incr” => count +=1
  }
}
```

make it stateful

```
class Counter extends Actor {
  var count = 0
  def receive = {
    case “incr” => count +=1
    case (“get”, customer: ActorRef) => customer ! count
  }
}
```

! sends a message asynchronously

use the implicit sender

```
class Counter extends Actor {
  var count = 0
  def receive = {
    case “incr” => count +=1
    case “get” => sender ! count
  }
}
```

Actors can send messages to addresses (ActorRef) they know.
Remember, actor knows about the sender implicitly!

change actor’s behavior
use *become* from
the ActorContext

```
class Counter extends Actor {
  def counter(n: Int): Receive = {
    case “incr” => context.become(counter(n+1))
    case “get” => sender ! n
  }
  def receive = counter(0)
}
```

State change is explicit
State is scoped to the current behavior

[“Principles of reactive programming.” Kuhn R.]

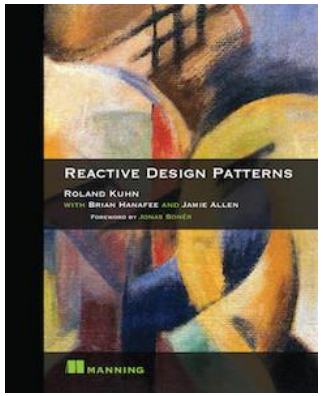
Creating and stopping actors

Actors are created by actors.

Create the actor

Name the actor

```
class Main extends Actor {
  var counter = context.actorOf(Props[Counter], “myCounter”)

  counter ! “incr”
  counter ! “incr”
  counter ! “get”

  def receive = {
    case count: Int =>
      println(s“count was $count”)
      context.stop(self)  // Stops after receiving the message from Counter actor
  }
}
```



Recommended online course: “Principles of Reactive Programming”
https://www.reactivedesignpatterns.com/

Reactive systems

Actor model – advantages and disadvantages

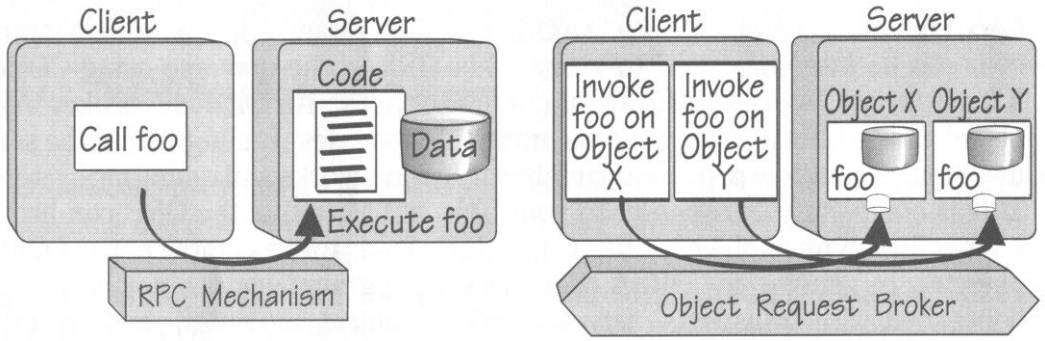
- Message-driven approach is more intuitive
- Directly and transparently applicable for distributed programming
- Type-safe systems
- Separation of concerns – clear distribution of responsibilities for actors
- Ensures loose coupling, isolation, and location transparency

- Message-passing is slower
- Actors do not work well when synchronous shared-state behavior is required
- Steep learning curve (Scala with Akka)
- Dealing when an actor crashes, a message is lost, or an external message is introduced

Software architectures and their trade-offs

Object brokers

- Object broker extends the **Remote Procedure Call** (RPC) paradigm to the object-oriented world; they provide services that simplify the development of **distributed object-oriented** applications.
- Difference to simple RPC: clients invoke a **method of an object**
  - Because of **inheritance** and **polymorphism** the function performed by the server object actually depends on the class to which the server object belongs.
  - The middleware has to bind clients with specific objects running on a server and manage the interactions between two objects.

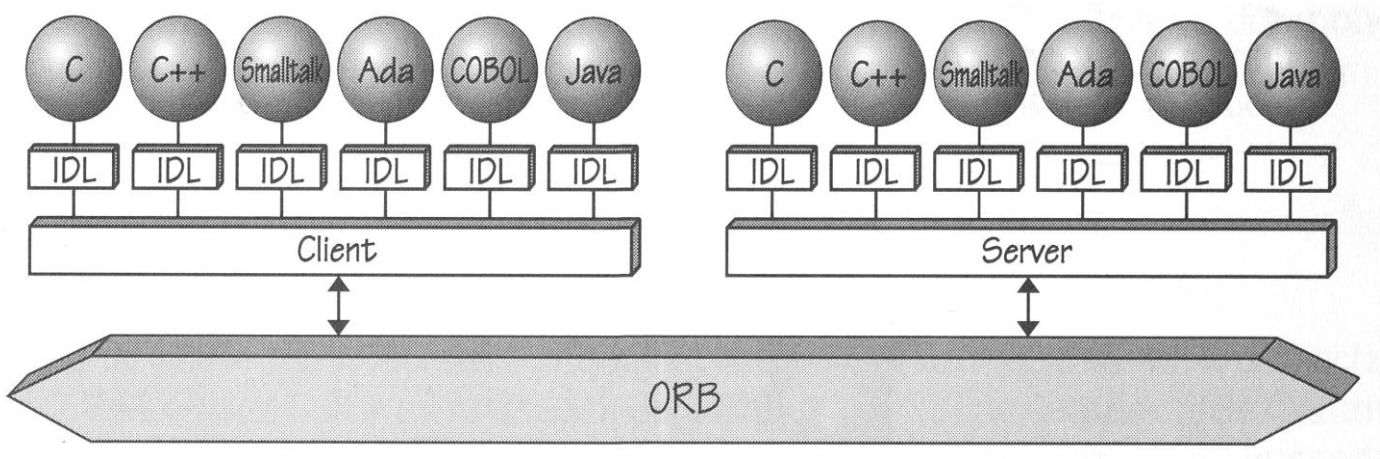


- With time, object brokers added features that went beyond basic interoperability, for example including location transparency, object lifecycle management, and persistence.

["Web services - concepts, architectures and applications." Alonso, G.et al. (2004)]

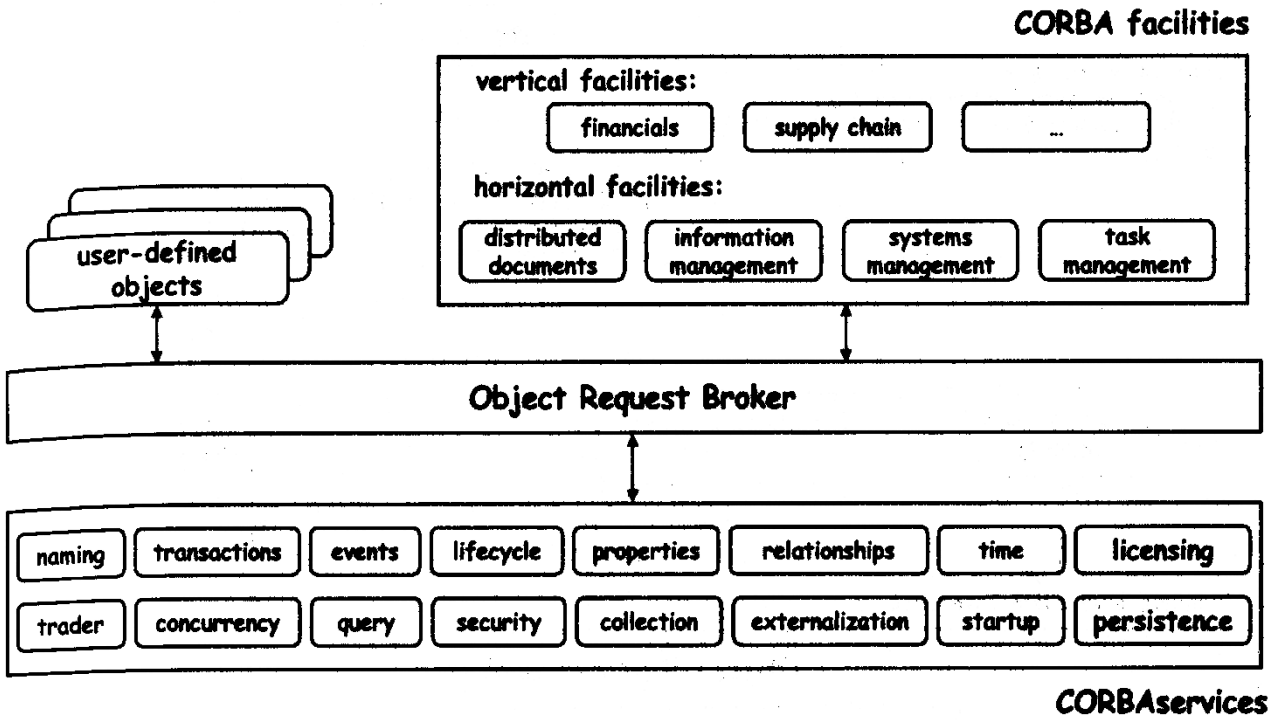
CORBA

- The best known example of object broker is the abstraction described in the **Common Object Request Broker Architecture** (CORBA) specification, which was developed in the early 1990s by the Object Management Group (OMG).
- It offers a **standardized specification** of an object broker rather than a concrete implementation.
- CORBA is agnostic with respect to both the programming language used to develop object-oriented applications and also the operating systems on which the applications run.



["Web services - concepts, architectures and applications." Alonso, G.et al. (2004)]

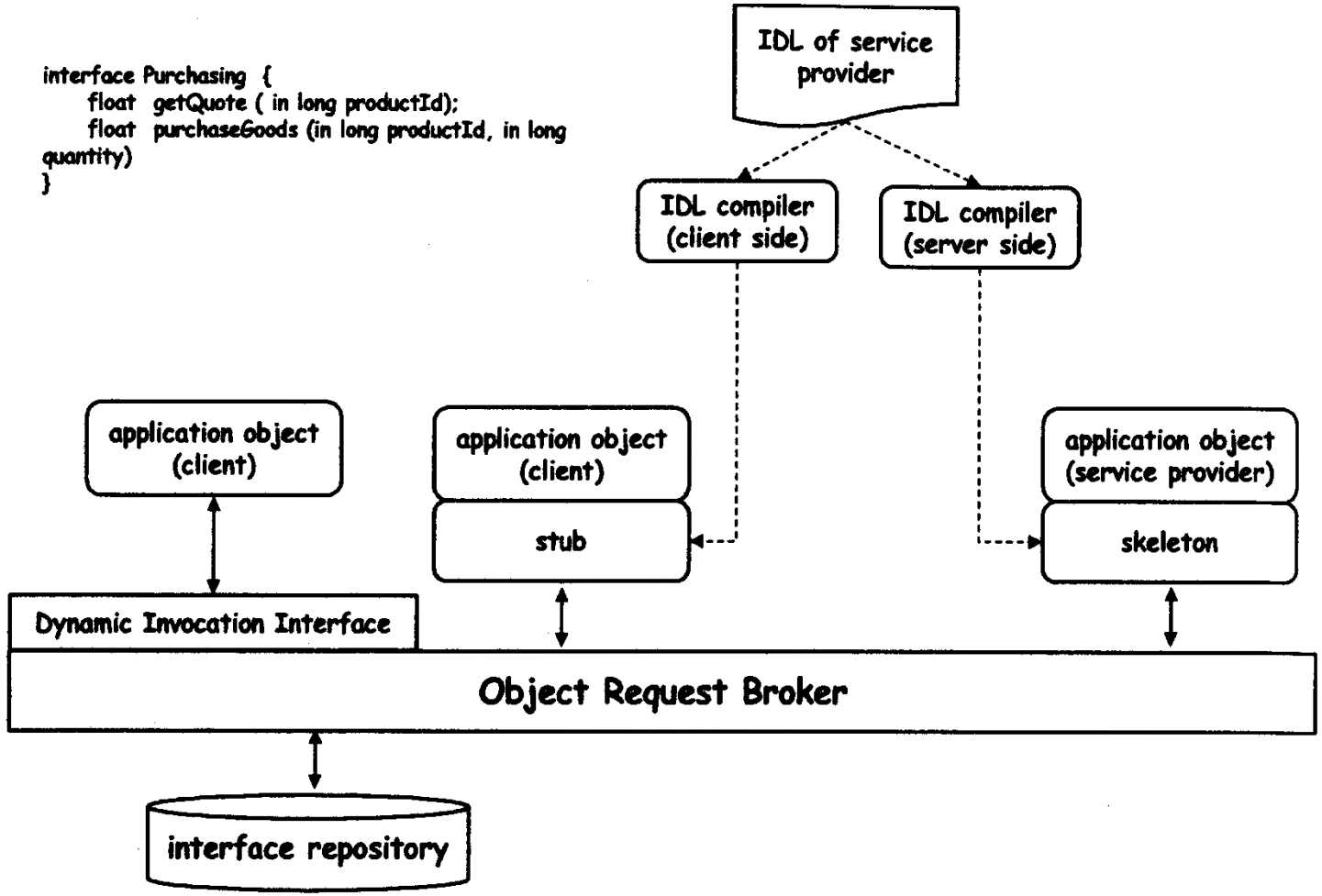
CORBA architecture



["Web services - concepts, architectures and applications." Alonso, G.et al. (2004)]

Dynamic service selection and invocation

```
interface Purchasing  {
     float  getQuote ( in long productId);
     float  purchaseGoods (in long productId, in long
quantity)
}
```



["Web services - concepts, architectures and applications." Alonso, G.et al. (2004)]

Summary

CORBA was needed because no technology addressed all of the following requirements:

- Bit protocol
- Open interfaces
- A single governance
- Object-orientation
- Language independence
- Operating system independence
- Freedom from technologies
- Data-typing
- High tunability
- Freedom from data-transfer details
- Compression

CORBA in practice had a set of drawbacks that ultimately led to low adoption for business applications:

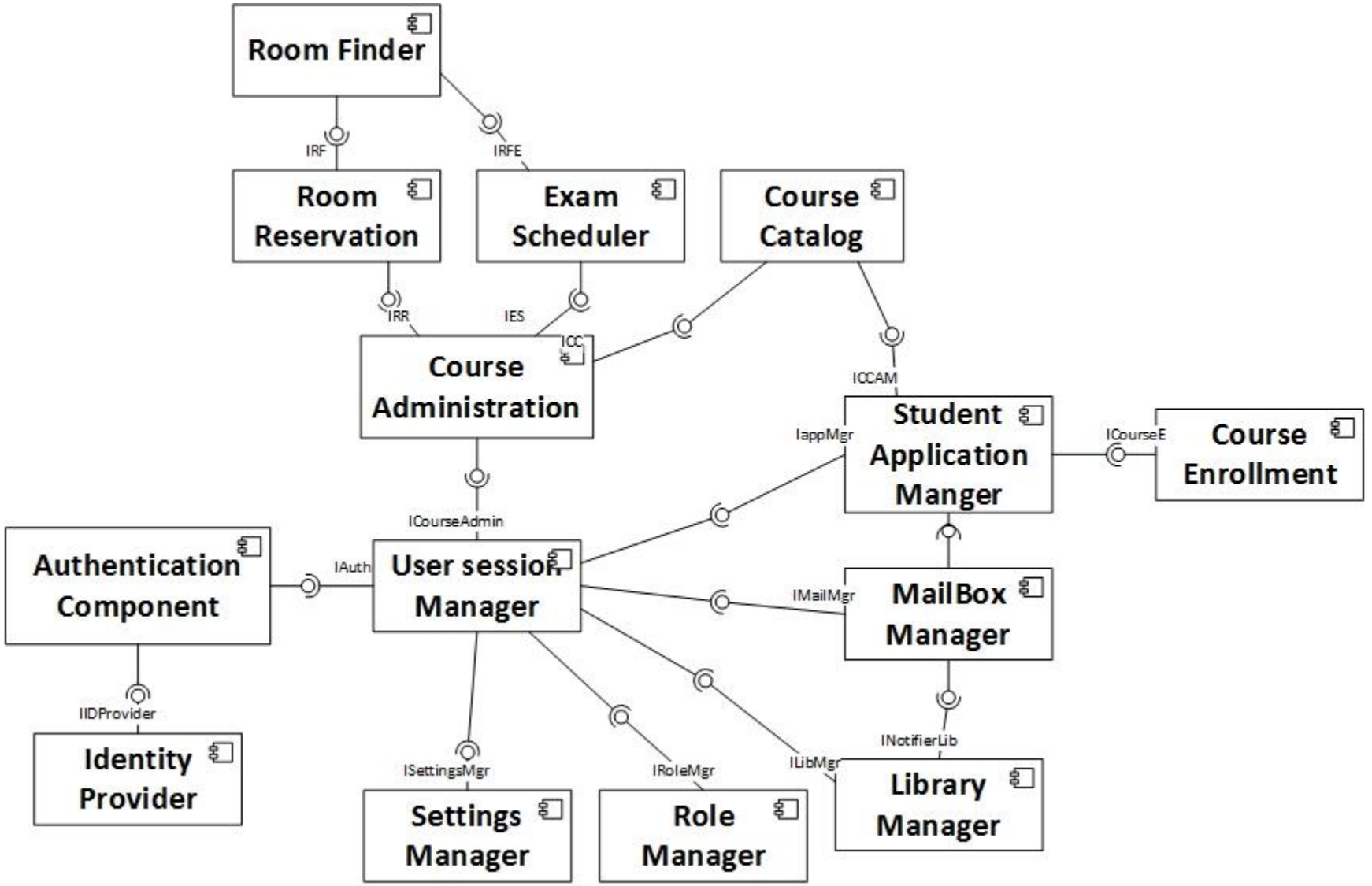
- No standard tools
- Hard to deploy
- Heavy weight
- Complex toolchain
- Initial implementation incompatibilities
- Location transparency
- Design and process deficiencies
- Problems with implementations
- Firewall incompatibilities

[Common Object Request Broker]

Software architectures and their trade-offs

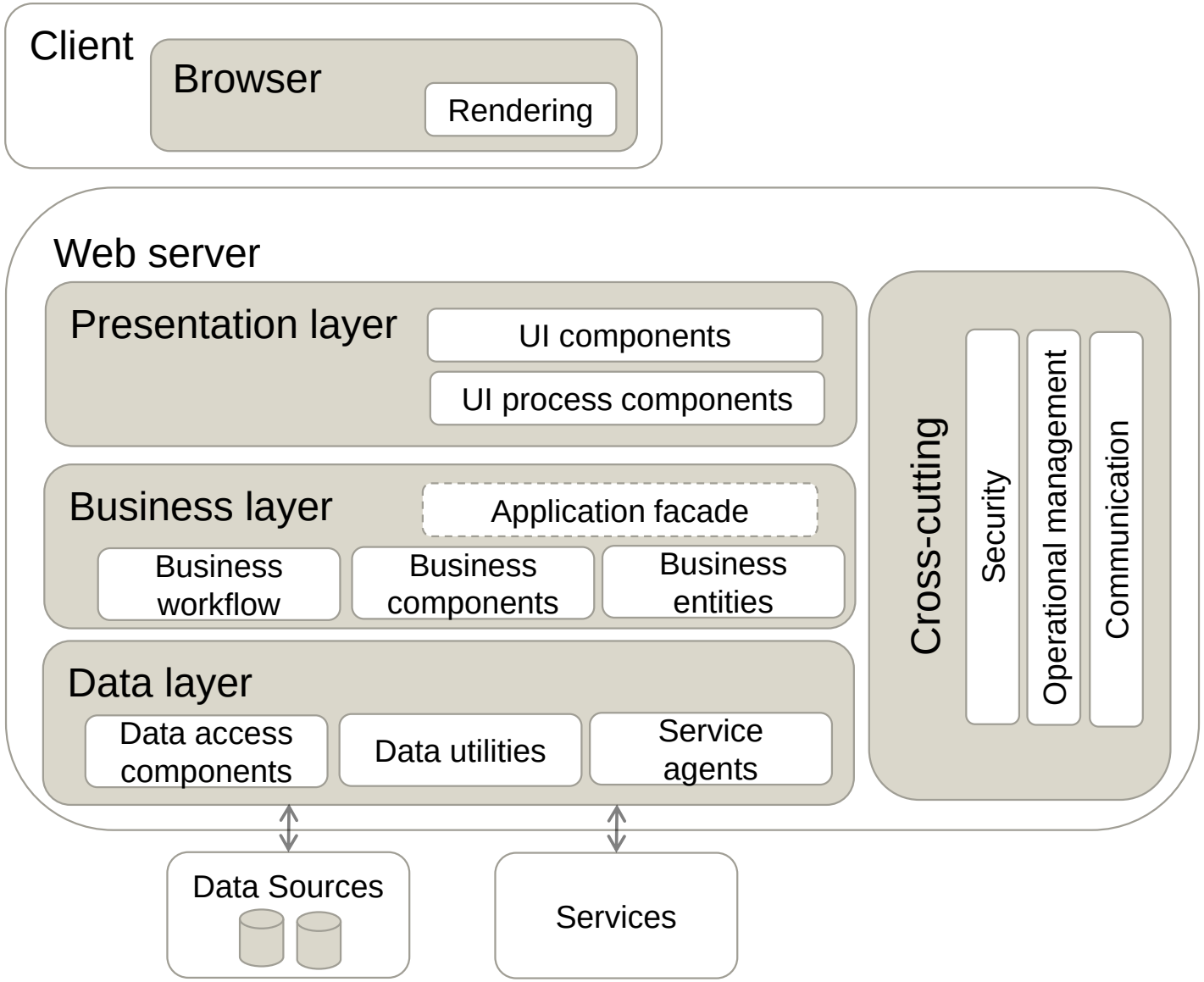
Component-based architectures

Motivation - TUMOnline



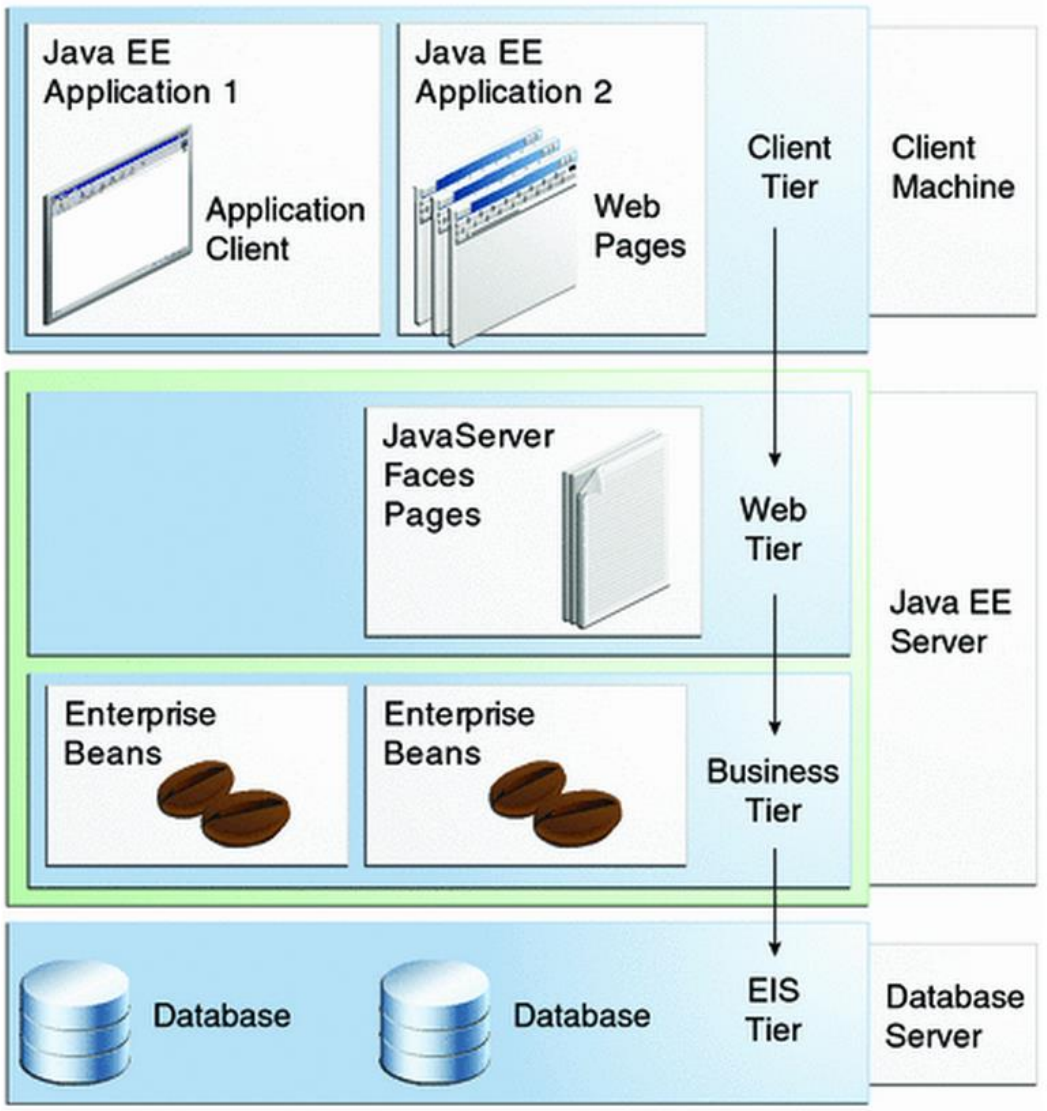
Software architectures and their trade-offs

A typical three-layered web architecture



We have discussed this from the database-centric architecture viewpoint

Multi-tier architecture (1)



[The Java EE 6 Tutorial]

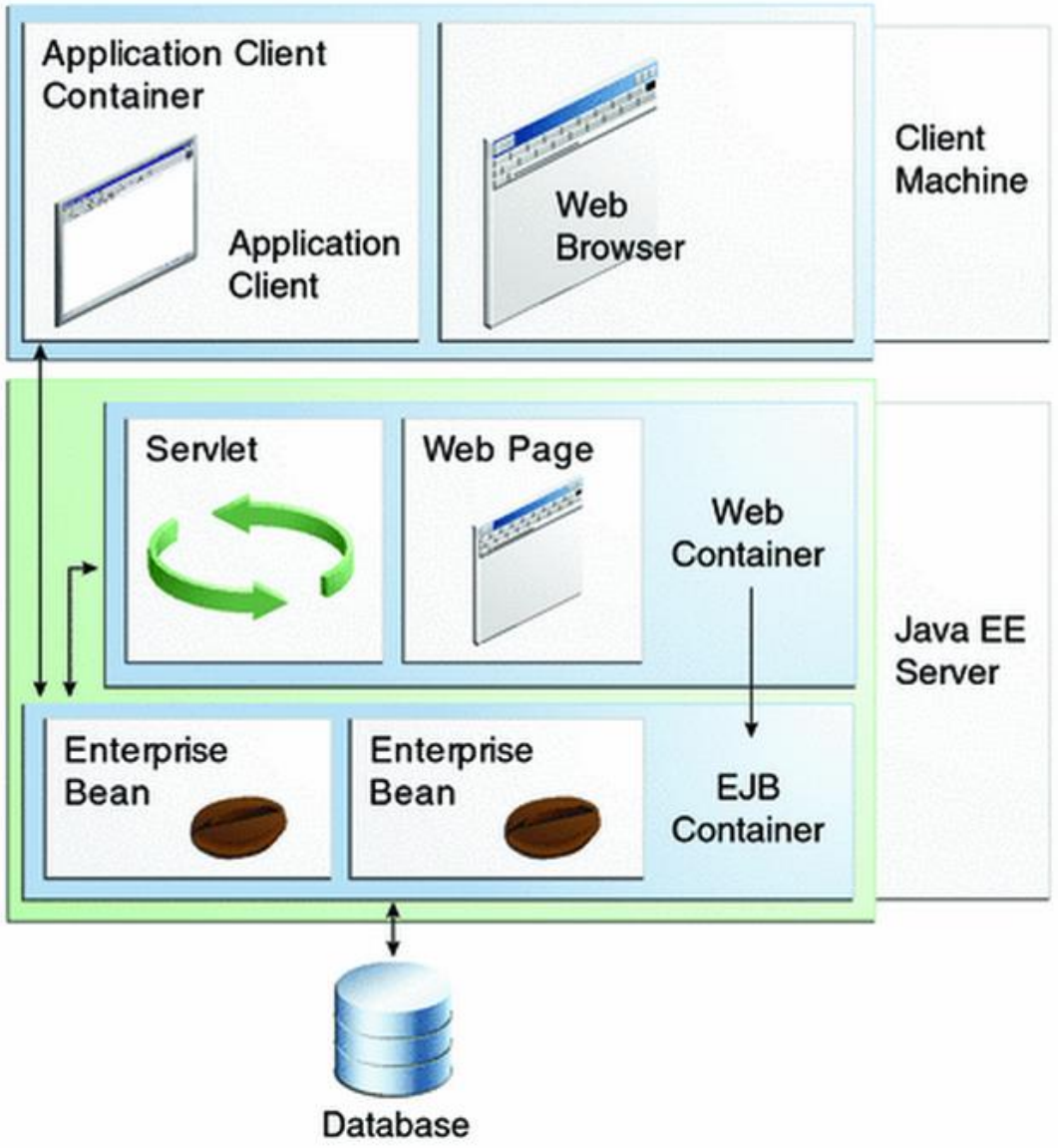
Multi-tier architecture (2)

- Client tier
  - (Remote) application clients that access JEE server
- Web tier
  - Manages interaction between client and business tier (content generation, session management, …)
- Business tier
  - Realizes business logic / business processes functionality
- Enterprise Information System tier
  - Consists of databases, ERP systems, data warehouses, … Usually not co-located with JEE server

Application server

- Mediates access to system resources
- Life-cycle and deployment management
- Communication and security management
- Transaction and persistence management
- Functionality encapsulated into containers
  - Containers may maintain multiple components
    - Application Client container
      Manages application client components
    - EJB container
      Manages Enterprise Java Bean components
    - Web container
      Manages JSP and servlet components

Containers



Enterprise Java Beans (EJB)

- Enterprise Java Beans realize business logic;
  deployed within EJB containers → RMI for interaction

- Three different type of beans
  - **Session beans**
    - Transient, only "live" for one session; one bean per client
      - Stateless: state only for one method invocation
      - Stateful: state maintained between method invocations
      - Singleton: one instance per server, shared among clients

  - **Message-driven beans**
    Asynchronous processing of messages (JMS message listener)
    no direct RMI invocation → only indirectly via messages

  - Entity beans (deprecated since J2EE 1.4)

Contents of an enterprise bean

- **Enterprise bean class**: Implements the business methods of the enterprise bean and any lifecycle callback methods.

- **Business interfaces**: Define the business methods implemented by the enterprise bean class. A *business interface* is not required if the enterprise bean exposes a *local, no-interface* view.

- **Helper classes**: Other classes needed by the enterprise bean class, such as exception and utility classes.

Accessing enterprise beans

- Clients access enterprise beans either through:
  - *no-interface* view: of an enterprise bean exposes the public methods of the enterprise bean implementation class to (local) clients.
  - *business interface:* is a standard Java interface that contains the business methods of the enterprise bean.
    - *@Local* or *@Remote*

- The client of an enterprise bean obtains a reference to an instance of an enterprise bean through
  - **Dependency injection**: using Java annotations (*@EJB*)
  - **JNDI lookup**: using the Java Naming and Directory Interface syntax
    - For remote enterprise beans [*java:global*]
    - For local enterprise beans [*java:module, java:app*]

Enterprise bean – stateless session bean example

```
@Remote
public interface RoomFinderRemote {
   public Room findRoom(String searchTerm, Organization org, Building building);
}

@Stateless
public class RoomFinderBean implements RoomFinderRemote {
   ….
   public Room findRoom(String searchTerm, Organization org, Building building) {
      Room room;
      // some magic happens here..
      return room;
   }
}

public class RoomFinderClient {
      @EJB
      static RoomFinderSession roomFinderSession;

     public static void main(String[] args) throws Exception {
      InitialContext ctx = new InitialContext();
      // RoomFinderSession roomFinderSession = (RoomFinderSession)
      //              ctx.lookup(RoomFinderSession.class.getName());
      Organization org = new Organization("TUINI19 Informatics 19");
      Building building = new Building("5601 FMI");
      Room room = roomFinderSession.findRoom("01.12.059", org, building);
      System.out.println(room.getCoordinates());
   }
}
```

Remote business interface

Stateless session bean

Client for the session bean

Java Persistence API - JPA

- Replaces concept of Entity Beans as persistence components prior to J2EE 1.4

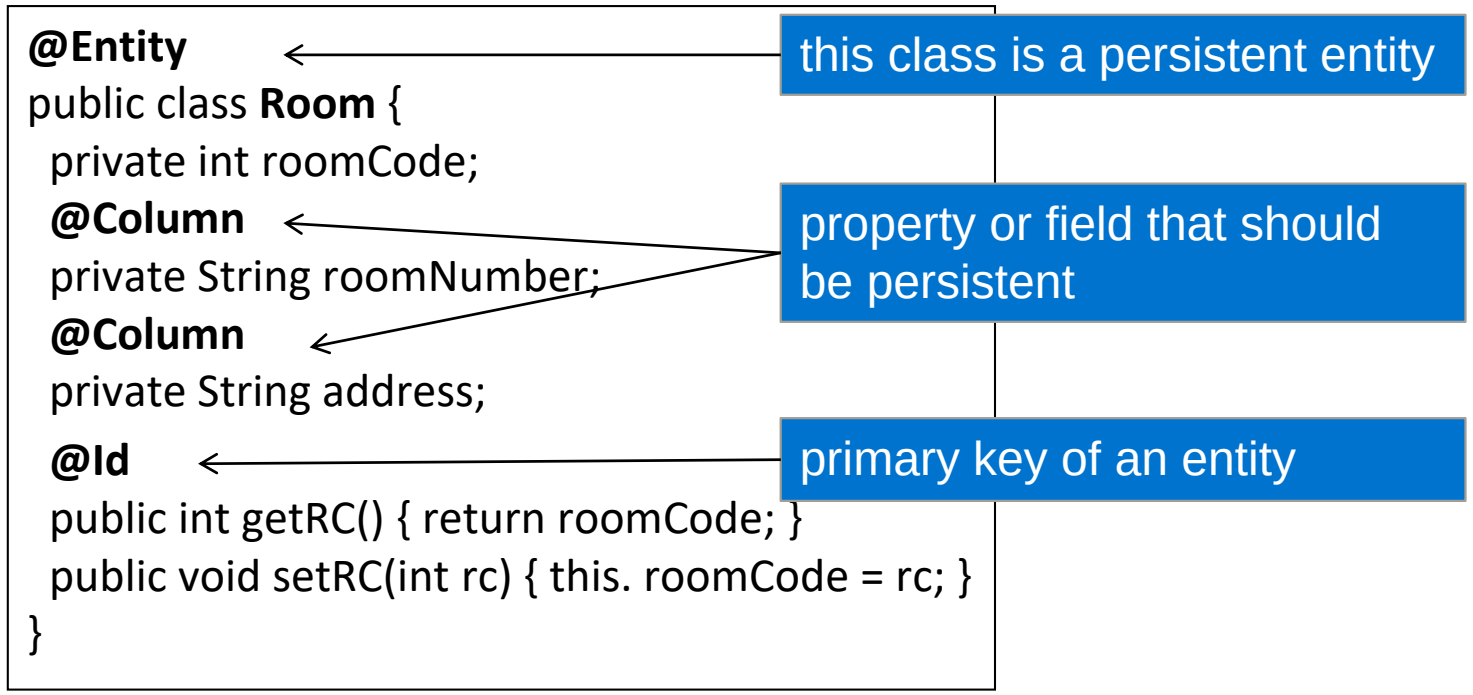
- Allows persistent deployment of POJOs to relational databases
  → Realizes **object-relational** mapping

- Mappings
  - Classes → Tables
  - Objects → Rows
  - Attributes → Columns

- Annotations mark persistent fields and properties

- EntityManager API to create and remove persistent mappings / query over entities

JPA example


@Entity
public class Room {
private int roomCode;
@Column
private String roomNumber;
@Column
private String address;
@Id
public int getRC() { return roomCode; }
public void setRC(int rc) { this. roomCode = rc; }
}
this class is a persistent entity
property or field that should be persistent
primary key of an entity

Java Messaging Service (JMS)

- Message Oriented Middleware for Java

- Supports loosely coupled, reliable, asynchronous, and message-based communication

- Two service types
  - **Point-to-Point** → communicating-processes style
    messages routed between one sender and one receiver;
    receiver maintains message queue → guaranteed delivery
  - **Publish-and-Subscribe** → publish and subscribe style
    sender publishes message with specific topic;
    multiple receivers may receive them if previously subscribed for topic
    → no guaranteed delivery

We discussed this in message-oriented architectures too.

Message Structure

- **Header**
  - Standard attributes
  - Set by JMS provider / message producer
  - E.g. message-id, delivery mode (persistent, non-persistent), destination, priority, redelivery, timestamp

- **Properties** (optional)
  - Optional attributes
  - Application- / vendor-specific

- **Body**
  - Actual payload
  - No specific format

Realize web applications: “view” part of JEE

Java Servlets

- Java classes that automatically respond to client requests
- Process HTTP requests and generate replies

JavaServer Pages

- Text-based documents (mostly html) with special JSP tags that contain java code
- JSP engine parses them: dynamically creates Java servlet classes
- Deprecated

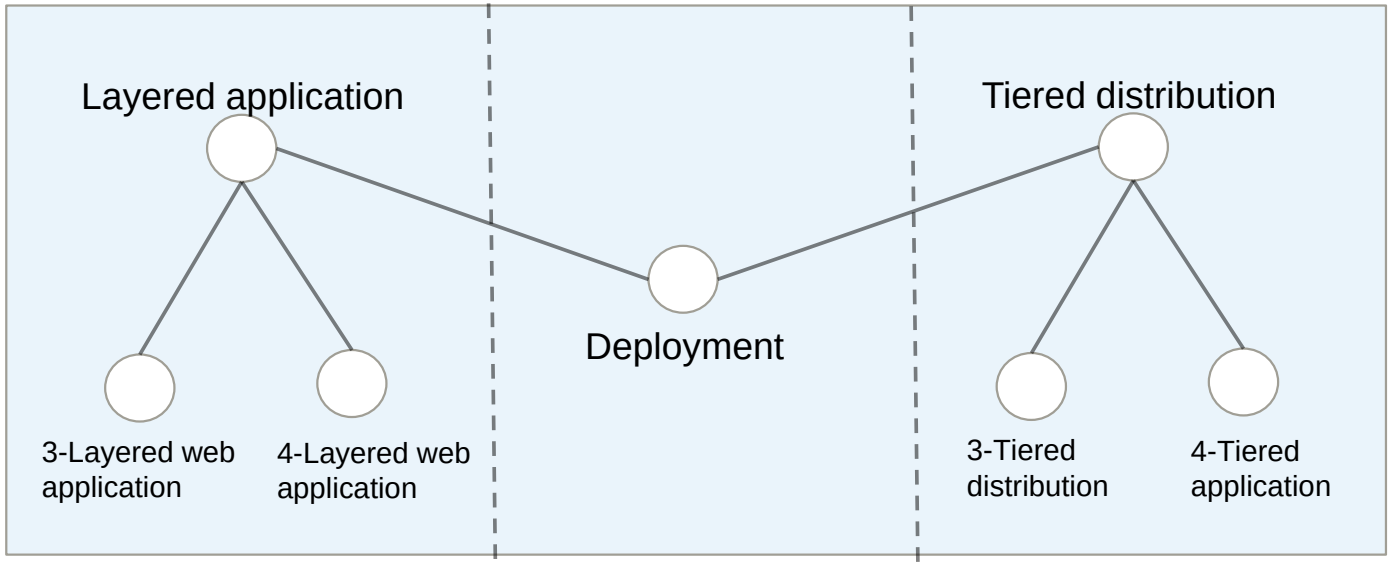
JavaServer Faces

- MVC pattern based web framework
- Uses templates (“facelets”) for views (but also support for XUL etc.)

Bringing applications and infrastructure together

Layer – is a logical structuring for the elements that make up your software solution.

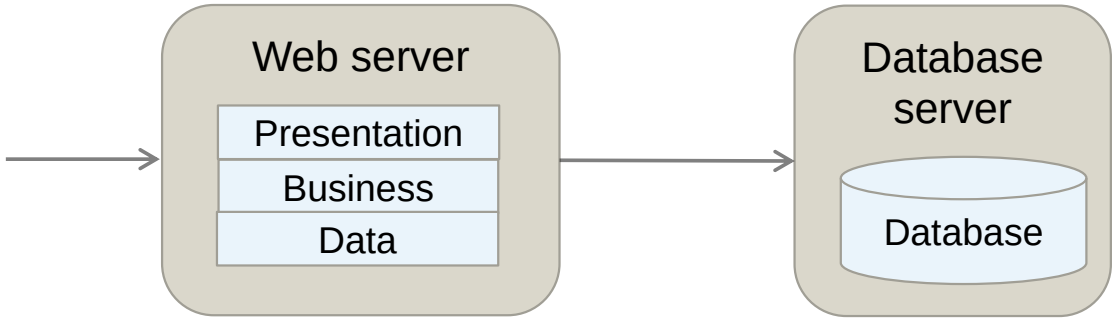
Tier – is a physical structuring mechanism for the system infrastructure.



["Microsoft application architecture guide."J.D. Meier et al. (2009)]

Deployment strategy

- **Non-distributed deployment**
  - all of the functionality and layers reside on a single server except for data storage functionality.



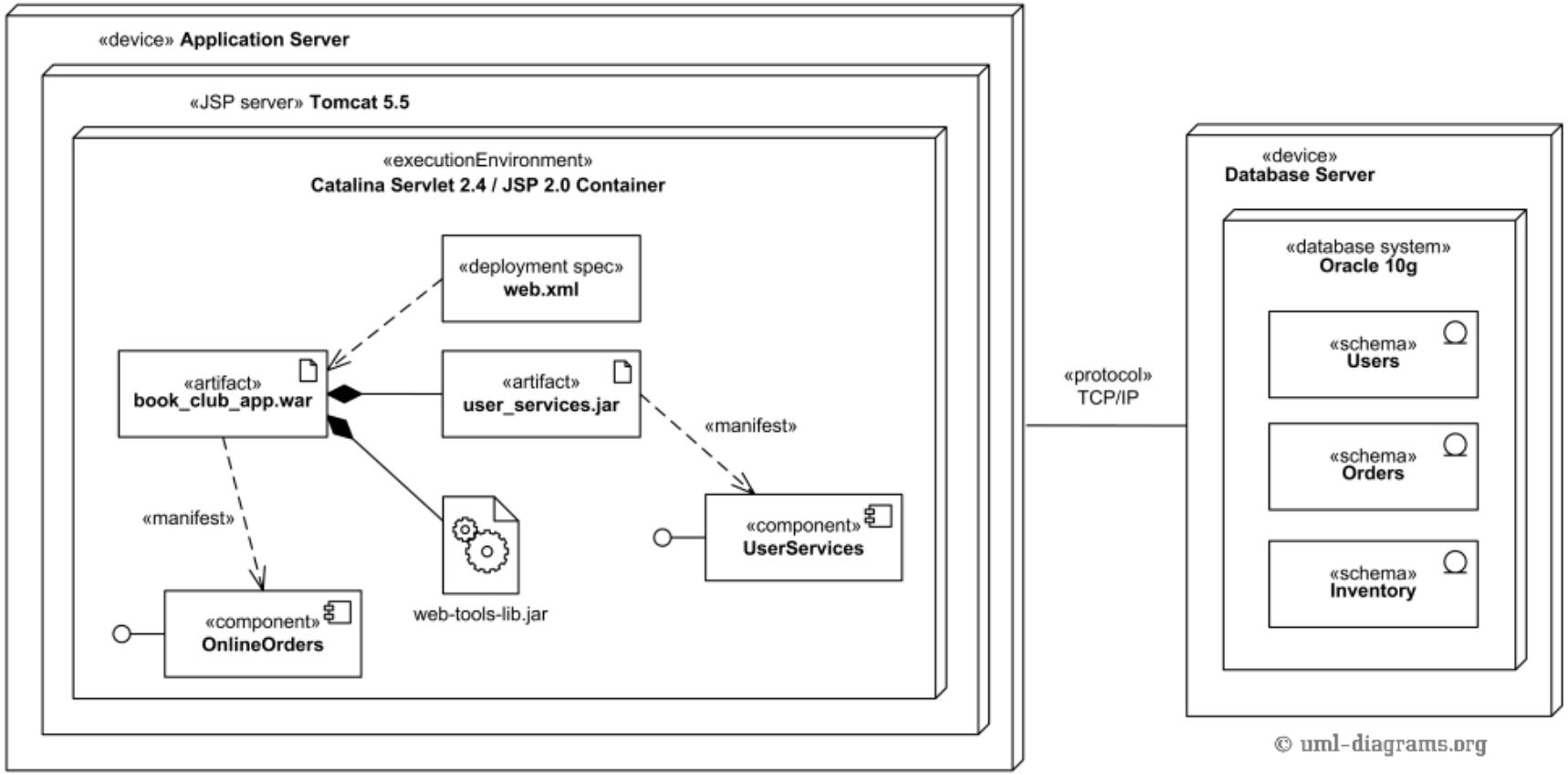
Pros

- Fewer number of physical servers
- Reduced communication overhead between multiple physical servers
- Advantage of simplicity and manageability

Cons

- All the layers share the same physical hardware
- Reduces the overall scalability
- Security

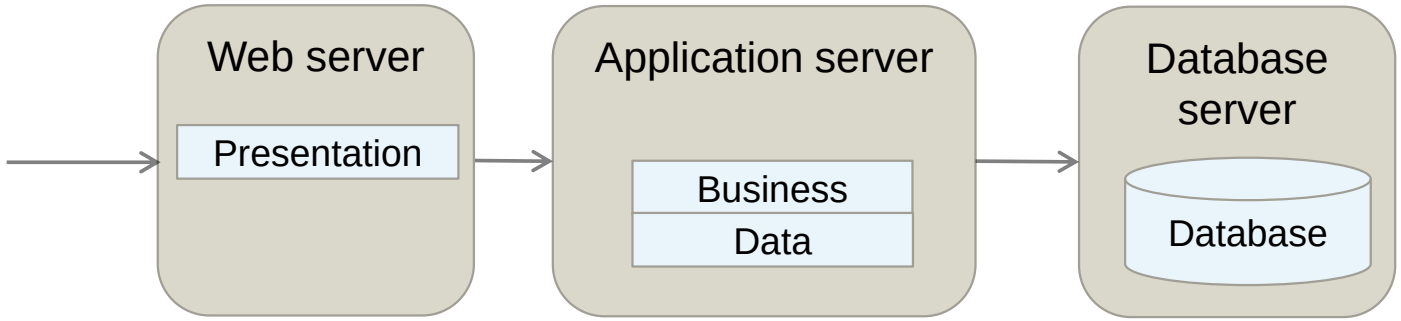
Non-distributed deployment



["UML Deployment Diagrams"]

Deployment strategy

- **Distributed deployment**
  - The layers of the application reside on separate physical tiers.



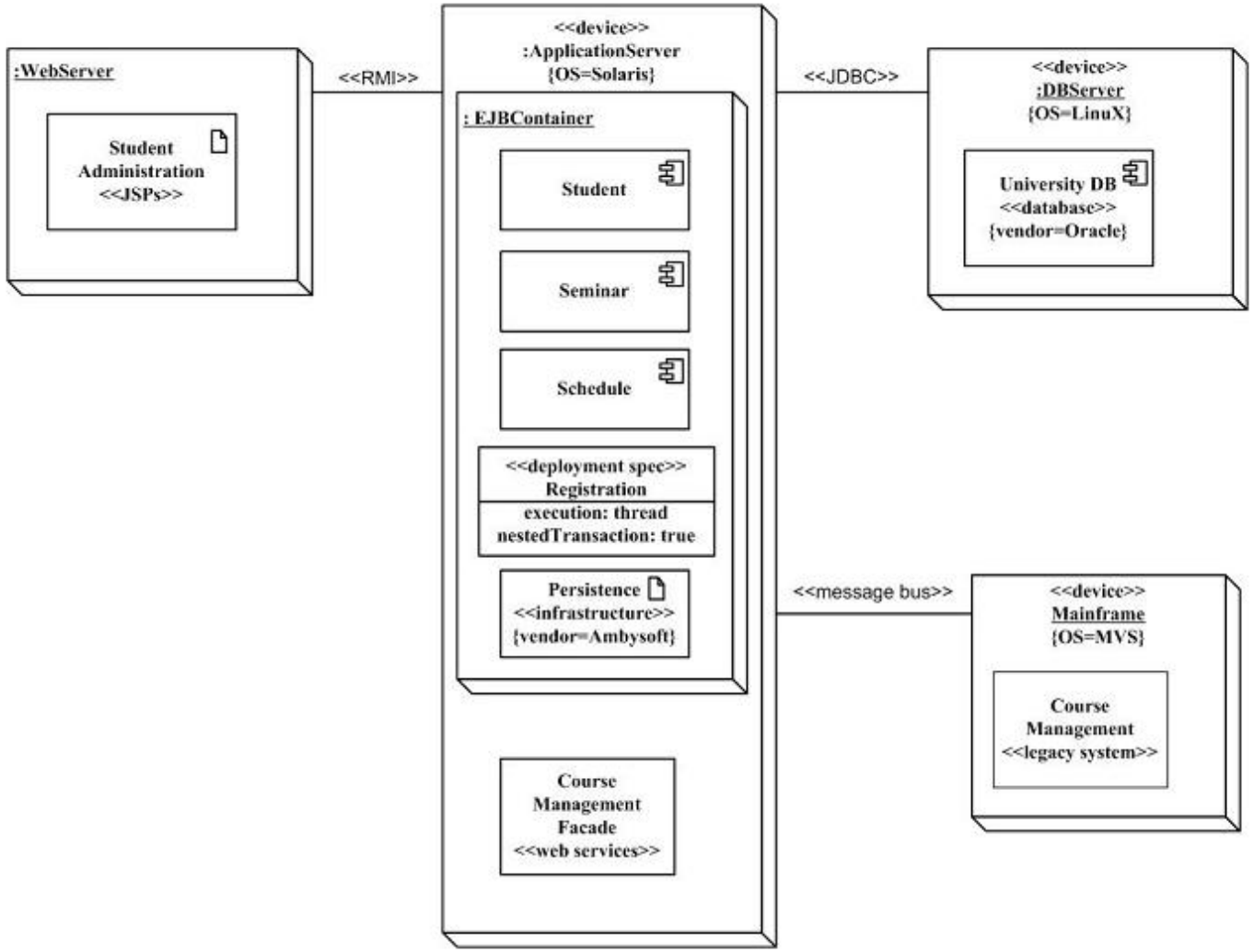
Pros

- More flexible environment
- Easy to scale out or scale up each physical tier
- Allows to add specific security policies to different tiers
- Better Security with DMZs

Cons

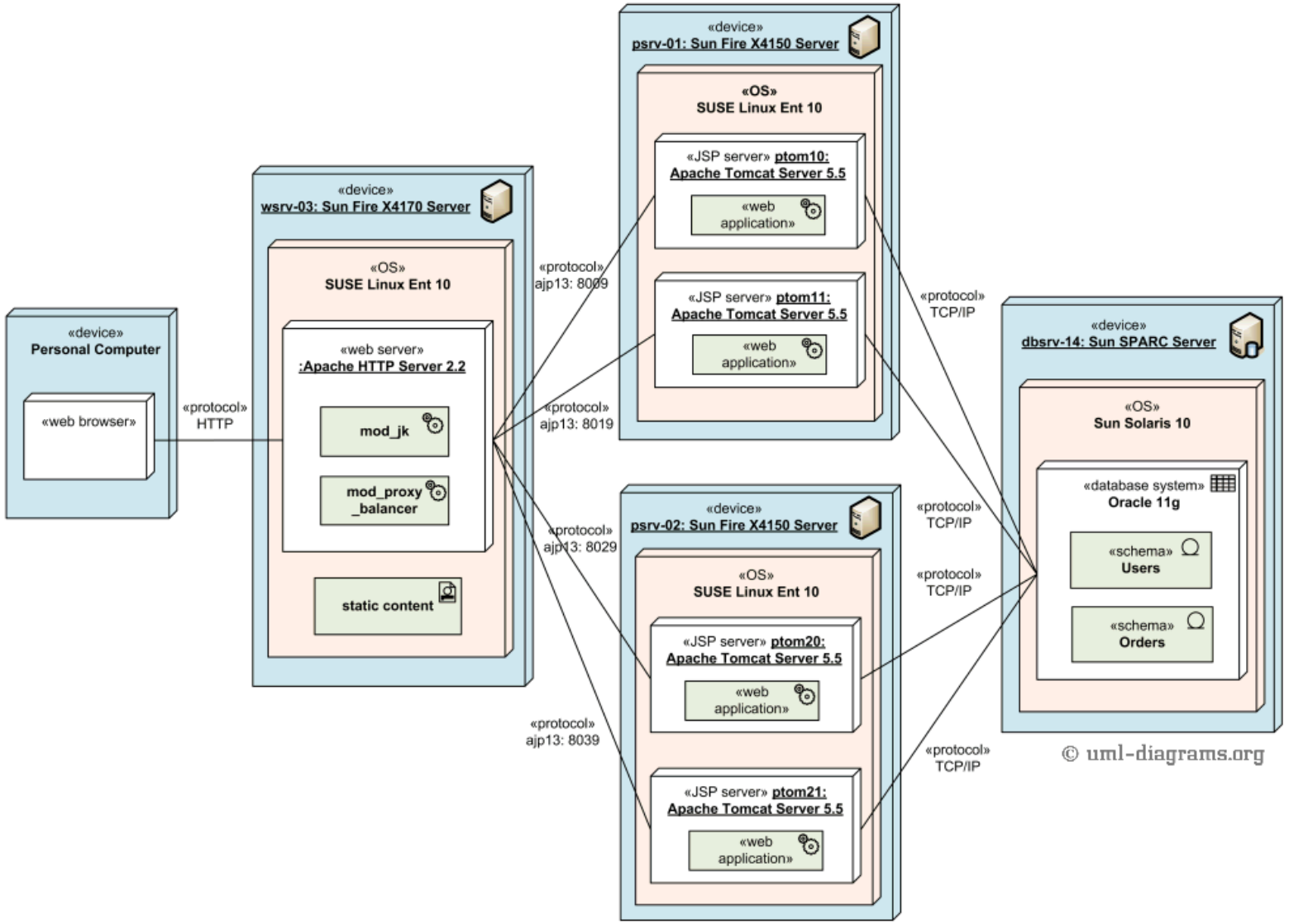
- Adding more tiers increases complexity, deployment effort and cost

Distributed deployment



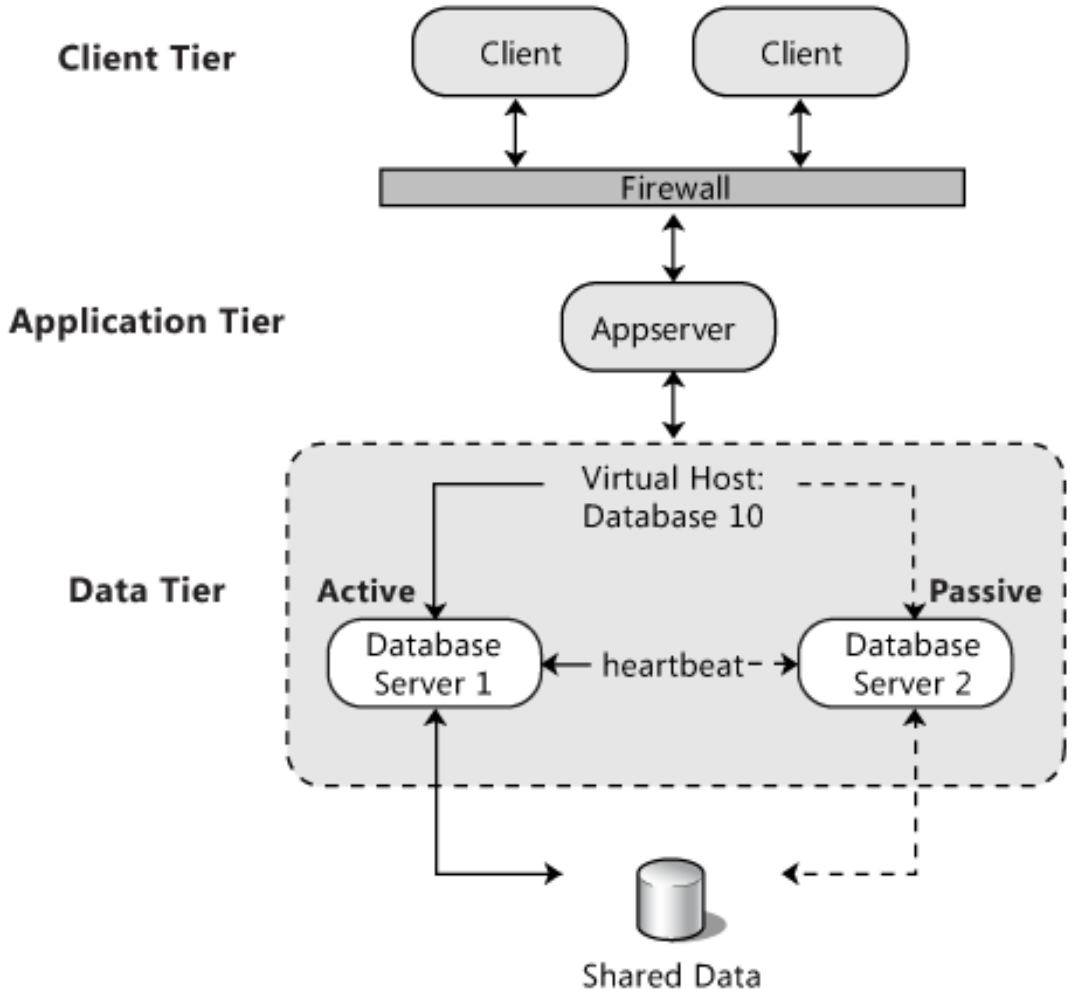
["The Object Primer 3rd Edition Agile Model Driven Development with UML.“ Ambler S. (2004)]

Load balanced cluster



["UML Deployment Diagrams"]

Failover cluster



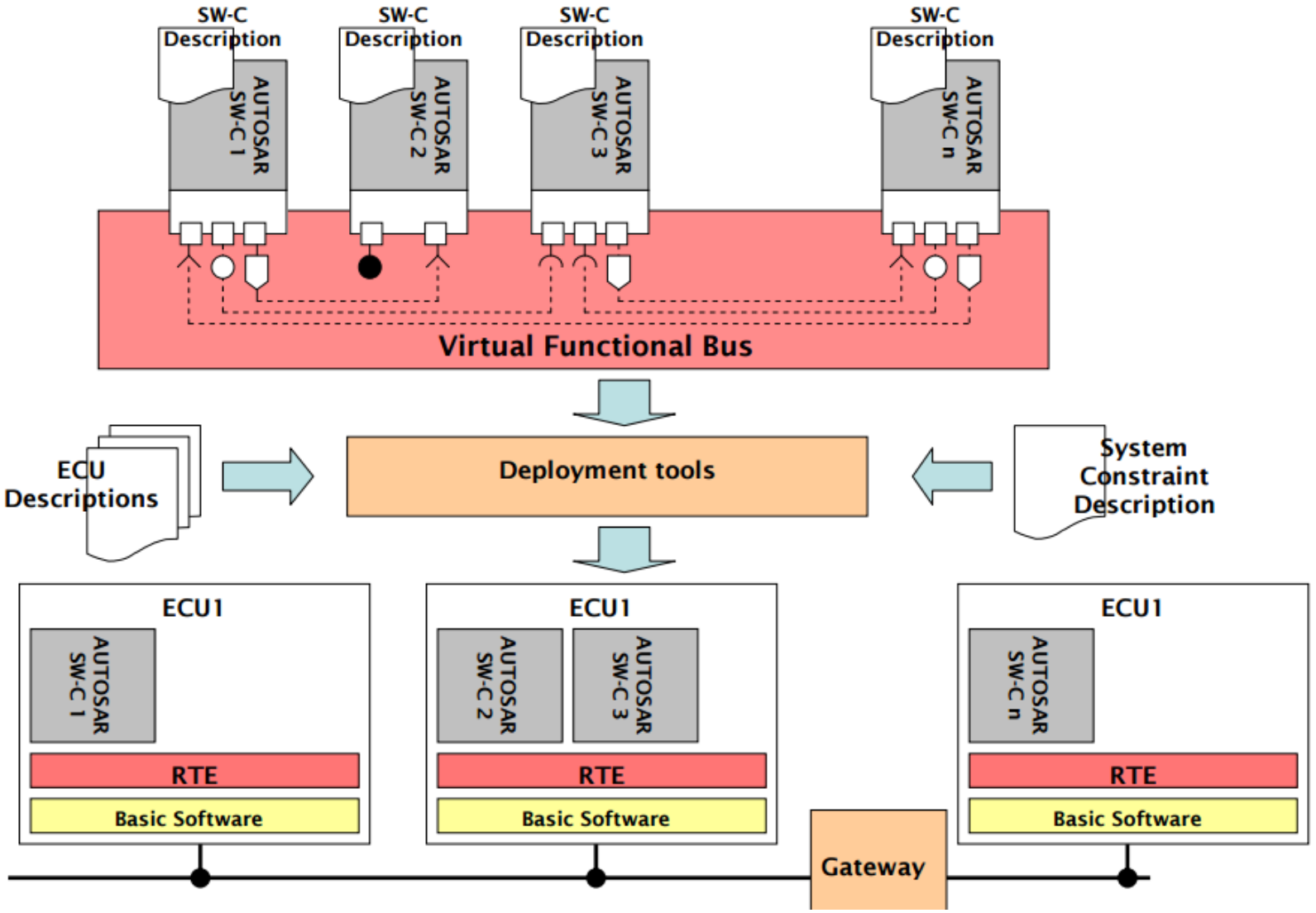
["Microsoft application architecture guide."J.D. Meier et al (2009)]

Software architectures and their trade-offs

AUTOSAR

- **AUT**omotive **O**pen **S**ystem **Ar**chitecture
- Standardized **automotive** reference architecture
- Based on cooperation of major car OEMs and suppliers (BMW, Daimler, Ford, Bosch, Continental, Siemens, …)
- Component-based / Model-based development model
- Focus on non-functional requirements
  - Modularity
  - Scalability
  - **Portability**
  - **Reusability**

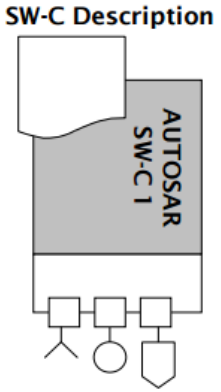
AUTOSAR
Architecture



[Embedded Systems Course - An introduction to AUTOSAR]

AUTOSAR SW-C

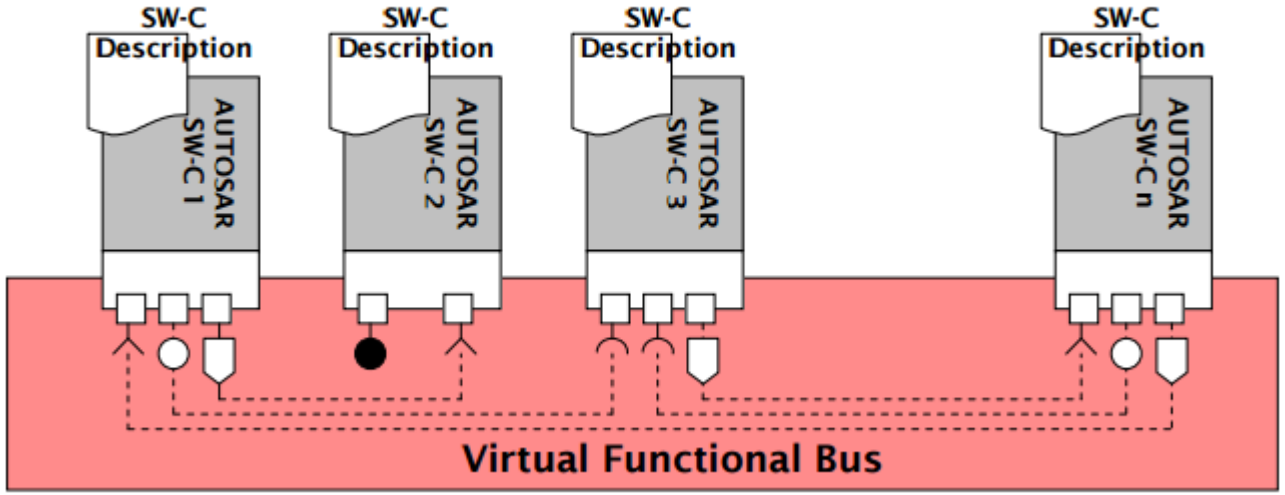
- The AUTOSAR Software Components encapsulate an application which runs on the AUTOSAR infrastructure. The AUTOSAR SW-C have well-defined interfaces, which are described and standardized.
- AUTOSAR provides a standard description format (SW-C Description) for SW-C



[Embedded Systems Course - An introduction to AUTOSAR]

Virtual Functional Bus (VFB)

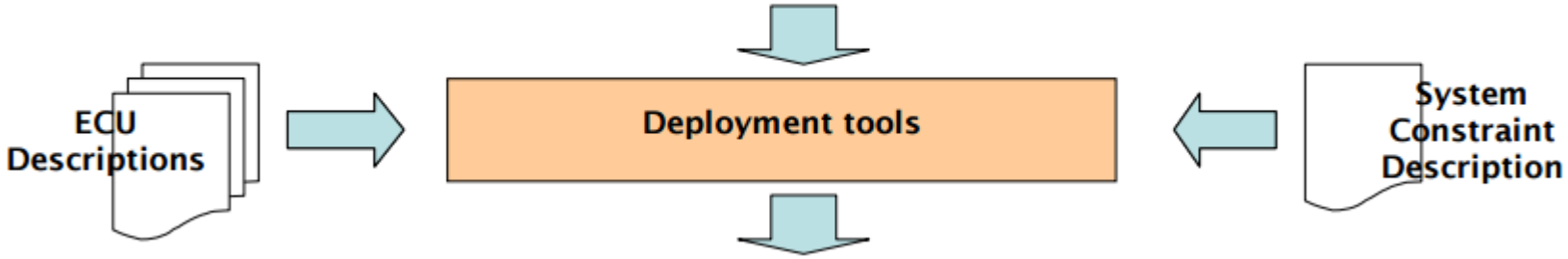
The VFB is the sum of all communication mechanisms (and interfaces to the basic software) provided by AUTOSAR on an abstract (technology independent) level. When the connections for a concrete system are defined, the VFB allows a virtual integration in an early development phase.



[Embedded Systems Course - An introduction to AUTOSAR]

System Constraint and ECU Descriptions

In order to integrate AUTOSAR SW-Components into a network of ECUs, AUTOSAR provides description formats for the system as well as for the resources and the configuration of the ECUs.

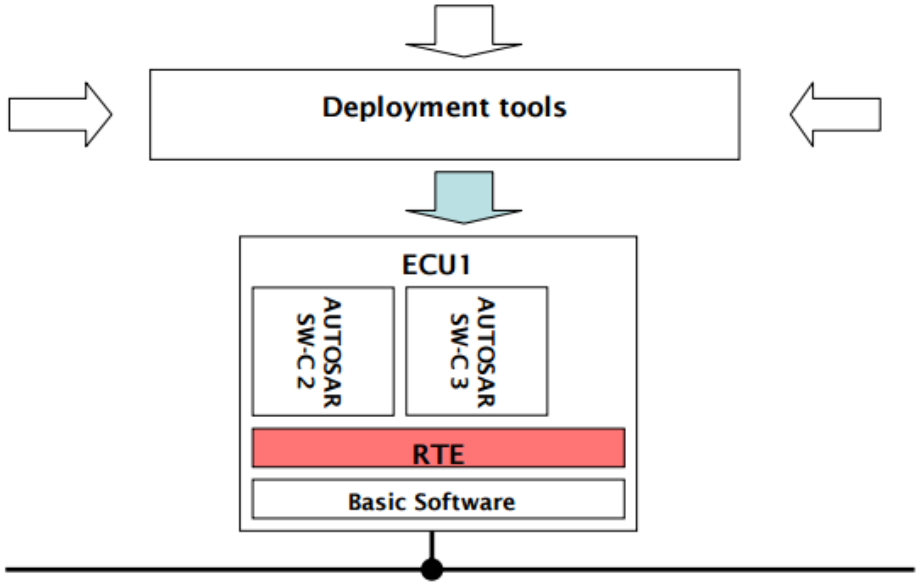


[Embedded Systems Course - An introduction to AUTOSAR]

AUTOSAR defines the methodology and tool support to build a concrete system of ECUs. This includes the configuration and generation of the Runtime Environment (RTE) and the Basic Software on each ECU.

Runtime Environment (RTE)

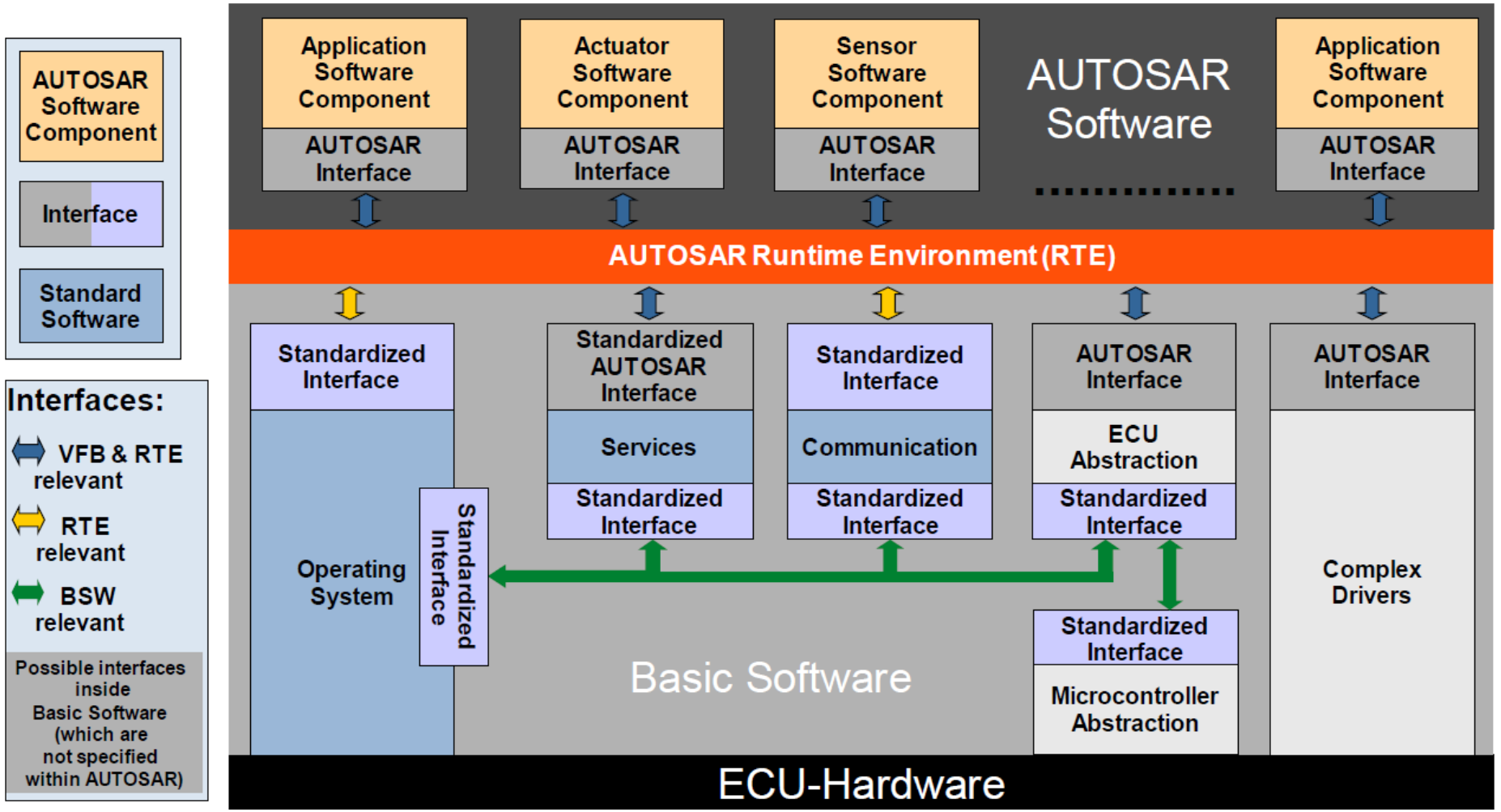
- From an abstract point of view (at system design time), the Run Time Environment (RTE) is the run-time implementation of the Virtual Functional Bus on a specific ECU.



["Autosar runtime environment and virtual function bus." Nico Naumann (2009)]

AUTOSAR

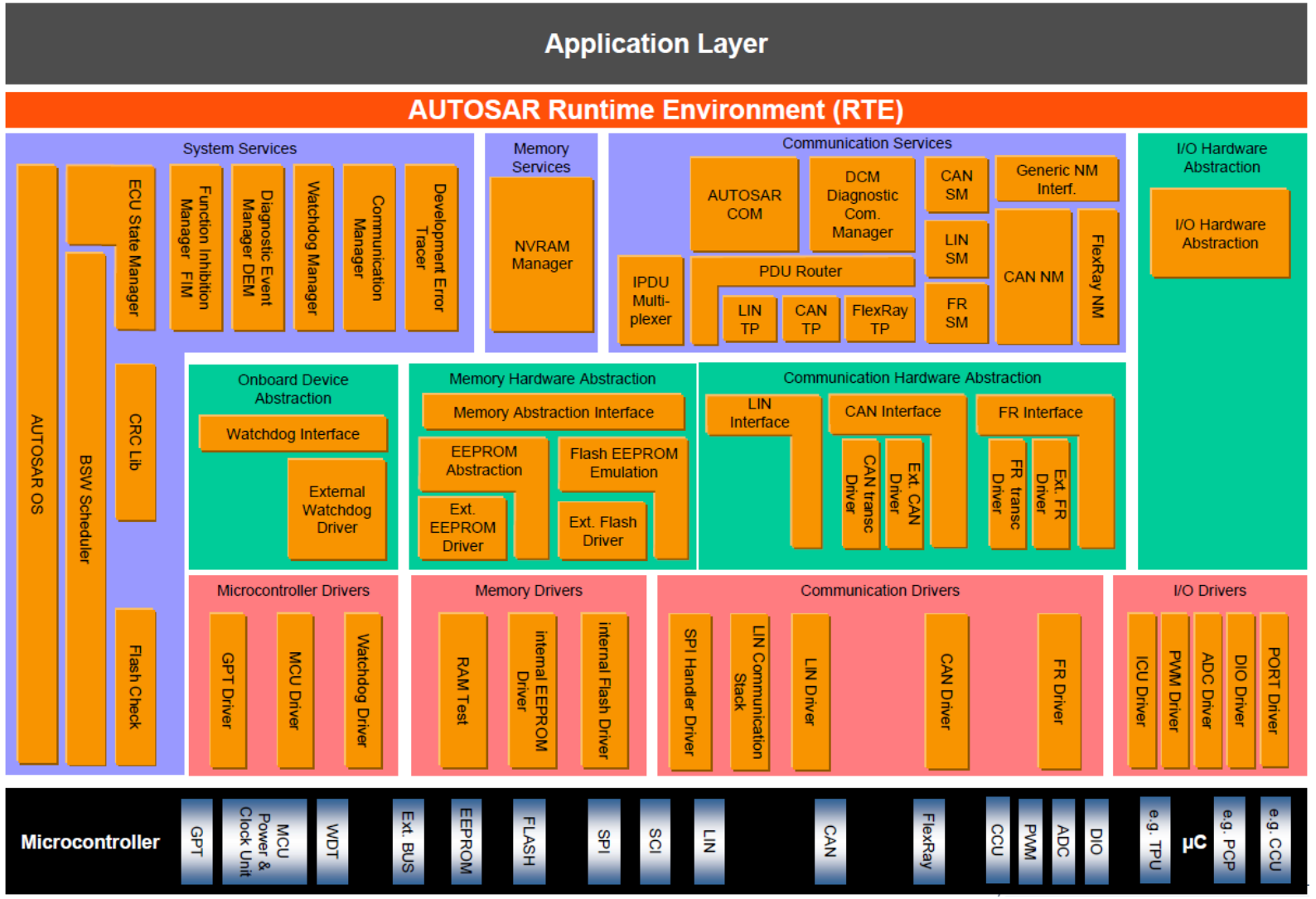
ECU Architecture



["Autosar–A worldwide standard current developments, roll-out and outlook.“Kirschke-Biller Frank (2011)]

AUTOSAR

Module View (detailed)



[Embedded Systems Course - An introduction to AUTOSAR]

- **Application layer**
  - Components developed against platform-independent RTE
  - Components realize functionality of sensors and actuators

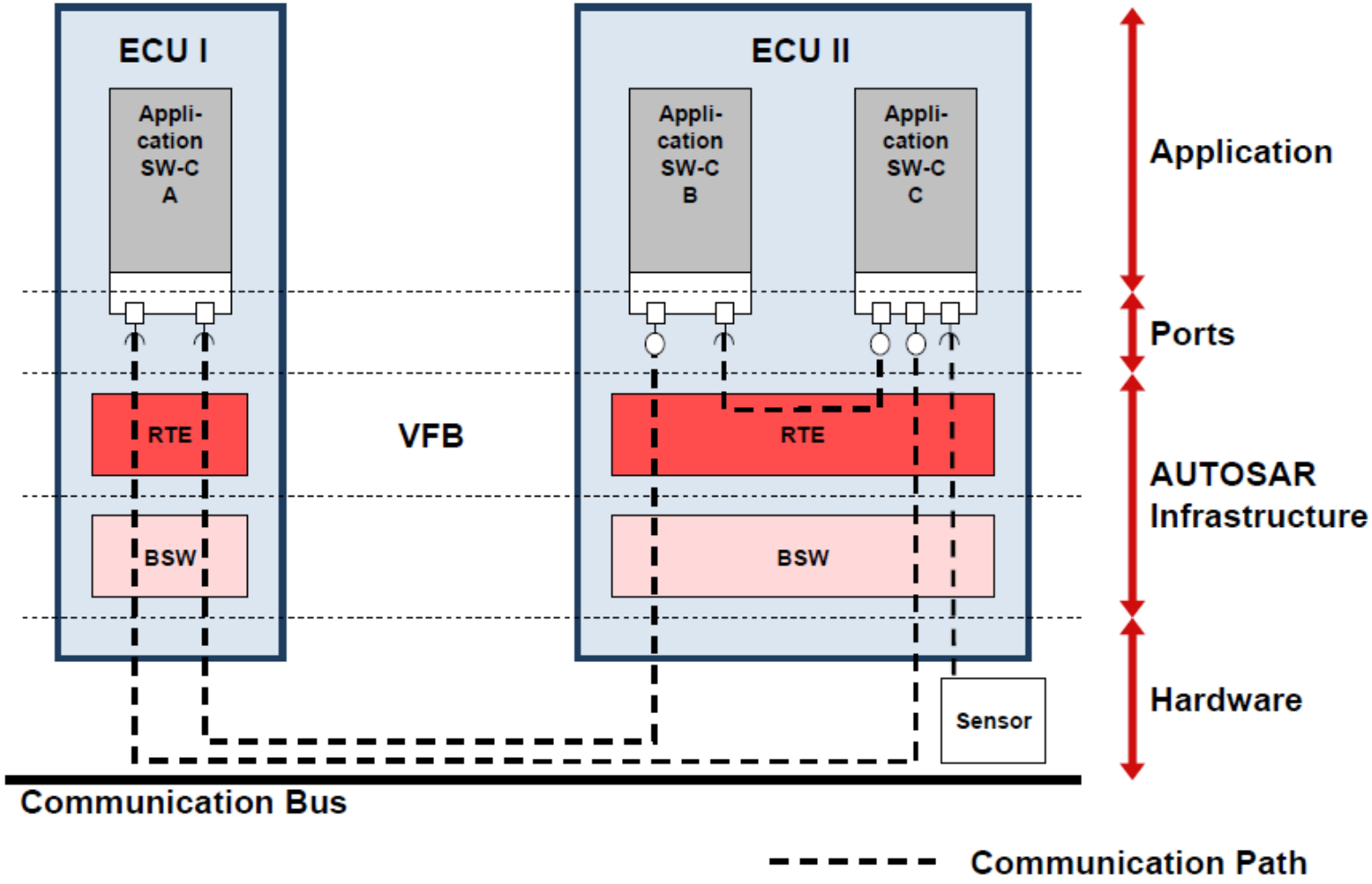
- **AUTOSAR RTE layer**
  - Defined interfaces to Basic Software standard library functions
  - Controls interaction between components and BS layer

- **Basic Software layer**
  - Contains operating system and offers variety of services (e.g. memory management, communication)

- **Microcontroller Abstraction layer**
  - Abstraction from I/O and communication with microcontroller

AUTOSAR

C&C view

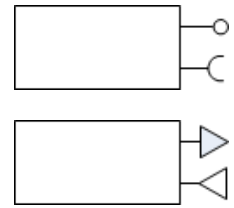


["Achievements and exploitation of the AUTOSAR development partnership." Helmut Fennel et al. (2006)]

AUTOSAR

C&C view

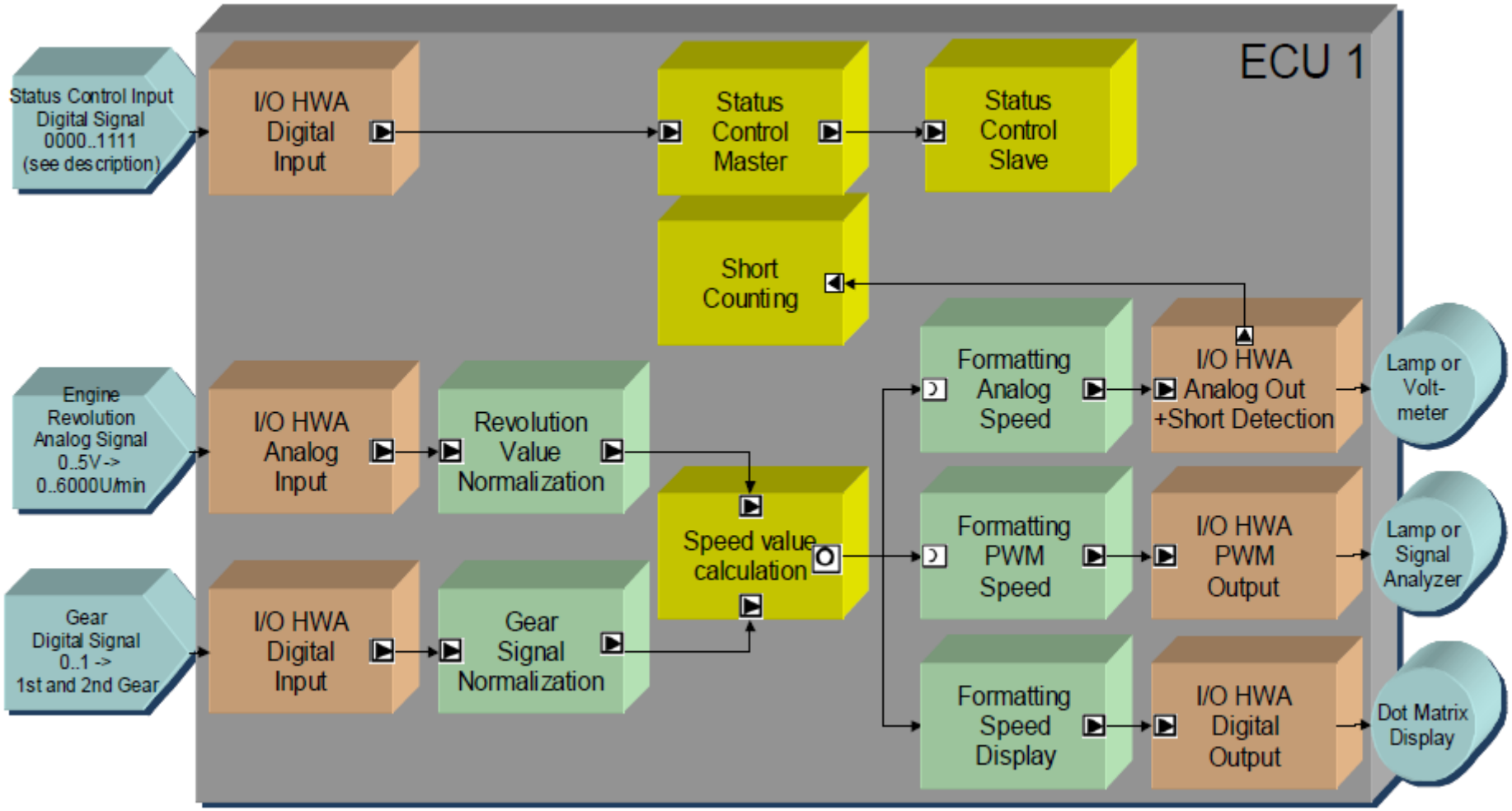
- Components have ports as communication endpoints
- Ports implement the component interfaces
- Connections routed / managed by Runtime Environment
- AUTOSAR supports different connection paradigms
  - **Client/Server** → Client-Server C&C style (synchronous, blocking)
  - **Sender/Receiver** → Publish-and-Subscribe style (asynchronous, non-blocking)



- Examples
  - Broadcast of error message → Sender/Receiver (one-way)
  - Close door message → Client/Server (two-way, close + isClosed)

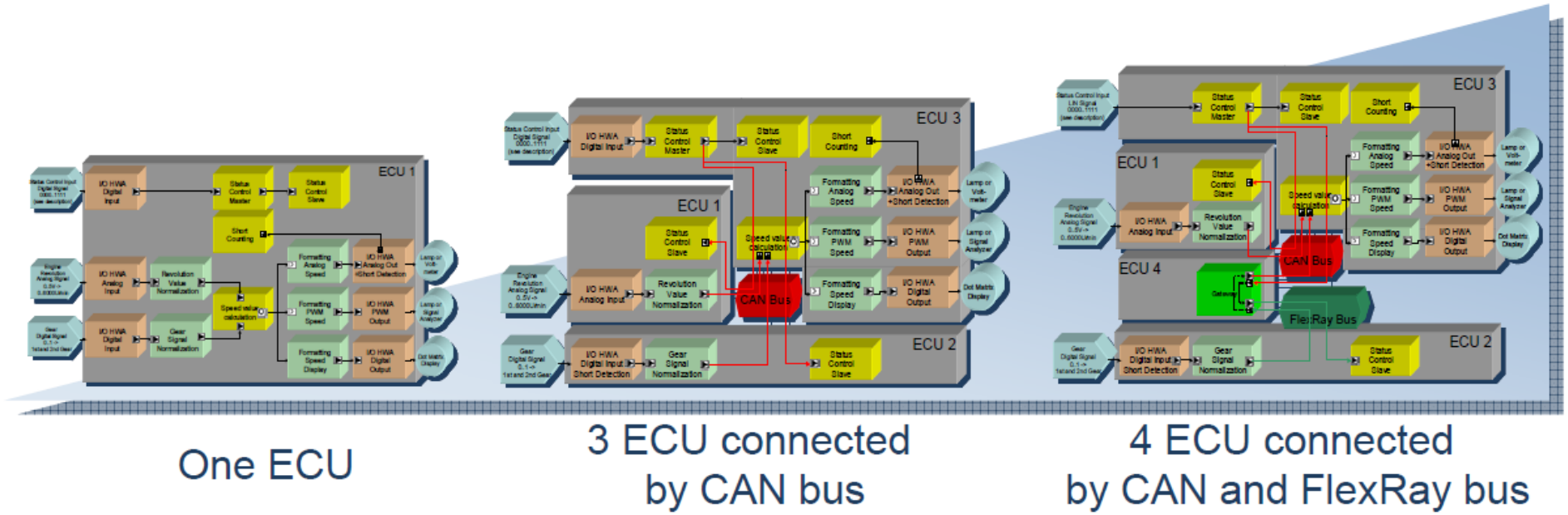
AUTOSAR

Allocation view



AUTOSAR

Allocation view



One ECU

3 ECU connected by CAN bus

4 ECU connected by CAN and FlexRay bus

- Different deployment possibilities
- Non ECU-specific components almost arbitrarily deployable
- One typical reason for distribution: increase safety by redundancy

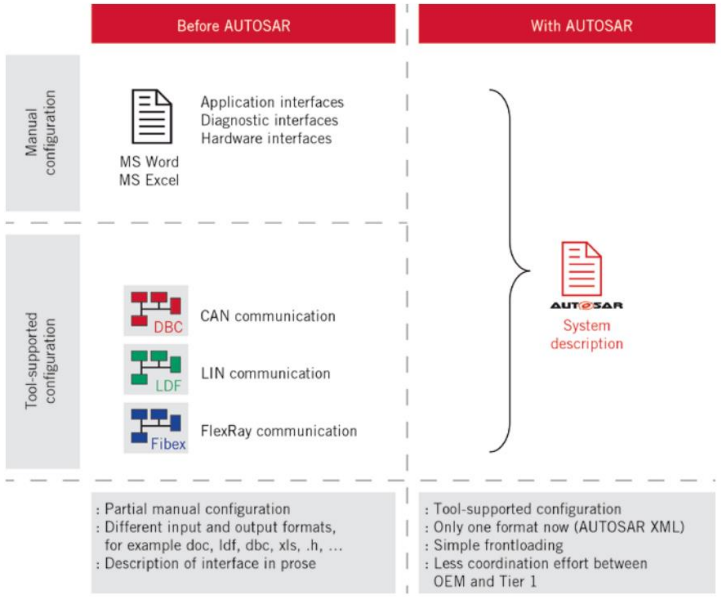
AUTOSAR

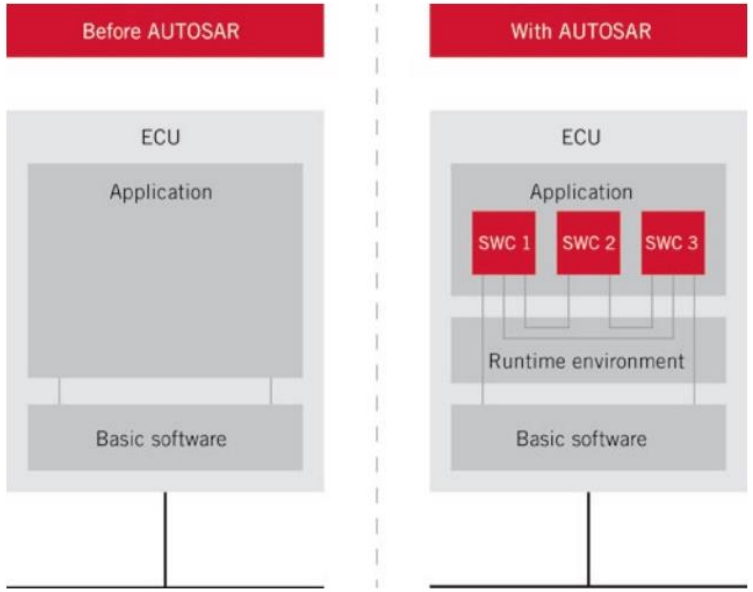
Allocation view

- Allows deployment of components on different nodes
- All nodes have an AUTOSAR infrastructure
- Communication between components via abstract bus interface
  → **Virtual Functional Bus**
- Deployment of purpose-specific components (e.g. light density control component) to corresponding ECUs
- Distributed deployment of vehicle-wide components
  - Intra-ECU communication over RTE
  - Inter-ECU communication over physical buses

AUTOSAR

Wrap-up





- Component-based automotive reference architecture
- **Goals**
  - Increase interoperability between different OEMs and suppliers via standardization
  - Decouple development from specific hardware
- **Core NFR emphasis**
  - Reusability
  - Portability
- **Consists of**
  - (Reference) architecture
  - Methodology
  - (Standardized) interfaces

["AUTOSAR Methodology in Practice." Ralf Belschner et al. (2013)]

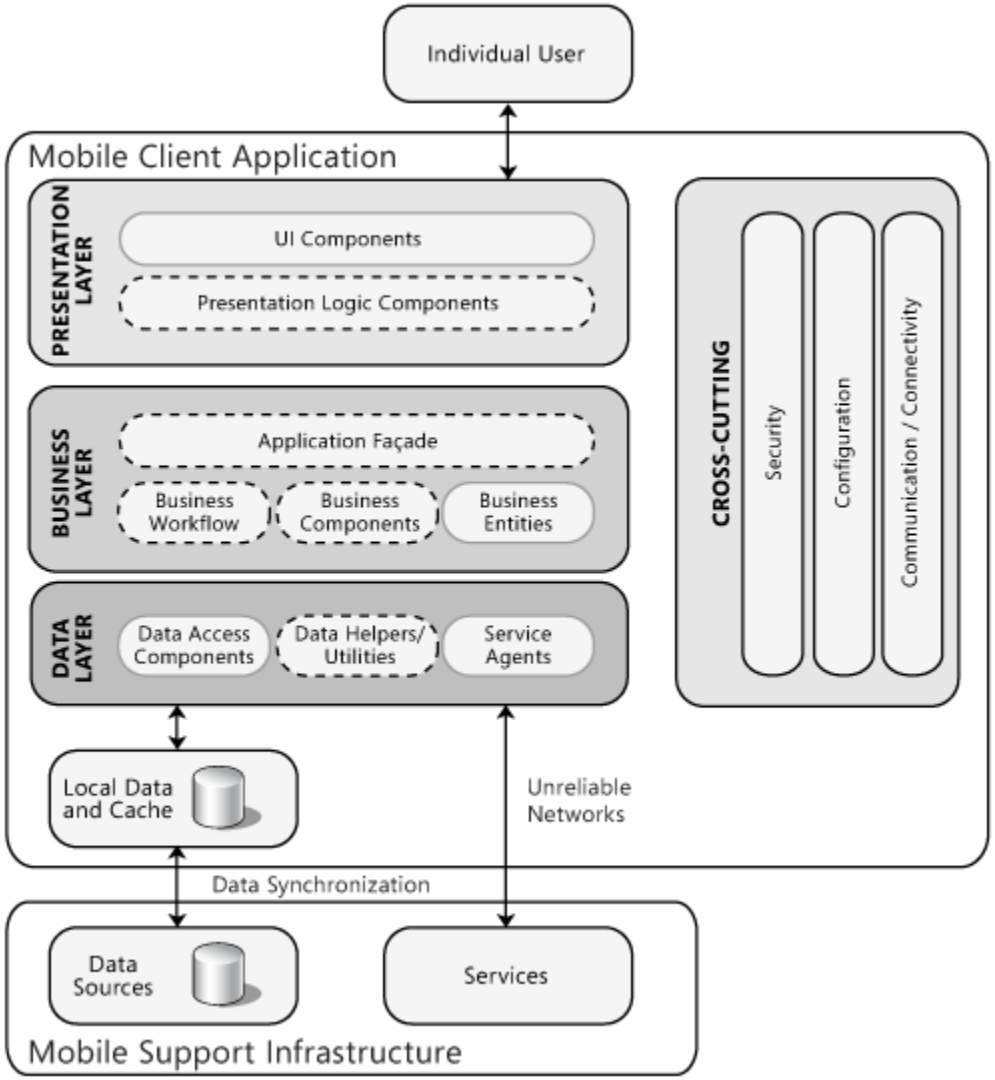
Software architectures and their trade-offs

Mobile application architectures

Basic requirements

- Your users depend on handheld devices.
- Your application supports a simple UI that is suitable for use on small screens.
- Your application must support offline or occasionally connected scenarios. In this case, a mobile rich client application is usually the most appropriate.
- Your application must be device-independent and can depend on network connectivity. In this case, a mobile Web application is usually the most appropriate.

The typical structure of a mobile application



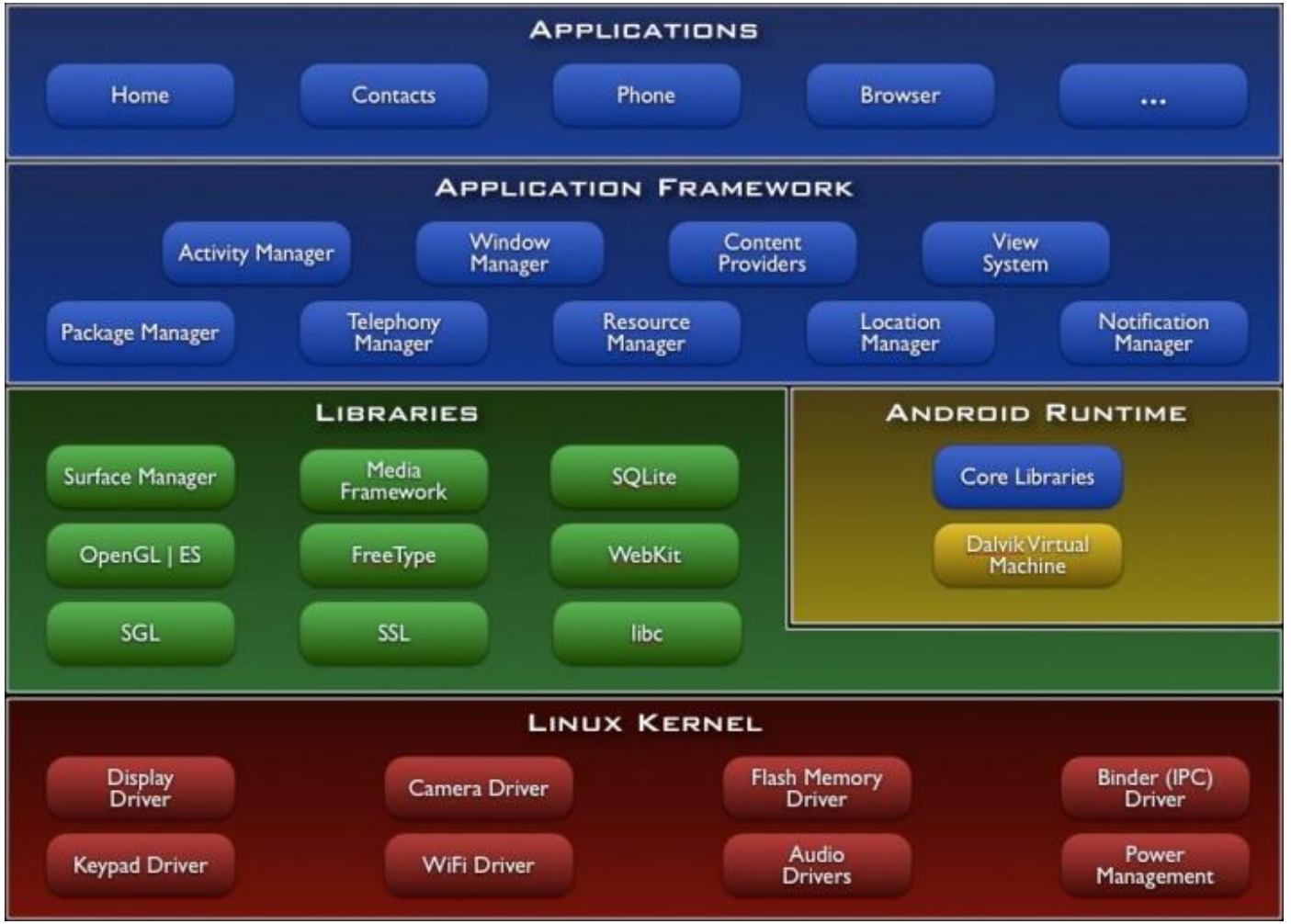
Android

- Open-source mobile system reference architecture
- Initiated by Open Handset Alliance under lead of Google (Google, HTC, Samsung, LG, Sony, Motorola, …)
- Combines operating system, middleware, and applications
- Component-based reference architecture → Apps (Java)
- Deployment focus: touch smartphones and tablets
- NFR focus
  - Modularity
  - Portability
  - Reusability
  - Security

- **Typical 4/5 layer architecture** → layered style
- Layers
  - Application Layer
  - Framework Layer
  - Library Layer
  - Runtime Layer
  - Kernel Layer
- **High extensibility**
  - Application Layer components arbitrarily deployable
- **High information hiding**
  - Every layer hides internal details from above layer

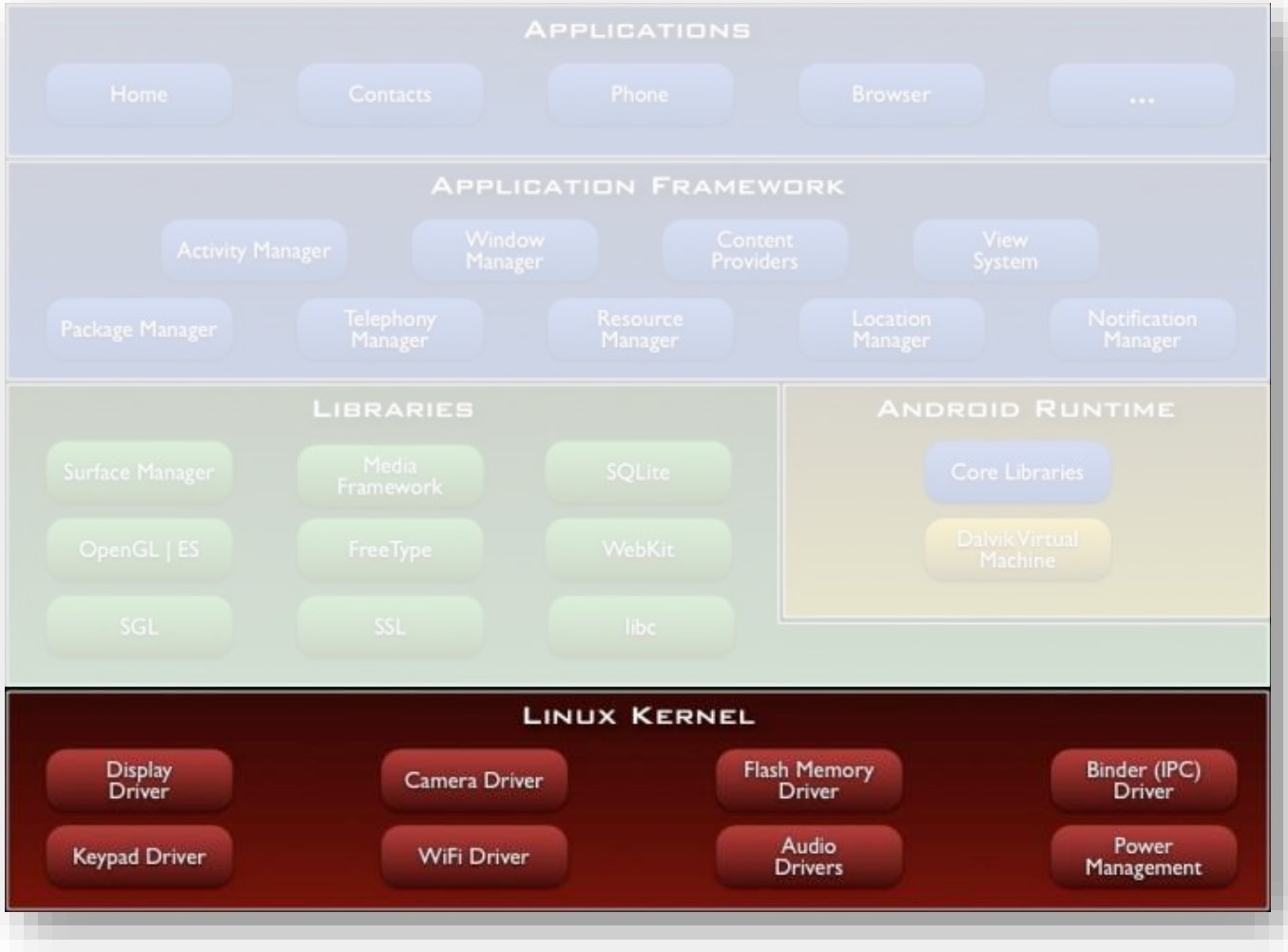
Android

Module view



Android

Module view

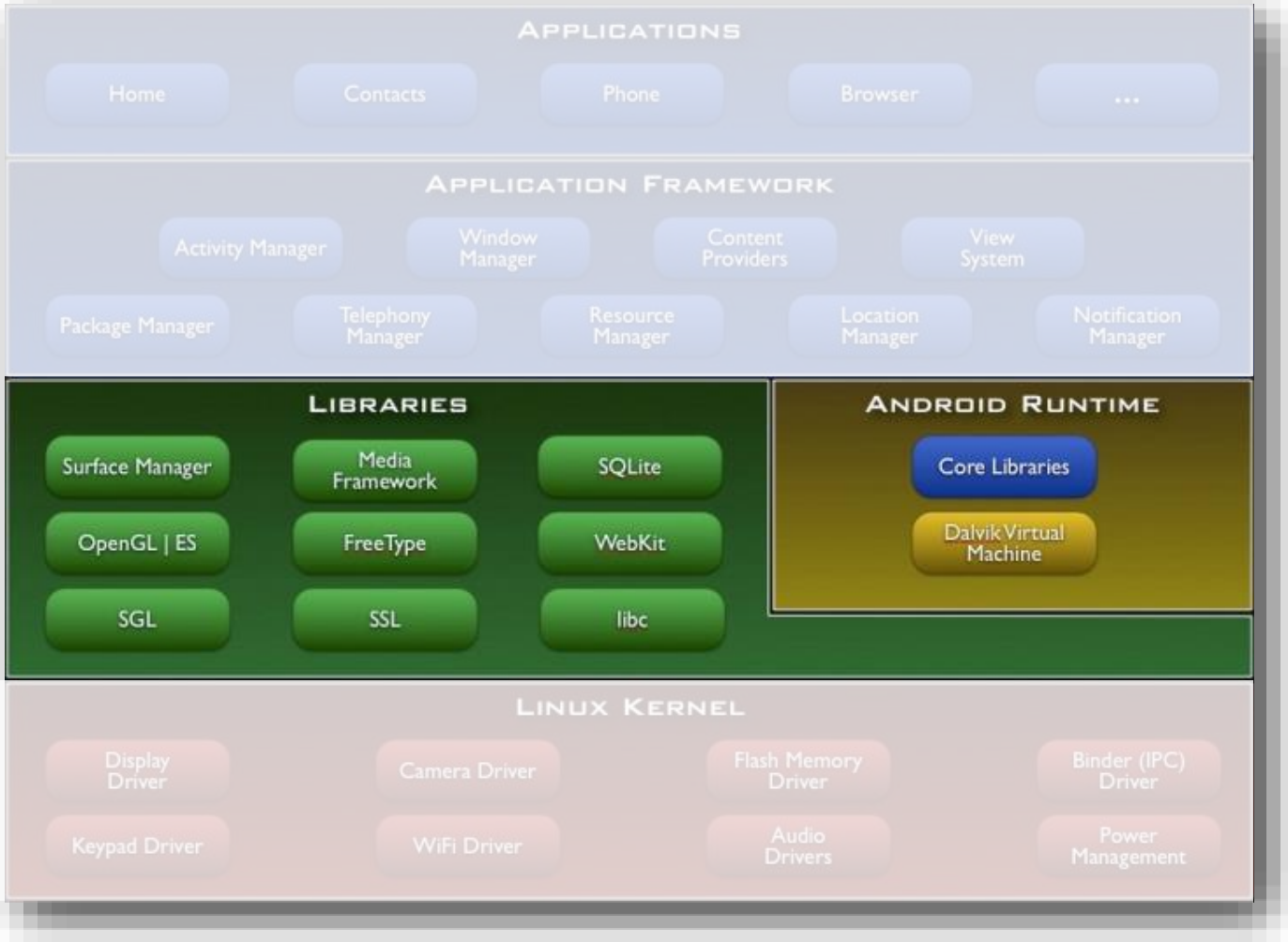


Linux Kernel Layer

- Memory management
- Process management
- Network stack
- Drivers
- Security mechanisms
- Hardware abstraction

Android

Module view



Library Layer

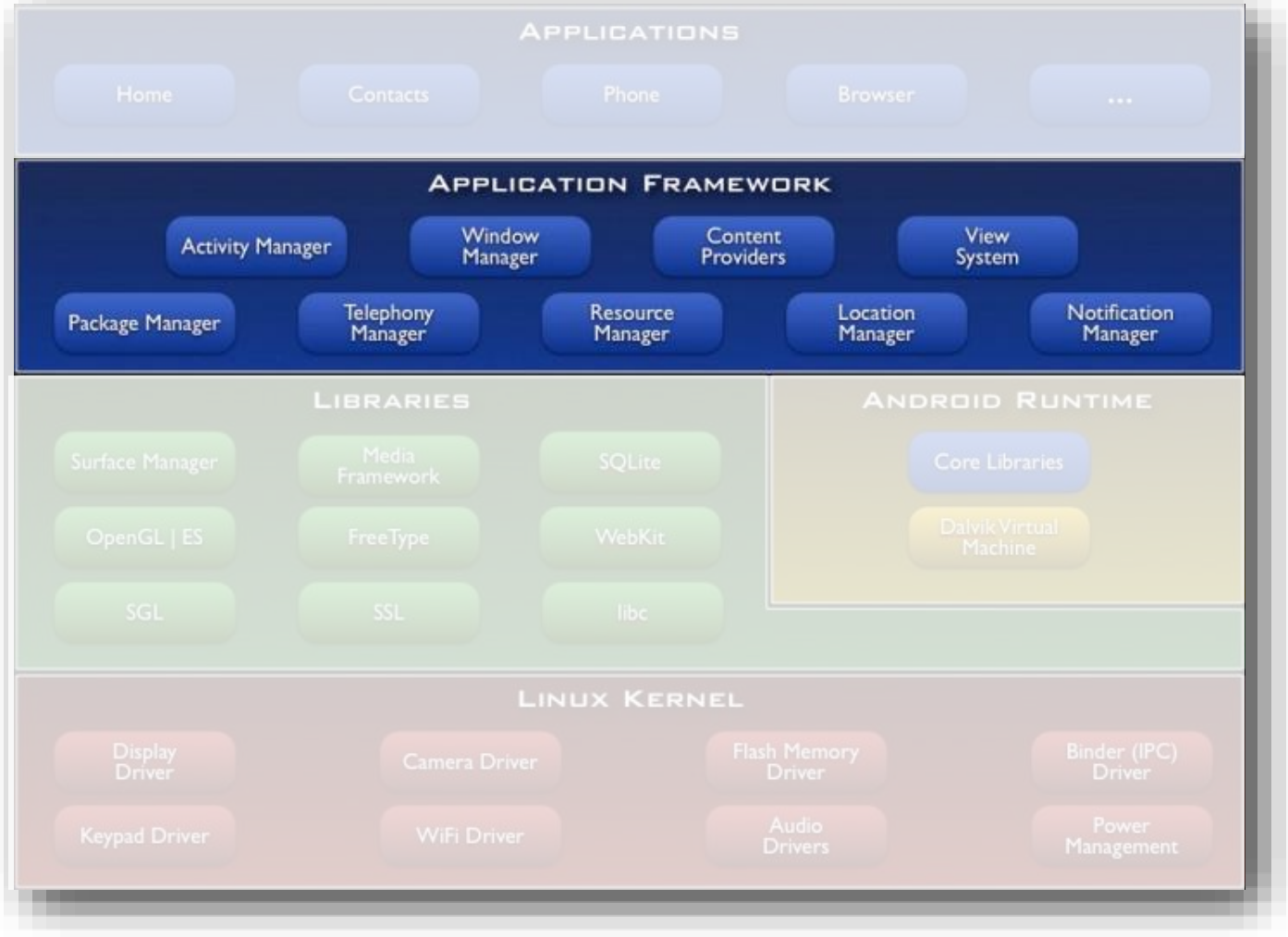
- Exposed via the application framework
- Hidden from Apps
- Embedded in firmware
  → C/C++ components

Runtime Layer

- Dalvik VM (Java)
- Optimized for performance
- Core Libraries for Java
- Sandboxed APPs
  → 1 App per VM
  → Security

Android

Module view

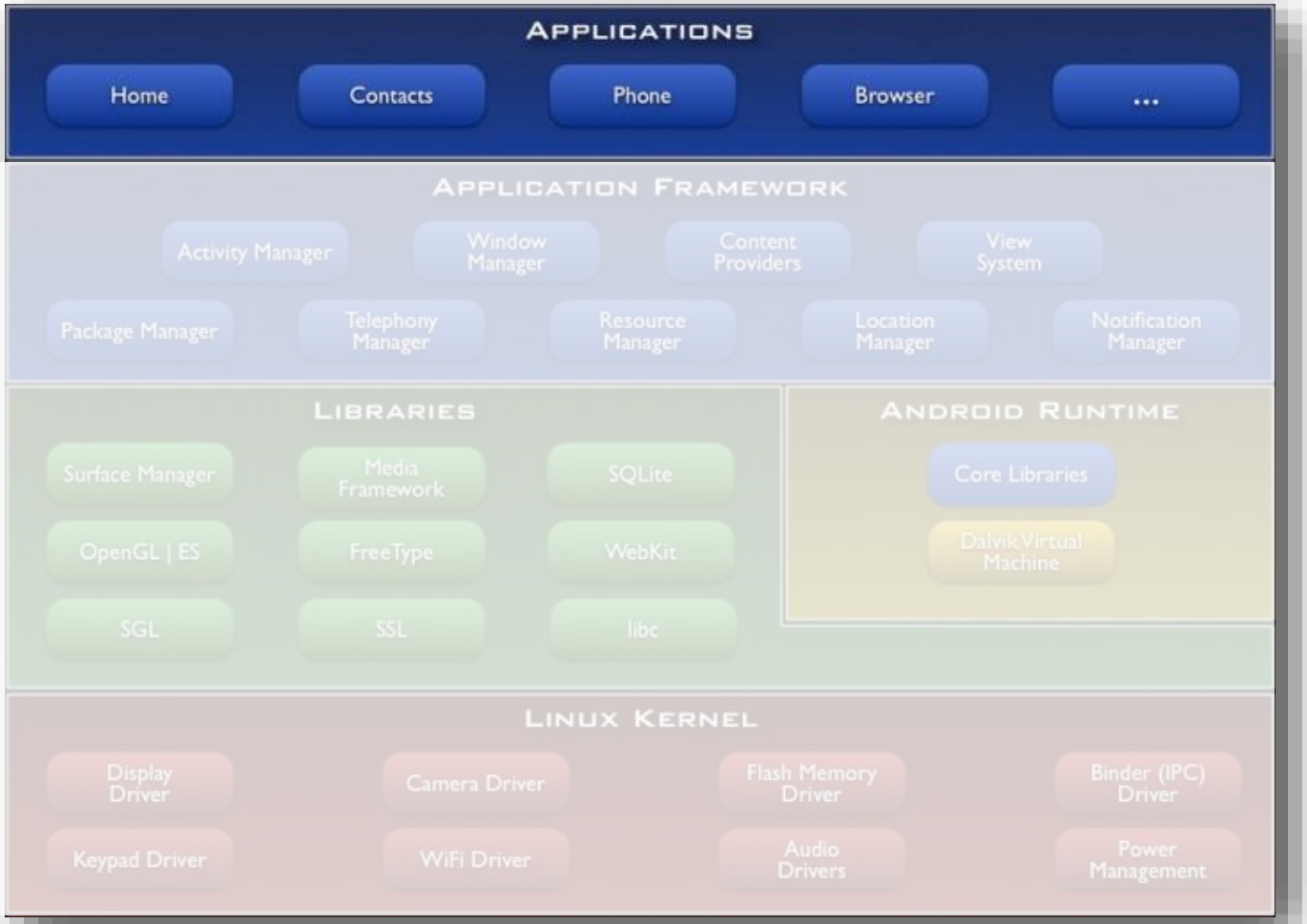


Framework Layer

- Open development platform
- Exports functionality of Library Layer via APIs
- Design for reuse – Applications can provide new APIs

Android

Module view



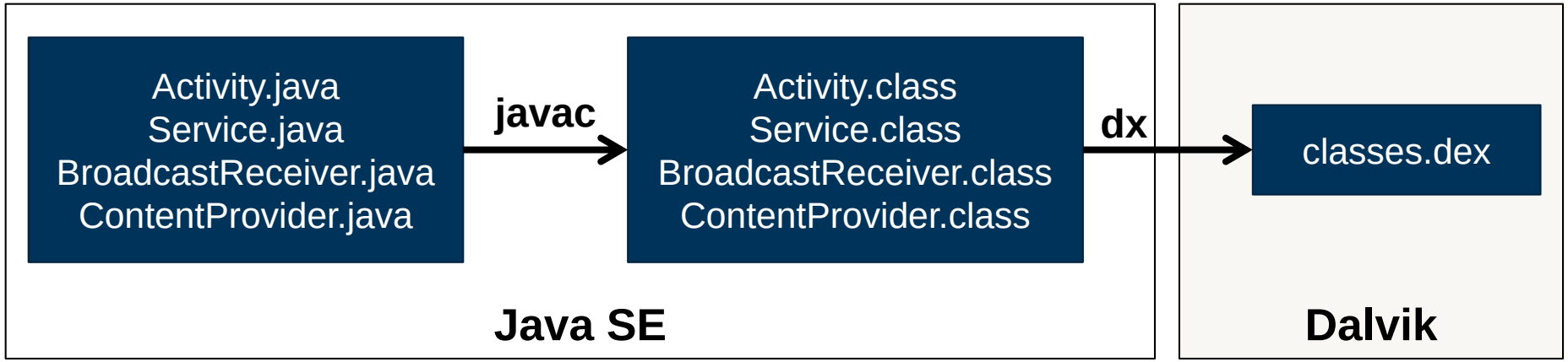
Application Layer

- Set of core applications
- Extendable by apps from market
- Plug-and-play concept
- Based on Java
  → Portability
- Realize actual „business logic“

Android

App development

- App development based on AndroidSDK
- Source code in standard Java (+ Android APIs)
- Compiled with Java compiler to Java Bytecode → .class
- Converted to Dalvik-compatible executables → .dex

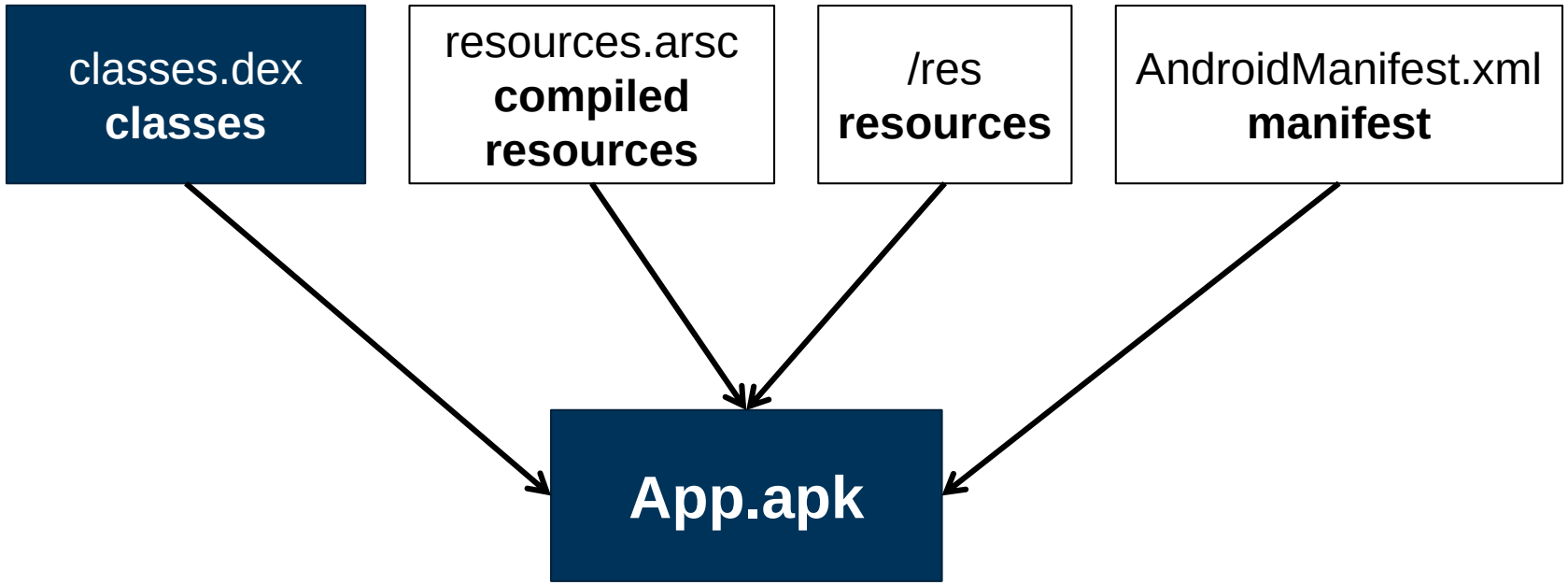


Android - example

Allocation view (implementation)

Implementation style

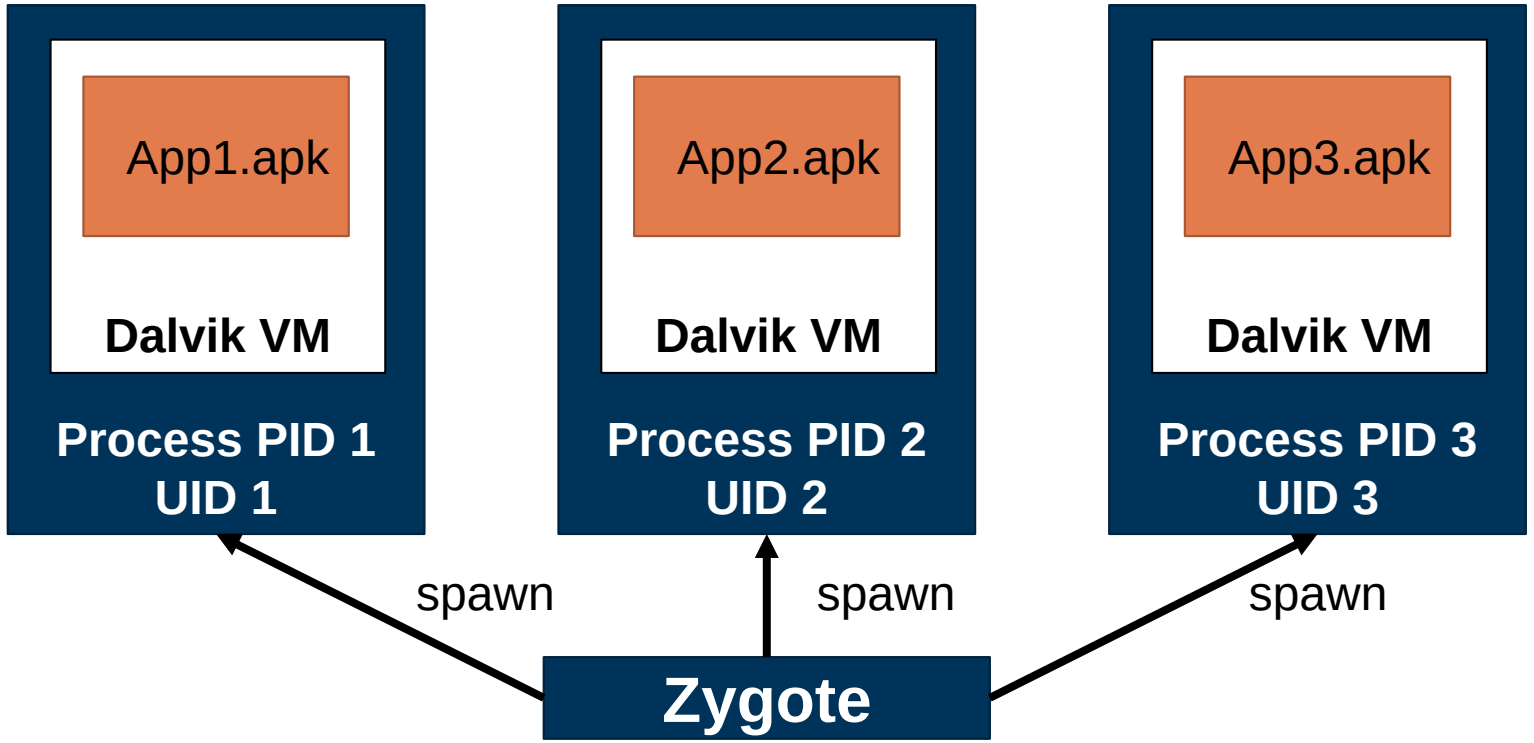
- Deployable Android App: App.apk
- **App.apk** consists of **classes** + **resources** + **manifest**



Android - example

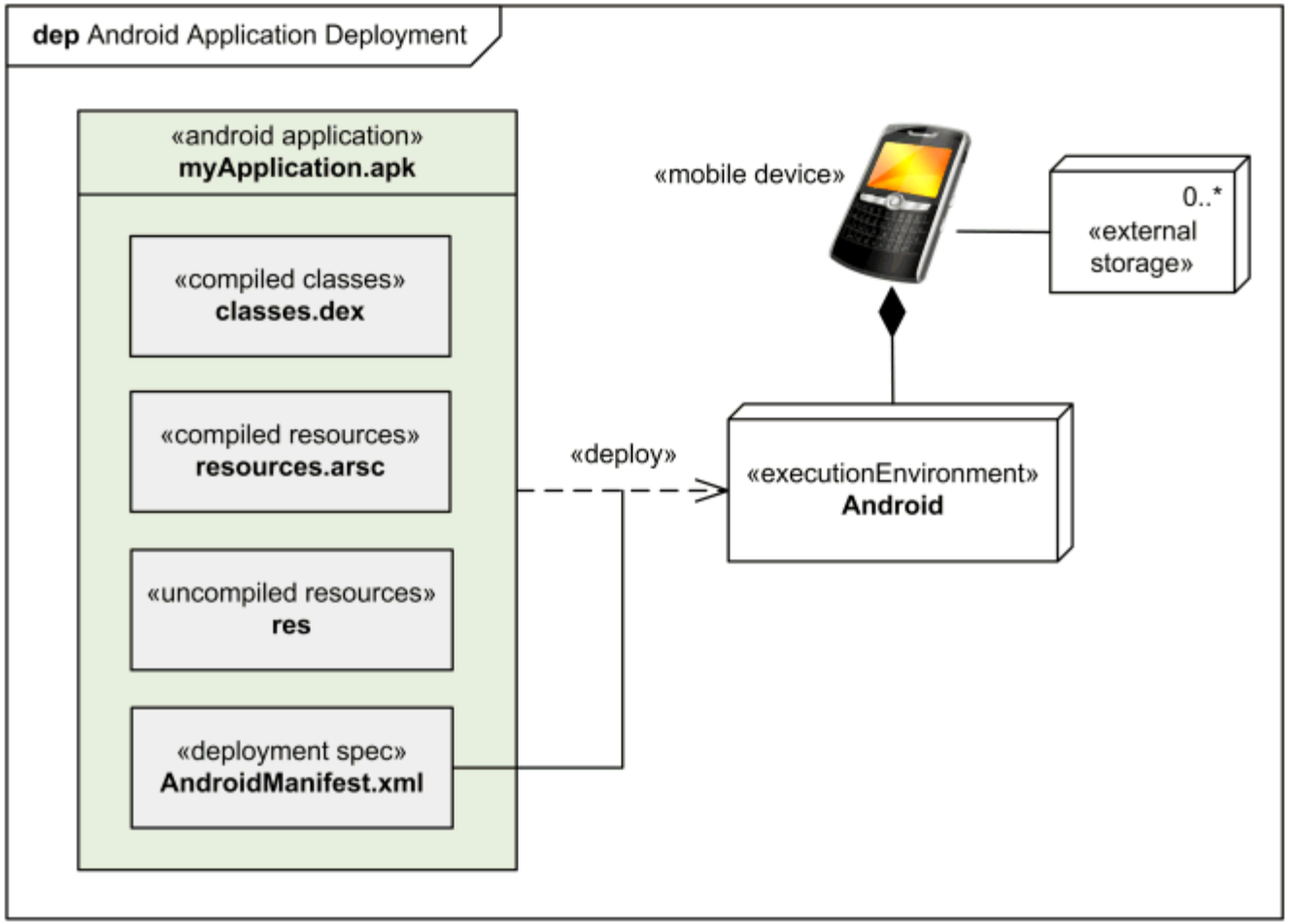
Allocation view (deployment)

Sandboxing of Apps into own processes with own VMs
→ security by isolation



Android

Application deployment



OSGi – Services in a JVM

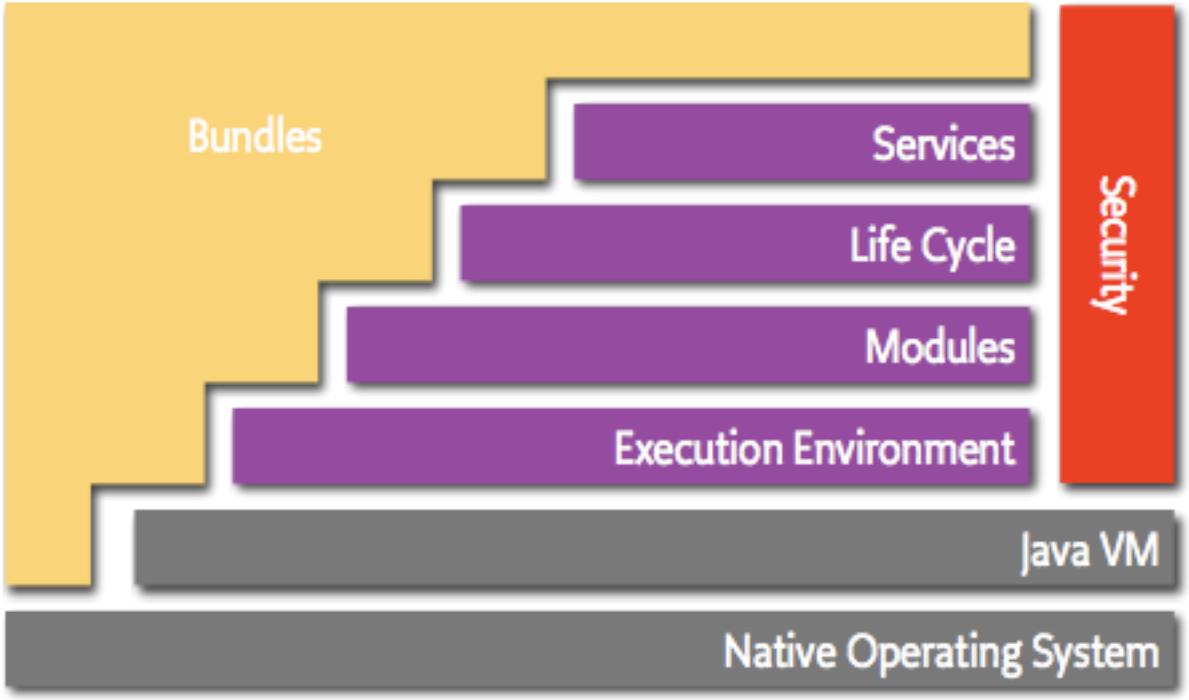
The **O**pen **S**ervices **G**ateway **i**nitiative (OSGi) technology is a set of *specifications* that define a dynamic component system for Java. These specifications enable a development model where applications are (dynamically) composed of many different (reusable) components.

- The OSGi specifications enable components to hide their implementations from other components while communicating through *services*
  - From POJOs to services
- No standard implementations but many frameworks
  - (Eclipse Equinox, Apache Felix, Knoplerfish, …)

- This simple model has far reaching effects for almost any aspect of the software development process.

[The OSGi Architecture]

OSGi – layered model



[The OSGi Architecture]

OSGi - Components

- **Bundles** - Bundles are the OSGi components made by the developers.
- **Services** - The services layer connects bundles in a dynamic way by offering a publish-find-bind model for plain old Java objects.
- **Life-Cycle** - The API to install, start, stop, update, and uninstall bundles.
- **Modules** - The layer that defines how a bundle can import and export code.
- **Security** - The layer that handles the security aspects.
- **Execution Environment** - Defines what methods and classes are available in a specific platform.

[The OSGi Architecture]

OSGi - Bundles

Bundles are deployed on an OSGi *framework*, the bundle runtime environment. This is not a container like Java Application Servers.

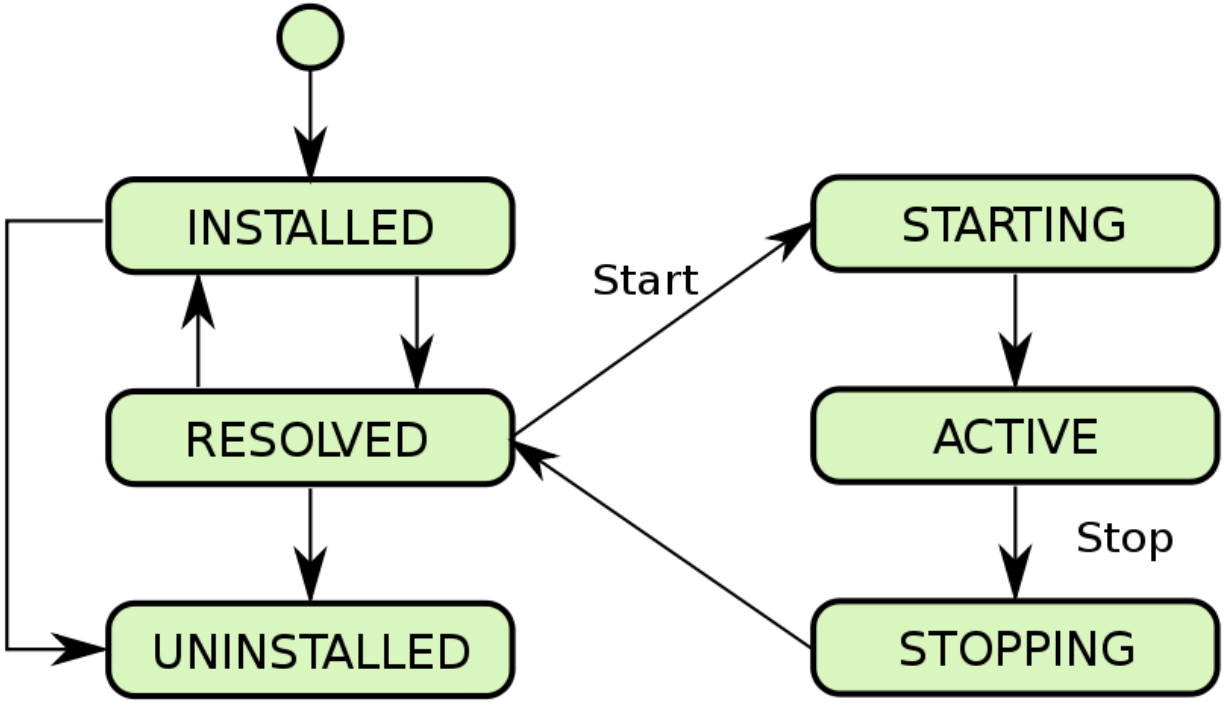
- It is a *collaborative environment*.
- Bundles run in the same VM and can actually share code.
- The framework uses the explicit imports and exports to wire up the bundles so they do not have to concern themselves with class loading.

Another contrast with the application servers is that the management of the framework is standardized.

A simple API allows bundles to install, start, stop, and update other bundles, as well as enumerating the bundles and their service usage.

[The OSGi Architecture]

OSGi – Bundle lifecycle



OSGi - the driving forces

- **Reduced Complexity** - Developing with OSGi technology means developing bundles: the OSGi components. Bundles are modules. They hide their internals from other bundles and communicate through well defined *services*. Hiding internals means more freedom to change later.

- **Reuse** - The OSGi component model makes it very easy to use many third party components in an application. An increasing number of open source projects provide their JARs ready made for OSGi.

- **Real World** - The OSGi framework is dynamic. It can update bundles on the fly and services can come and go. Developers used to more traditional Java see this as a very problematic feature and do not emphasize the advantages.

- **Easy Deployment** - The OSGi technology is not just a standard for components. It also specifies how components are installed and managed.

- **Dynamic Updates** - The OSGi component model is a dynamic model. Bundles can be installed, started, stopped, updated, and uninstalled without bringing down the whole system

[The OSGi Architecture]

OSGi - the driving forces

- **Adaptive** - The OSGi component model is designed from the ground up to allow the mixing and matching of components. This requires that the dependencies of components need to be specified and it requires components to live in an environment where their optional dependencies are not always available.

- **Transparency** - Bundles and services are first class citizens in the OSGi environment. The management API provides access to the internal state of a bundle as well as how it is connected to other bundles

- **Versioning** - OSGi technology solves JAR hell. JAR hell is the problem that library A works with library B;version=2, but library C can only work with B;version=3. In standard Java, you're out of luck. In the OSGi environment, all bundles are carefully versioned and only bundles that can collaborate are wired together in the same *class space*.

[The OSGi Architecture]

OSGi - the driving forces

- **Simple** - The OSGi API is surprisingly simple. The core API is only one package and less than 30 classes/interfaces. This core API is sufficient to write bundles, install them, start, stop, update, and uninstall them and includes all listener and security classes.

- **Small** - The OSGi Release 4 Framework can be implemented in about a 300KB JAR file. This is a small overhead for the amount of functionality that is added to an application by including OSGi

- **Fast** - One of the primary responsibilities of the OSGi framework is loading the classes from bundles. OSGi pre-wires bundles and knows for each bundle exactly which bundle provides the class. This lack of searching is a significant speed up factor at startup.

[The OSGi Architecture]

OSGi - the driving forces

- **Lazy** - Lazy in software is good and the OSGi technology has many mechanisms in place to do things only when they are really needed. For examples, bundles can be started eagerly, but they can also be configured to only start when another bundle is using them.

- **Secure** - Java has a very powerful fine grained security model at the bottom but it has turned out very hard to configure in practice. The result is that most secure Java applications are running with a binary choice: no security or very limited capabilities.

- **Humble** - Many frameworks take over the whole VM, they only allow one instance to run in a VM. OSGi is so flexible that one application server can easily host multiple OSGi frameworks.

[The OSGi Architecture]

OSGi - the driving forces

- **Non Intrusive** - Applications (bundles) in an OSGi environment are left to their own. They can use virtually any facility of the VM without the OSGi restricting them. there is no special interface required for OSGi services.

- **Runs Everywhere** - There are two issue to take care of. First, the OSGi APIs should not use classes that are not available on all environments. Second, a bundle should not start if it contains code that is not available in the execution environment. Both of these issues have been taken care of in the OSGi specifications.

- **Widely Used** - The OSGi specifications started out in the embedded home automation market but since 1998 they have been extensively used in many industries. Since 2003, the highly popular Eclipse Integrated Development Environment runs on OSGi technology.

[The OSGi Architecture]

OSGi – Example (activator)

```
package edu.tum.architecture.helloworld;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import edu.tum.architecture.helloworld.impl.HelloWorld;

public class Activator implements BundleActivator {
 private HelloWorld helloWorld;
 public void start(BundleContext context) throws Exception {
        helloWorld = new HelloWorld();
        helloWorld.start();
 }
 public void stop(BundleContext context) throws Exception {
        helloWorld.stopThread();
 }
}
```

Software architectures and their trade-offs

Service-oriented architectures

Motivation

Service oriented architectures solve specific problems developers face when building distributed systems.

- Multiple implementation languages
- Multi-governance
- Interoperability via standard
- Independent of vendor or technology
- Loosely coupled

Critics argue that SOA has a set of severe drawbacks that arise in every implementation.

- No service ecosystem
- Scalability
- Unfunctional browsing (UDDI, discovery)
- High complexity of ecosystem
- Reduce agility through hard-coding and dependencies

["Service-oriented architecture."]

Service-oriented architecture (SOA)

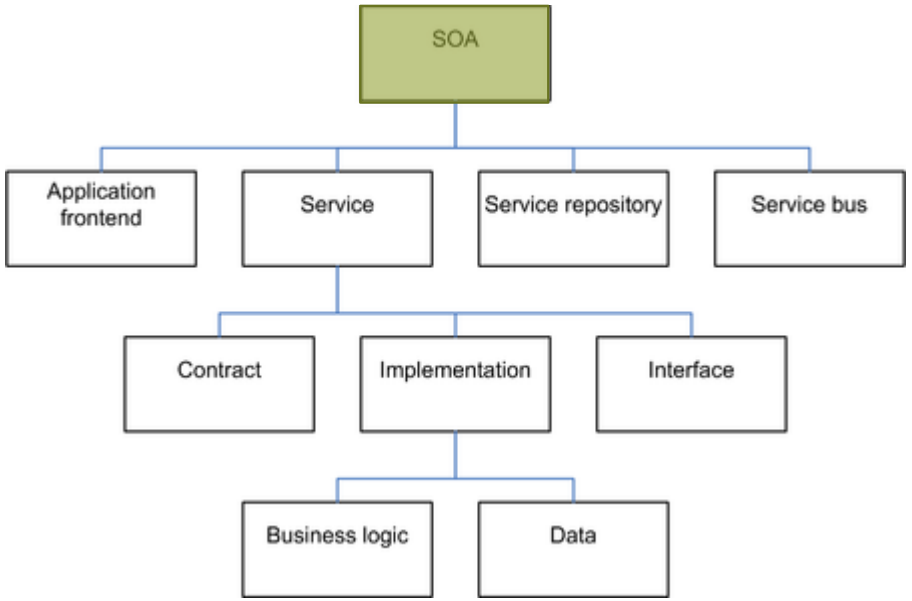
> “A Service-Oriented Architecture (SOA) is a software architecture that is based on the key concepts of an application frontend, service, service repository, and service bus. A service consists of a contract, one or more interfaces, and an implementation”

[“Enterprise SOA - Service-oriented architecture best practices.” Krafzig, D. et al. (2004)]

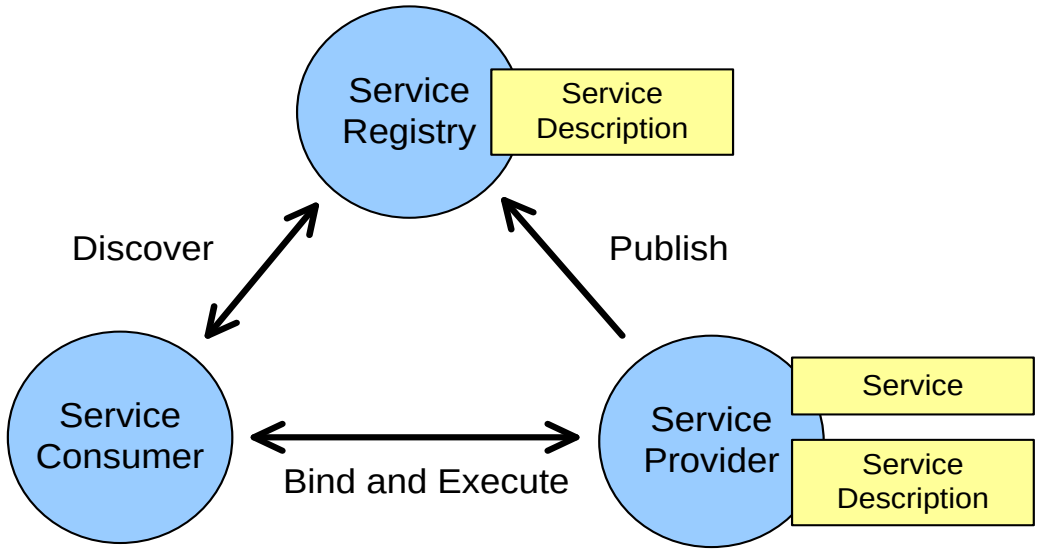
> “SOA is an architectural style that promotes business process orchestration of enterprise-level business services”

[“Dissecting service-oriented architectures.” Lublinsky, B. and Tyomkin, D. (2003)]

Service-oriented architecture (SOA)



["Enterprise SOA - Service-oriented architecture best practices." Krafzig, D. et al. (2004)]



A SOA has three involved roles: service **consumer**, service **provider**, and service **registry**.

The service consumer

- Is an application, a software module or another service that requires a service,
- Initiates the enquiry of the service in the registry, binds to the service over a transport medium, and executes the service function.

The service provider

- Is a network-addressable entity that accepts and executes requests from service consumers,
- Publishes its services and interface contract to the service registry so that the service consumer can discover and access the service.

A service registry

- Is the enabler for service discovery,
- Contains a repository of available services and allows for the lookup of service provider interfaces to interested service consumers.

Loose Coupling

Services are using each others functionality while still remaining independent.

Modularization

Support for distributed deployment.

Service Contract

A service in a SOA is described in a document which contains all information necessary for using it.

Discoverability

Of services at design time enables their reuse.

Abstraction and autonomy

A service hides implementation details and will be accessed only through its (standardized) interface.

Reusability

Services can be consumed by more than one consumers.

Composability

A coarse-grained service may orchestrate several services, which are of a finer granularity.

Stateless

Services should minimize the amount of state information they manage, as well as the duration for which they remain stateful.

Services

A service is a provider/client interaction that creates and captures value and provides some functionality over a *standardized interface*, which encapsulates the actual implementation.

A service can be divided into these parts: Service **description**, service **interface**, business **logic**, **implementation**, and **data**.

[“Enterprise SOA - Service-oriented architecture best practices.” Krafzig, D. et al. (2004)]

Web service

> “Self-contained, modular business applications that have open, Internet-oriented, standards-based interfaces”

[“UDDI Consortium: UDDI Executive White Paper.” (2001)]

> “A software application identified by a URI, whose interfaces and bindings are capable of being defined, described, and discovered as XML artifacts. A Web service supports direct interactions with other software agents using XML-based messages exchanged via Internet-based protocols.”

[“Web Services Architecture Requirements.” Daniel A. et al.]

There are two main styles of Web services:

- Simple Object Access Protocol (**SOAP**) **web services.**
- Representational State Transfer (**REST**) **web services**.

SOAP

Motivation

SOAP has been adopted widely due to its core capabilities.

- Foundation layer of web service stack
- Extensibility
- Technology agnostic (especially the transport protocol)
- Multi-language (independent of base language)

In combination with WS this leads to

- Evolution friendliness

On the other hand SOAP still has a few disadvantages.

- Big (bloated) XML messages
- Extensibility leads to fragmentation of capabilities
- Verbose messages and conversations

The critique in combination with WS is often that

- It's too complex
- Typically used in industry only

[Web Service] [SOAP]

Web services

"A standardized way of integrating Web-based applications using the XML, SOAP, WSDL, and UDDI *open standards* over an *Internet protocol* backbone. XML is used to tag the data, SOAP is used to transfer the data, WSDL is used for describing the services available, and UDDI is used for listing what services are available."

[Webopedia]

Web service technologies

Service Description

- Common base language: XML
- Interfaces: WSDL
- Business protocols: BPEL (WSCL)

Service Interactions

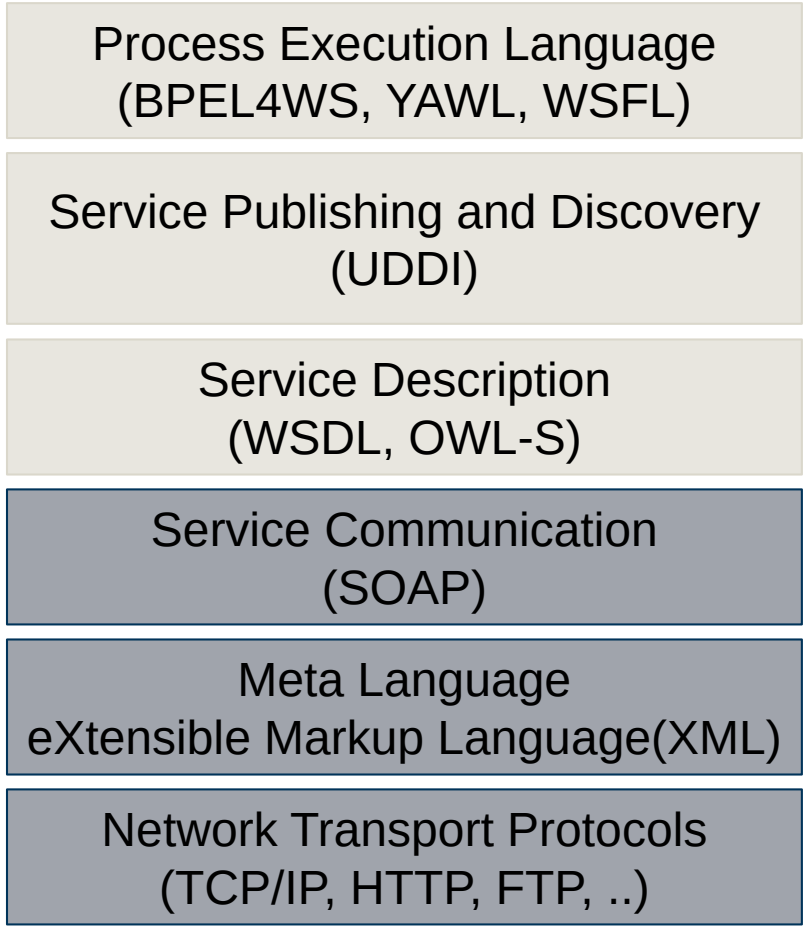
- Messaging: SOAP
- Transport, e.g., HTTP

Service Discovery

- Service descriptions are stored in a service directory (UDDI).
- Can be hosted and managed by a trusted entity or otherwise each company can host and manage a directory service.
- Can be done both at design-time by browsing the directory and identifying the most relevant services, and at run-time, using dynamic binding techniques.

Web services technology stack

- The basic layers (HTTP – WSDL) are well established.
- Higher layers are part of the overall architecture, but those layers have never been adopted at scale.
- Further concepts:
  - Semantics
  - Security
  - Transaction management
  - Quality of service management

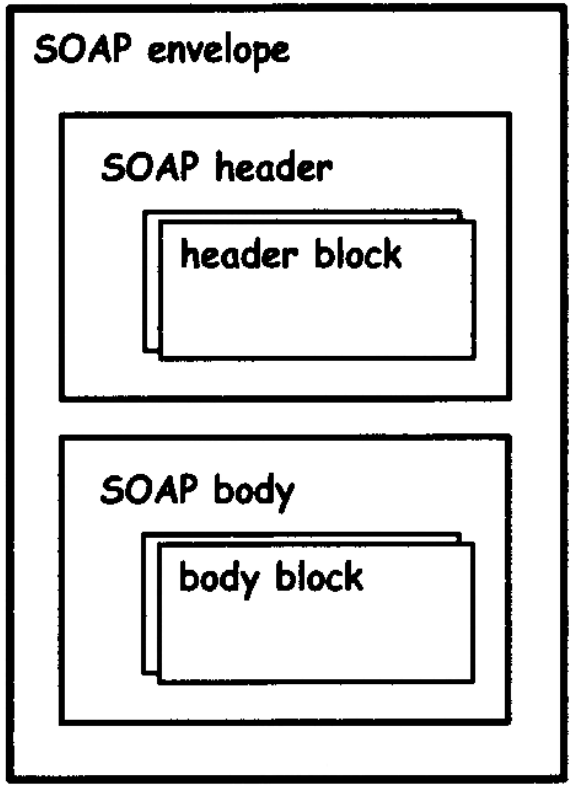


SOAP

- Originally, SOAP was an acronym for Simple Object Access Protocol, which was dropped with version 1.2.
- Is standardized by the W3C, current version: 1.2 (2007).
- Is the protocol that underlies all interactions among Web services.
- Defines how to use XML in a structured and typed manner, so that it can be exchanged between peers.
- Is stateless and one-way.
- Ignores the semantics of the messages being exchanged through it.

Structure and contents of a SOAP message

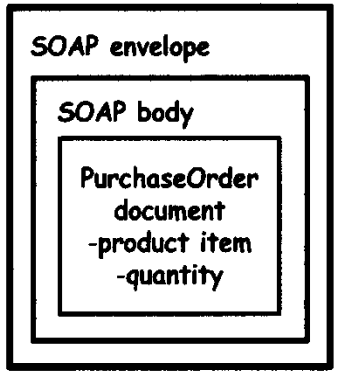
- SOAP exchanges information using messages.
- These messages are used as an **envelope** where the application encloses whatever information needs to be sent.
- Each envelope contains two parts: a header and a body.
- The header is optional, the body is mandatory, i.e., all SOAP messages must have a body.
- SOAP does not **require** any further structure within the content placed in header or body blocks.
- **Typically**, there are two aspects that influence how the header and body of a SOAP message are constructed:
  1. Interaction style
  2. Encoding style.

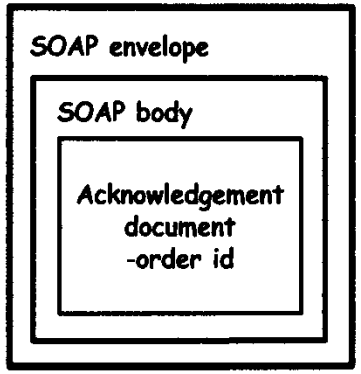


Interaction styles (1)

There are two interaction styles:

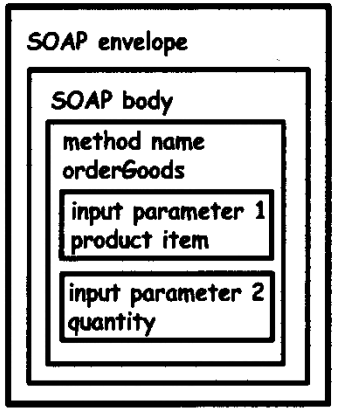
1. In **document-style** interaction the two interacting applications agree upon the structure of (often hierarchical) documents exchanged between them.

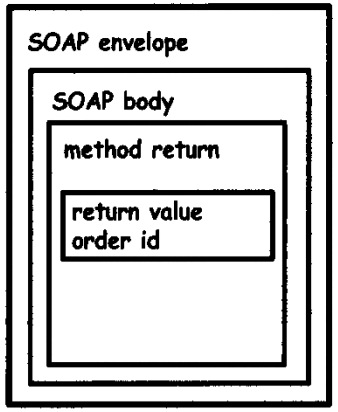




2. In **RPC-style** interaction, one SOAP message encapsulates the request while another message encapsulates the response.

   The two interacting applications have to agree upon the RPC method signature as apposed to the document structure.





Interaction styles (2)

Whether an interaction is synchronous or asynchronous is orthogonal to the interaction style and does not impact the structure of the exchanged SOAP messages.

Asynchronous communication: (Client Code)

```
while (...) {
        purchaseOrderDocument = ...;
        send(purchaseOrderDocument);
}
```

Synchronous communication: (ClientCode)

```
while (...) {
        purchaseOrderDocument = ...;
        send(purchaseOrderDocument);
        acknowledgeDocument = receive(...);
}
```

Encoding

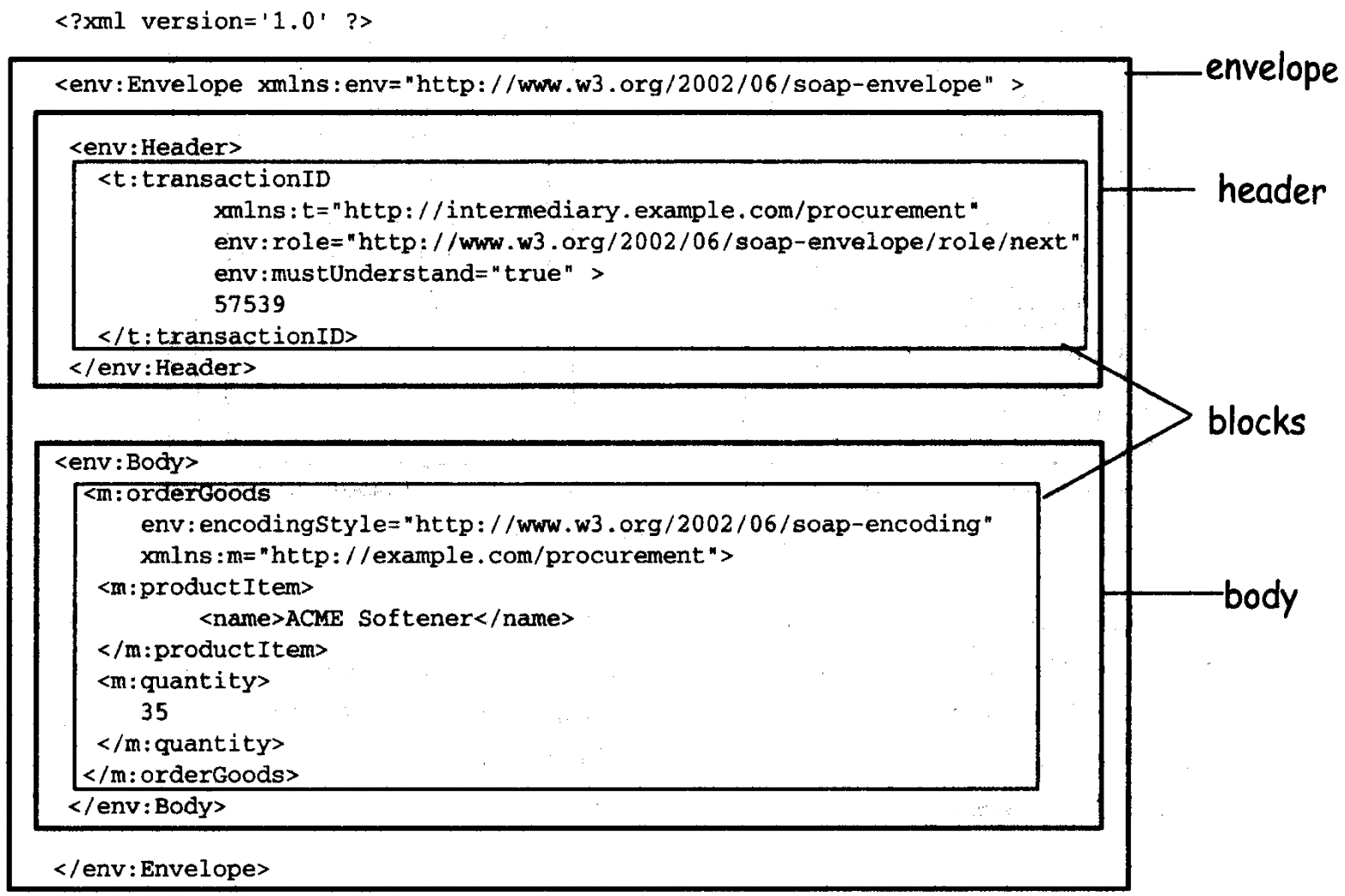
The structure of a SOAP message is also influenced by **encoding rules**, which define how a particular entity or data structure is represented in XML.

```
<ProductItem>
  <name>…</name>
  <type>…</ty<pe>
  <make>…</make>
</ProductItem>
```

```
<ProductItem
  <name="…"
   type="…"
   make="…"
/>
```

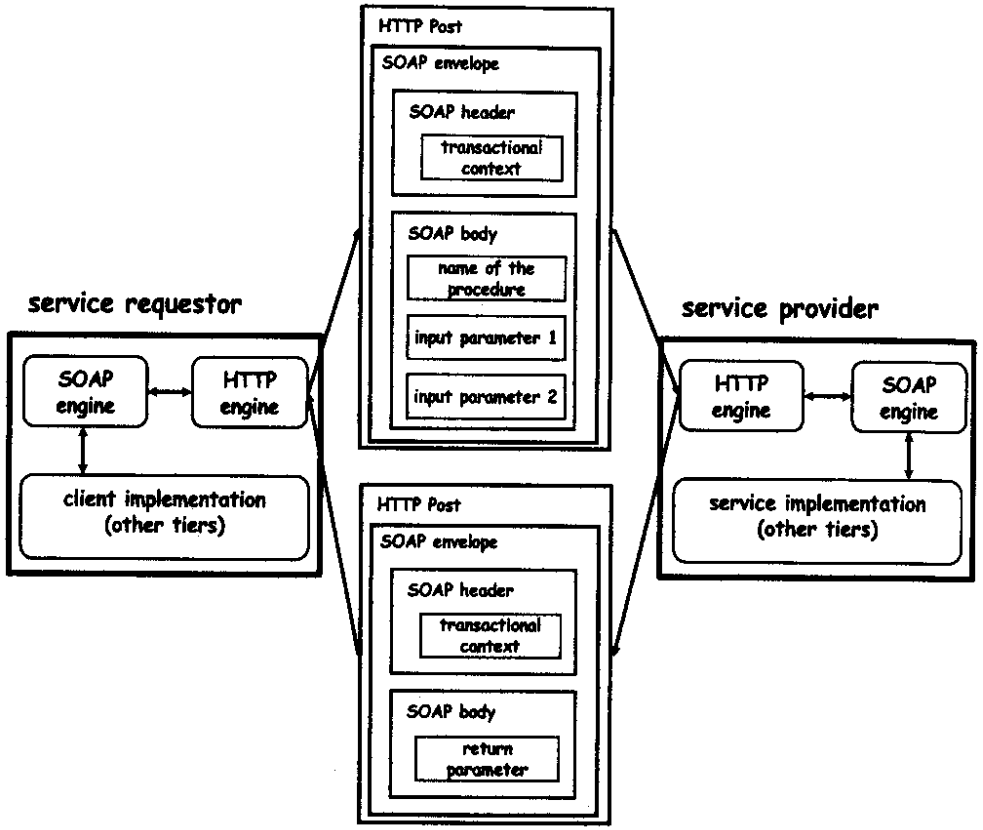
```
<ProductItem
name="…">
  <type>…</type>
  <make>…</make>
</ProductItem>
```

Example of a SOAP message



Binding SOAP to a transport protocol

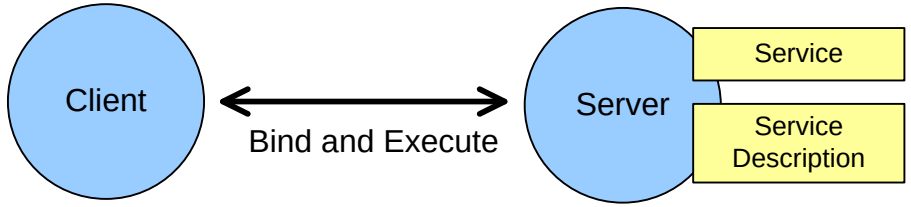
SOAP does not impose any transport protocol. Typically it is associated with HTTP. The specification of which protocol to use is called a **binding:**



A **binding** defines how a message is wrapped within a transport protocol and how the message has to be treated using the primitives of the transport protocol used.

WSDL – Web Service Description Language

WSDL has a role and purpose similar to that of IDLs in conventional middleware.



In WSDL, specifications are XML documents that describe web service interfaces.

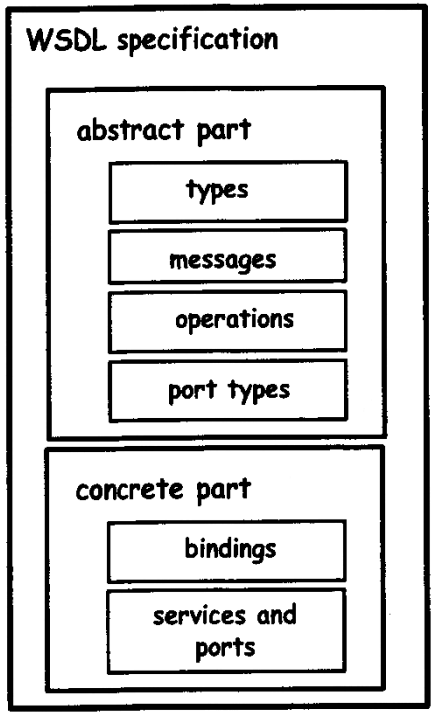
A significant difference with respect to conventional middleware is that, in addition to specifying the operations offered by a service, WSDL also needs to define the mechanisms to access the Web service (transport details: HTTP, FTP, SMTP).

- The location at which the service is available also has to be defined.
- The separation of interfaces and bindings to protocols makes the case for modular specifications.

Structure of a WSDL interface

- WSDL specifications are often characterized by an **abstract part**, conceptually analogous to conventional IDL, and a **concrete part**, that defines protocol binding and other information.

- The abstract part is made of **port type** definitions, which are analogous to interfaces in traditional middleware IDLs.
- Each port type is a logical collection of related **operations.**
- Each operation defines a simple exchange of **messages.**

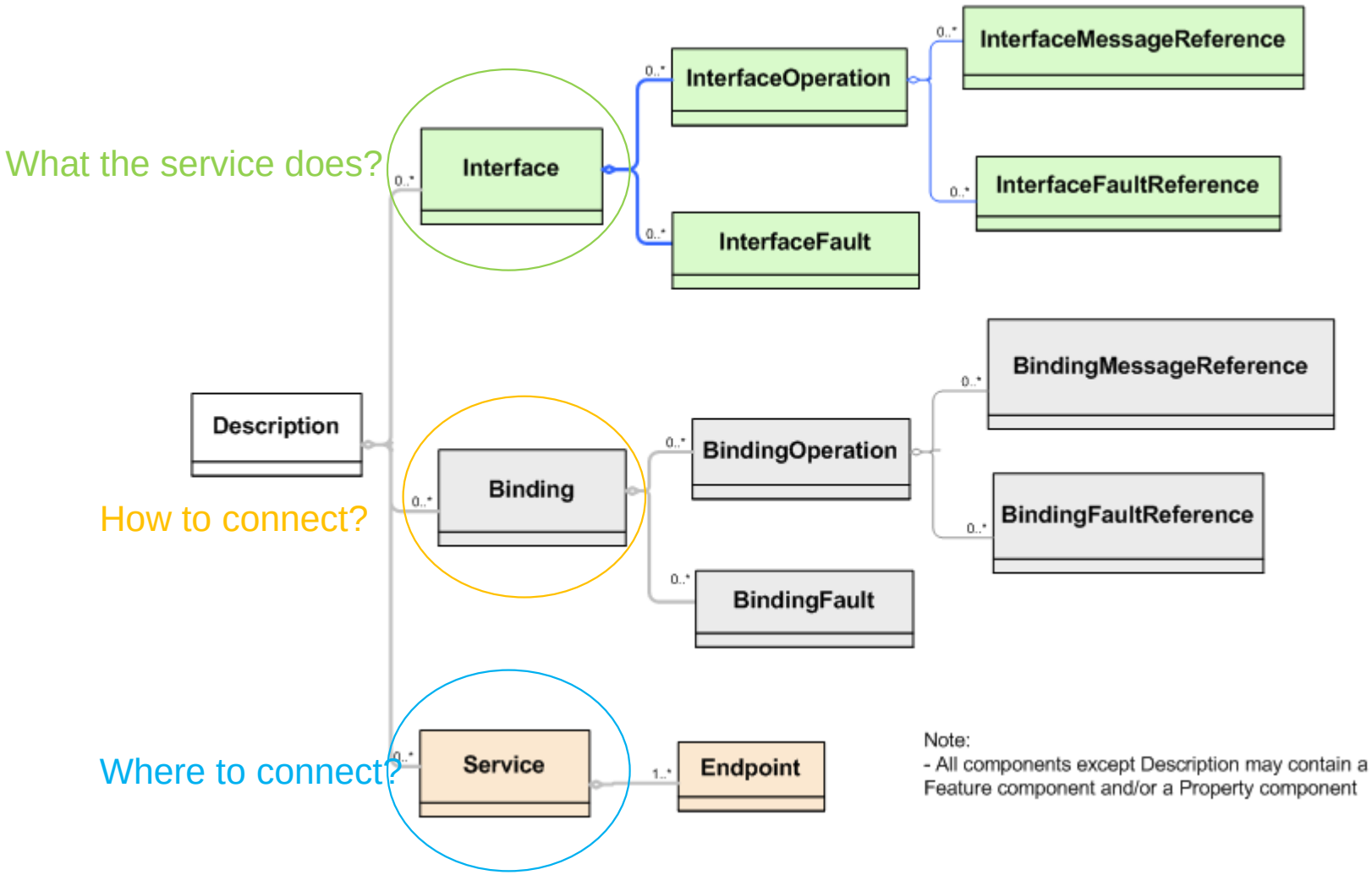


- A large number of XML technologies and standards are required to precisely specify the encoding style, the protocol style, bindings, ...

➔ WSDL specifications are not targeted at humans

➔ Interoperability of services requires additional conventions or standards to reduce the protocol design space opened by WSDL. (see, e.g. WS-I, http://www.ws-i.org/)

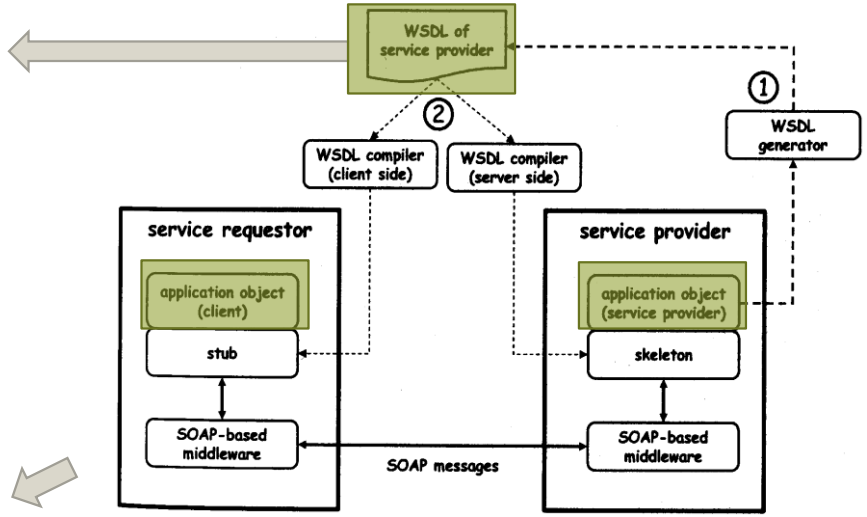
WSDL 2.0 component model



[Web Services Description Language (WSDL) v2.0]

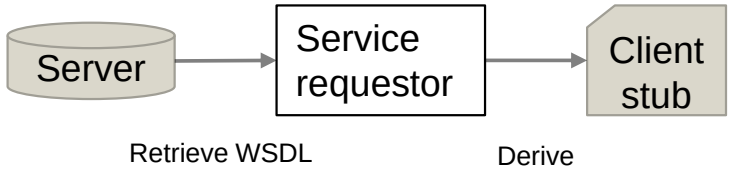
Using WSDL

```
<wsdl2:interface name=`ServiceInterface`>
        <wsdl2:operation
            name=`sayHello`
            style=`http://www.w3.org/ns/wsdl/style/rpc`
            pattern=`http://www.w3.org/ns/wsdl/in-out`>
                    <wsdl2:input
                            element=`ns:sayHello`
                            wsaw:Action=`urn:sayHello` />
                    <wsdl2:output
                            element=`ns:sayHelloResponse`
                            wsaw:Action=`urn:sayHelloResponse`/>
        </wsdl2:operation>
</wsdl2:interface>
```




```
try {
    HelloworldWSDL2Stub stub = new HelloworldWSDL2Stub();
    HelloworldWSDL2Stub.SayHello request = new  HelloworldWSDL2Stub.SayHello();
    request.setName("World");
    HelloworldWSDL2Stub.SayHelloResponse response = stub.sayHello(request);
    System.out.println(response.get())
} catch (Exception e) {};
```

```
public demo.SayHelloResponse sayHello(SayHello sayHello)
throws HelloworldWSDL2ExceptionException {
      //business logic goes here!
      throw new java.lang.UnsupportedOperationException(
          "Please implement " + "#sayHello");
};
```



SOA governance

- Governance plays an important role in adopting and managing an SOA.
- SOA governance is important at three different levels [Kel07]:
  - At the **strategic** level the management of a company defines which role SOA should play.
  - At the **operational** level decisions, which may cross departments, but do not have influence on the whole organization are made:
    - Who is the owner of a service?
    - Who pays for implementation and maintenance of a service?
    - Which non-functional requirements have to be fulfilled by a service, e.g., availability, performance?
  - At the **technical** level tools help to ensure technical integrity.

REST is a set of rules, the most important address the following NFRs:

- Performance of component interactions
- Scalability of the ecosystem
- Simplicity of interfaces
- Visibility
- Language agnostic
- URIs
- Designed for the Internet

Using REST has its downsides though:

- Not the silver bullet it is currently perceived as
- Bloated protocol (HTTP)
- HTTP is not RPC
- No semantic descriptions

[“Architectural styles and the design of network-based software architectures.” Fielding, R. (2000)]

REST web services

- Representational State Transfer (**REST**) is a software architectural style for distributed hypermedia systems like the world wide web.
- The term has been coined by Roy Fielding in his doctoral dissertation
- REST provides a set of architectural constraints that, when applied as a whole, emphasizes
  - Scalability of component interactions,
  - Generality of interfaces,
  - Independent deployment of components, and
  - Intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems.
- The REST architectural style has been used to guide the design and development of the architecture for the modern Web.
- The abstract discussion about architectural styles enables judgments over whether particular practices are consistent with the architecture of the Web.

[“Architectural styles and the design of network-based software architectures.” Fielding, R. (2000)]

The Null style

- Is simply an empty set of constraints
- Describes a system in which there are no distinguished boundaries between components
- Is the starting point for the description of REST
- Examples: Mainframe application, Desktop application, “Closed” Distributed System (e.g. World of Warcraft)

[“Architectural styles and the design of network-based software architectures.” Fielding, R. (2000)]

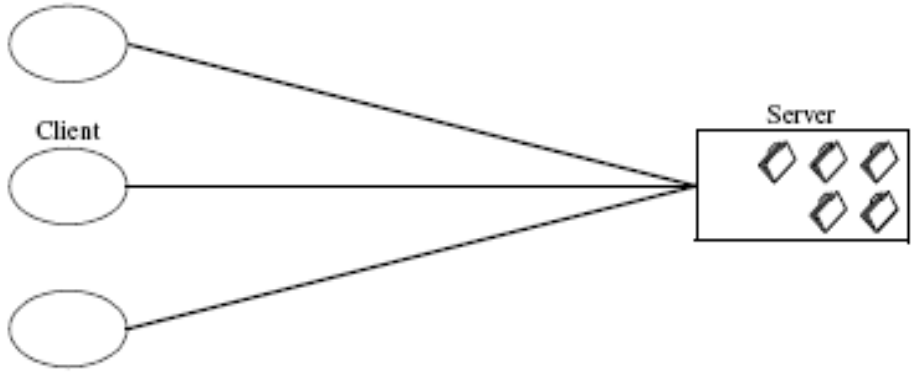
Client-server

- Separate the user interface concerns from the data storage concerns.
- Improves the portability of the user interface across multiple platforms.
- Improves scalability by simplifying the server components.
- Allows the components to evolve independently.

[“Architectural styles and the design of network-based software architectures.” Fielding, R. (2000)]

Stateless

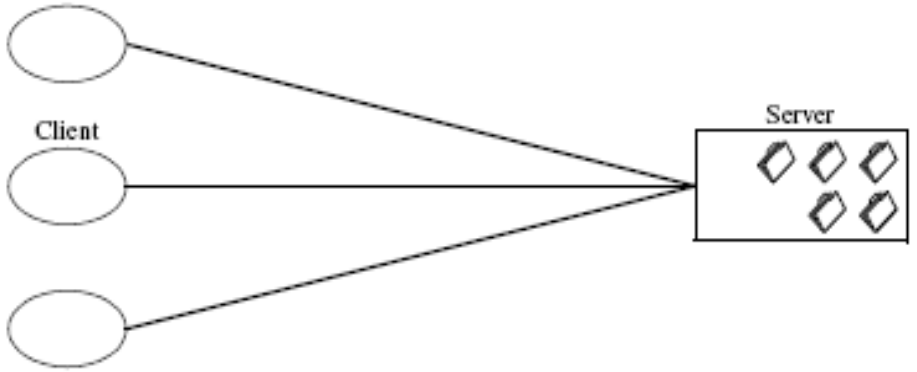
- Communication must be stateless in nature: each request from client to server must contain all of the information necessary to understand the request.
- Visibility is improved because a monitoring system does not have to look beyond a single request datum in order to determine the full nature of the request.
- Reliability is improved because it eases the task of recovering from partial failures.



[“Architectural styles and the design of network-based software architectures.” Fielding, R. (2000)]

Stateless

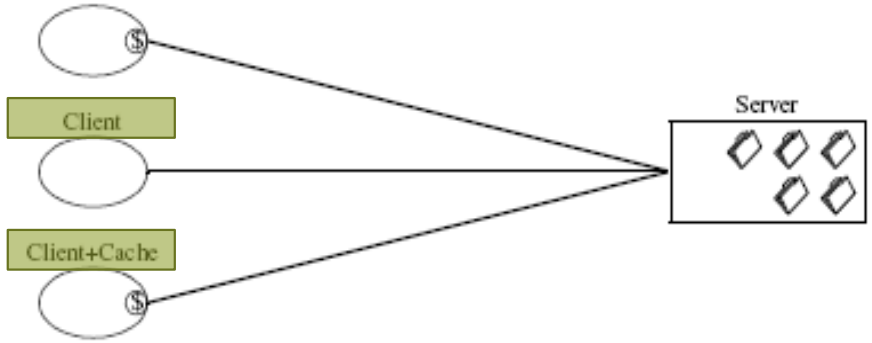
- Scalability is improved because not having to store state between requests allows the server component to quickly free resources, and further simplifies implementation because the server doesn't have to manage resource usage across requests.
- Disadvantage: it may decrease network performance by increasing the repetitive data sent in a series of requests, since that data cannot be left on the server in a shared context



["Architectural styles and the design of network-based software architectures." Fielding, R. (2000)]

Cacheable

- Require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable.
- Improves efficiency, scalability, and user-perceived performance by reducing the average latency of a series of interactions.
- Trade-off: a cache can decrease reliability if stale data within the cache differs significantly from the data that would have been obtained had the request been sent directly to the server.

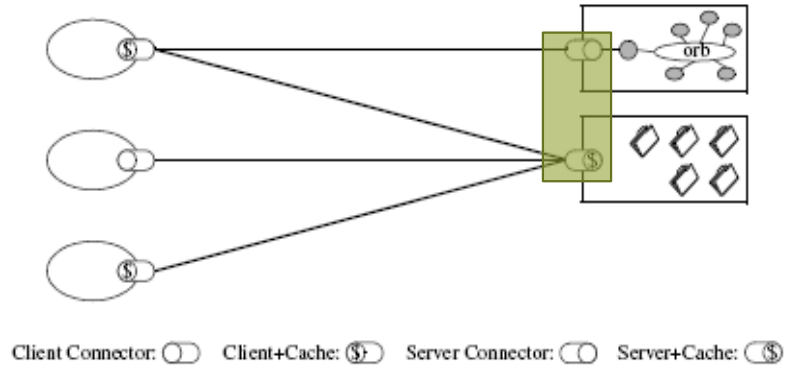


["Architectural styles and the design of network-based software architectures." Fielding, R. (2000)]

Uniform interfaces

- Emphasis on a uniform interface between components:
  - Uniform identification scheme,
  - Uniform representation of information exchanged.
- Implementations are decoupled from the services they provide.
- Trade-off: information is transferred in a standardized form rather than one which is specific to an application's needs.

This is the central feature that distinguishes the REST architectural style from other network-based styles.



["Architectural styles and the design of network-based software architectures." Fielding, R. (2000)]

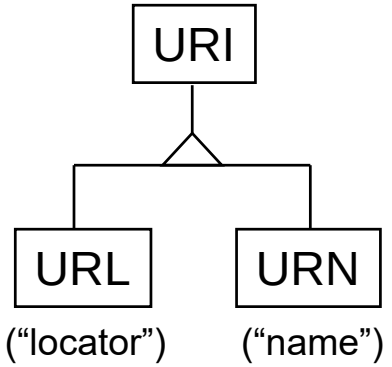
Uniform representation of information exchanged

- The type of a representation is specified in the HTTP header **`Content-Type`** of the response.
- Content types are identified using MIME (Multipurpose Internet Mail Extension) types.
- A MIME type contains a type and a subtype.
- Types are generic and predefined, e.g., **`text`**, **`image`**, **`audio`**, **`video`**, and **`application.`**
- Subtypes provide more information about the content, e.g., **`text/plain`**, **`text/html`**, **`image/jpeg.`**
- The Web uses standardized (often human-readable) exchange content formats, in particular HTML and XML.
- Contrast with “optimized” record layout & binary representations.

[“Architectural styles and the design of network-based software architectures.” Fielding, R. (2000)]

Uniform identification scheme

- URI: Uniform Resource Identifier (general term)
  - There are two mechanisms: naming and location
- URN: Uniform Resource Name
  - Identification of objects by name for the purpose of persistent labeling
  - E.g., ISBN
- URL: Uniform Resource Locator
  - Identification via the primary access mechanism



- An example of an URI:

scheme		authority	path		query		fragment
http	://	example.org	/mysite/page	?	name=cat	#	whiskers

- A URI points to a hierarchical space reading from left to right; each block that follows is a branch from the previous block.

[“Architectural styles and the design of network-based software architectures.” Fielding, R. (2000)]

The URI dissected

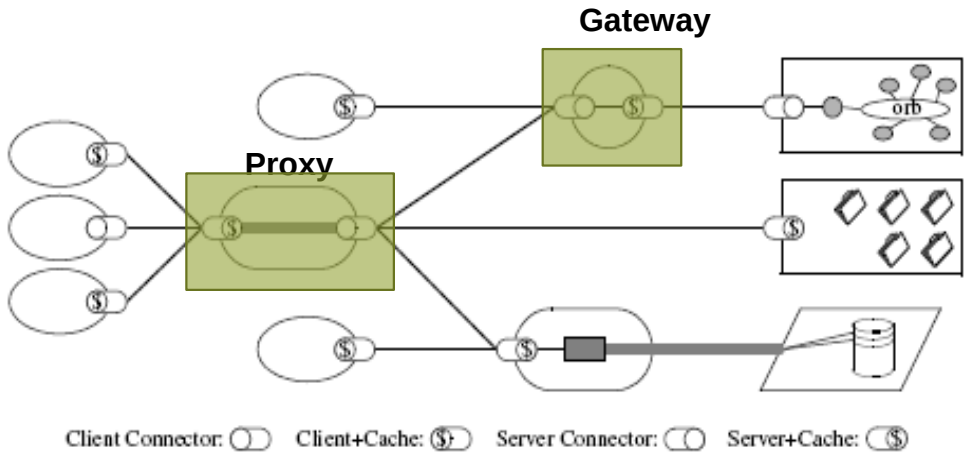
- **Scheme** – Defines how the URI should be interpreted.
- **Authority** – Has the structure userinfo@host:port
- **Path** – Looks like the path on a file system and is often used in a hierarchical fashion to address files on a system.
  - Example: http://del.iciou.us/danja/owl refers to all items tagged by danja with owl.
- **Query** – The spec describes the query part as being a non-hierarchical part of the URI.
- **Fragment** – Is used to identify a secondary resource.

- URIs are not being used uniformly on the Web, but: when a URI has been used to identify a resource, it should continue to be used to identify the same resource.

["Architectural styles and the design of network-based software architectures." Fielding, R. (2000)]

Layered system

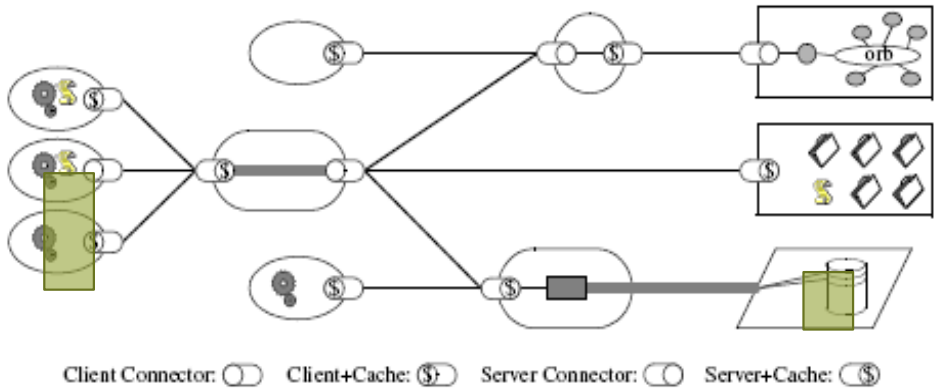
- Allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot “see” beyond the immediate layer with which they are interacting.
- Disadvantage: adds overhead and latency to the processing of data.



[“Architectural styles and the design of network-based software architectures.” Fielding, R. (2000)]

Code-on-demand

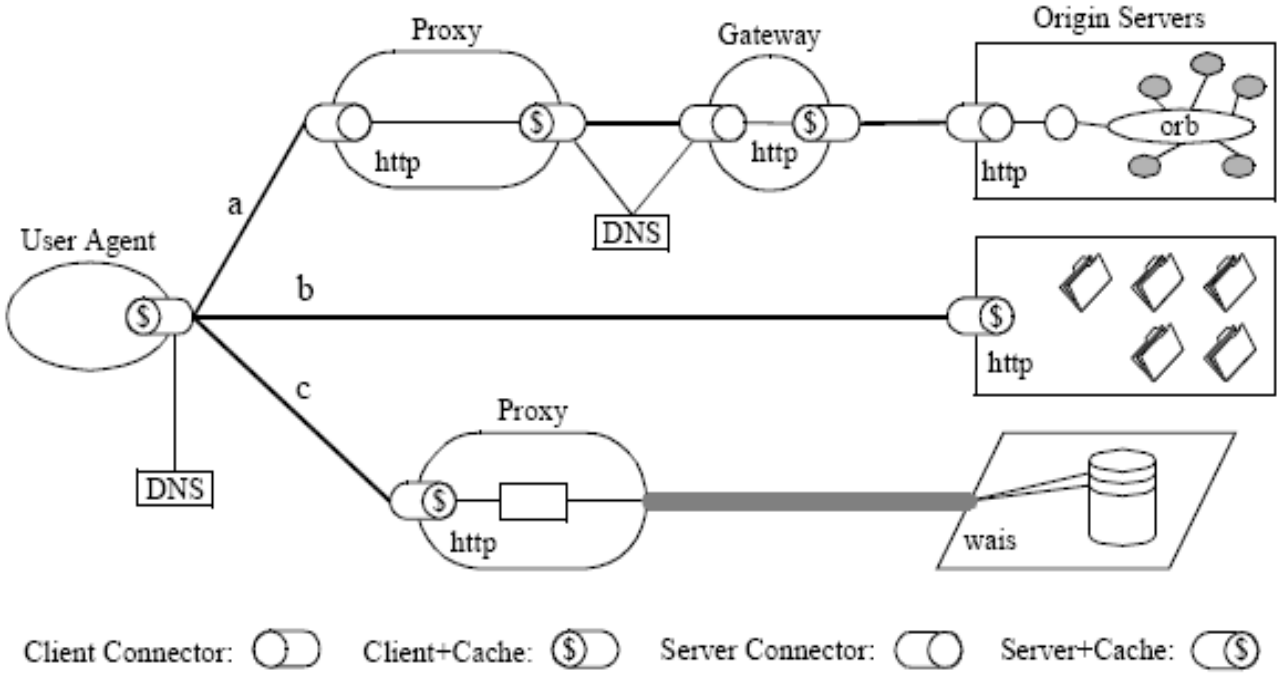
- Allows client functionality to be extended by downloading and executing code in the form of applets or scripts.
- Simplifies clients by reducing the number of features required to be pre-implemented.
- Allowing features to be downloaded after deployment improves system extensibility.
- Is an *optional* constraint within REST.



["Architectural styles and the design of network-based software architectures." Fielding, R. (2000)]

Process view of a REST-based architecture

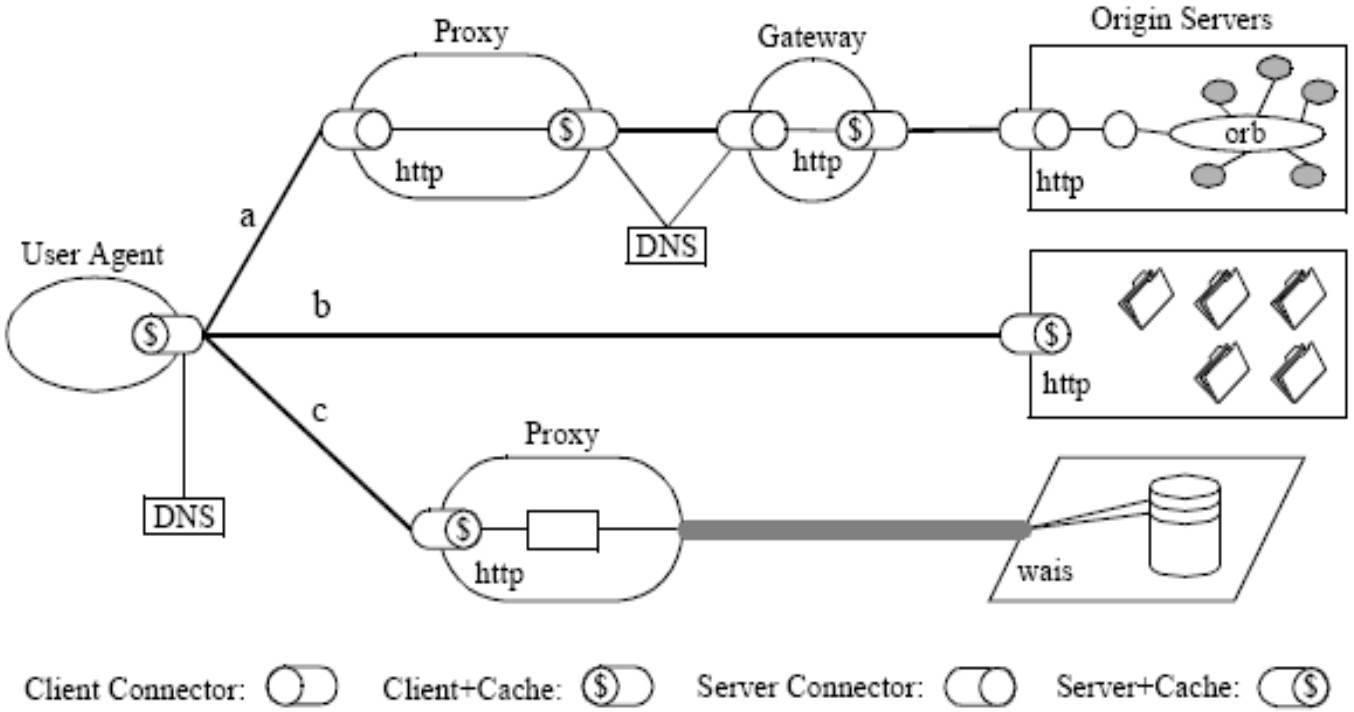
- Request (a) has been sent to a local proxy, which in turn accesses a caching gateway found by DNS lookup, which forwards the request on to be satisfied by an origin server whose internal resources are defined by an encapsulated object request broker architecture.
- Request (b) is sent directly to an origin server, which is able to satisfy the request from its own cache.



["Architectural styles and the design of network-based software architectures." Fielding, R. (2000)]

Process view of a REST-based architecture

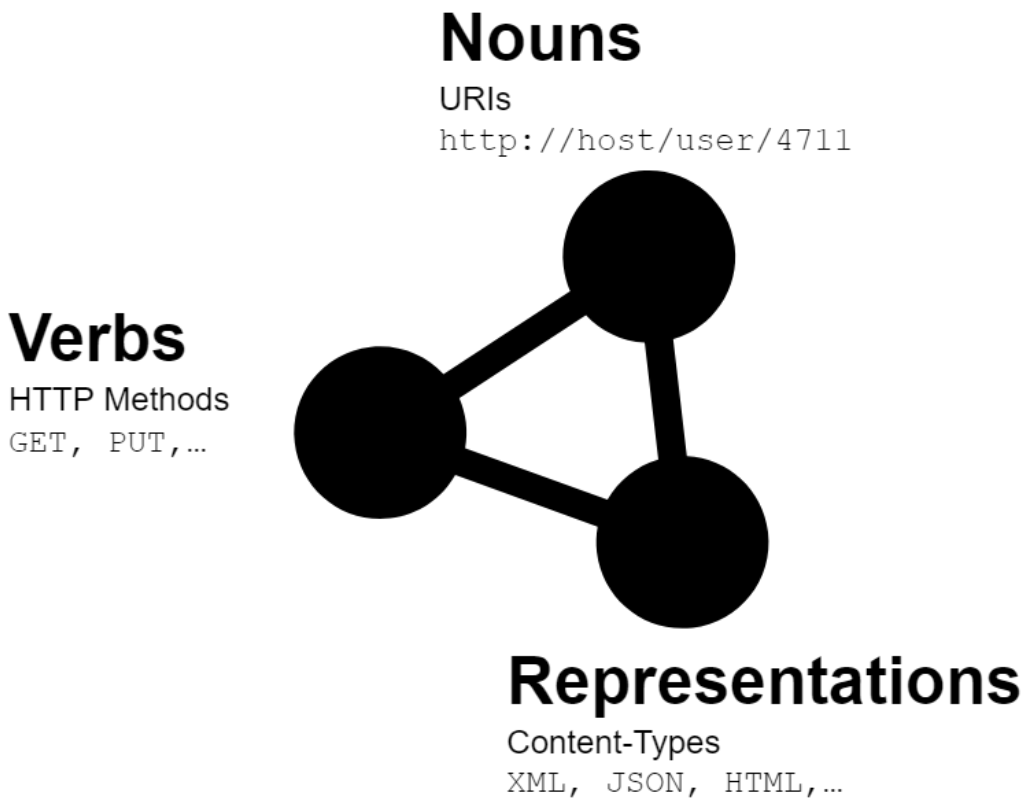
- Request (c) is sent to a proxy that is capable of directly accessing WAIS, an information service that is separate from the Web architecture, and translating the WAIS response into a format recognized by the generic connector interface.
- Each component is only aware of the interaction with their own client or server connectors; the overall process topology is an artifact of our view.



["Architectural styles and the design of network-based software architectures." Fielding, R. (2000)]

REST triangle

- The main problem domains identified in REST are the **nouns**, the **verbs**, and the **content-type** spaces.
- The *things that exist*, the *things you can do to them*, and the *information you can transfer* as part of any particular operation.
- REST requires a **standardized** set of state transfer operations.



[REST triangle]

[“Architectural styles and the design of network-based software architectures.” Fielding, R. (2000)]

REST methods

Minimum methods: GET, PUT, POST, DELETE

- GET is the HTTP equivalent of COPY
  - Transfers a representation from resource to client.
- PUT is the HTTP equivalent of PASTE OVER
  - Transfers state from a client to a resource.
  - GET and PUT are fine for transferring state of existing resources.
- POST is the PASTE AFTER verb
  - Don't overwrite what you currently have: Add to it
  - Create a resource.
  - Add to a resource.
- DELETE is the HTTP equivalent of CUT
  - Requests the resource state being destroyed.

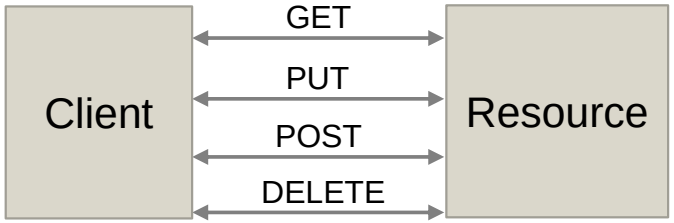
REST vs arbitrary interfaces

RESTful Interface

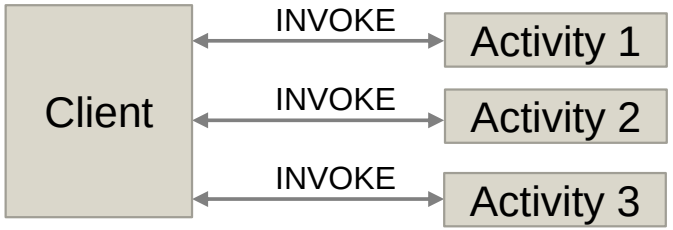
- Stateless
- Uniform interface
- Based on HTTP
- Simple, lightweight
- No XML overhead
- Resource-oriented

Arbitrary Interface

- Stateless or stateful
- Non-uniform interface
- Based on SOAP or HTTP
- Complex, powerful
- XML overhead
- Activity-oriented



Resource-oriented services



Activity-oriented services

```
http://example.com/users/ [GET, POST]
http://example.com/users/{Id} [GET, PUT, DELETE]
```

```
getUsers()
getUser(id)
addUser(user)
removeUser(id)
updateUser(user)
```

Web service composition

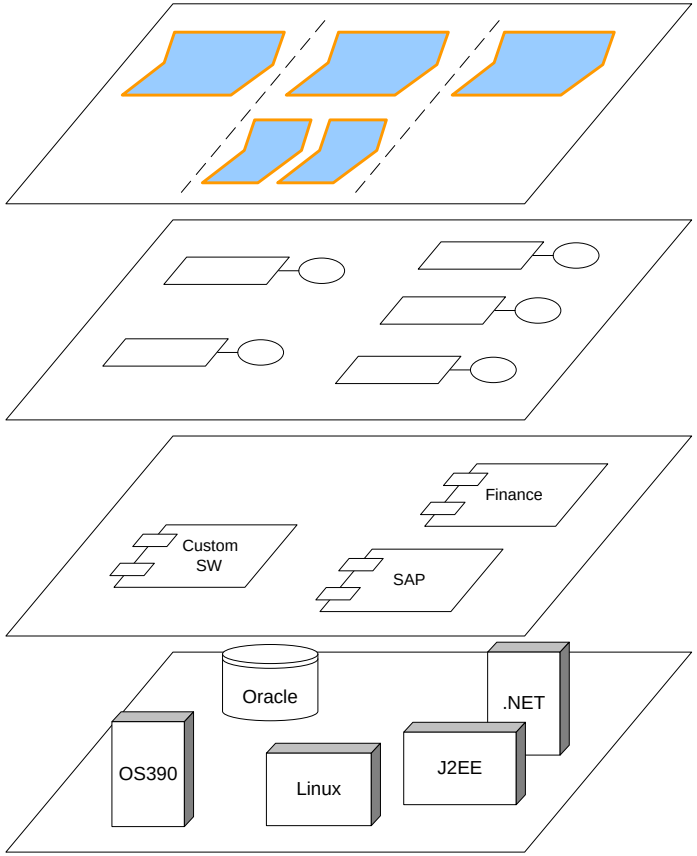
Alignment of Business and IT

Processes

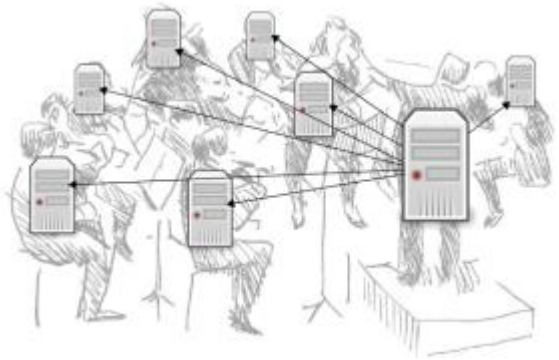
Services

Applications

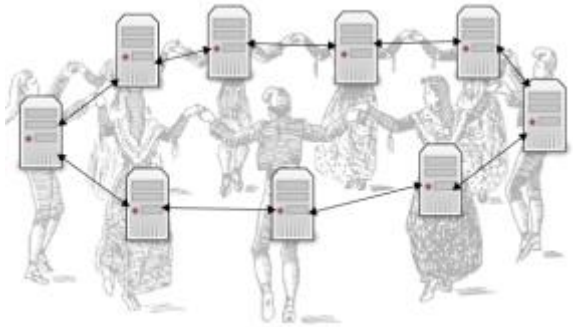
Technology



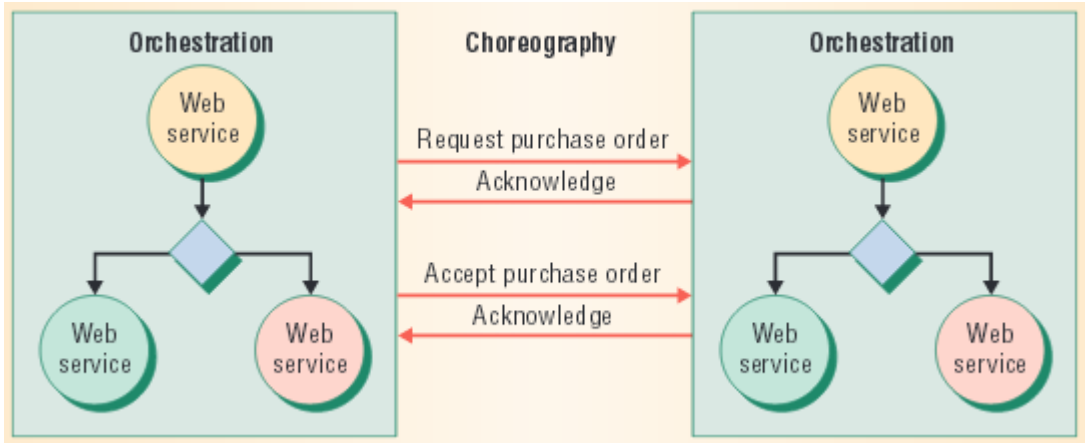
Web service composition



Orchestration



Choreography



Orchestration and Choreography

["Web services orchestration and choreography." Peltz, Chris. (2003)]

Software architectures and their trade-offs

Microservice architecture

The term “Microservice Architecture” has sprung up over the last few years to describe a particular way of designing software applications as suites of independently deployable services. While there is no precise definition of this architectural style, there are certain common characteristics around organization, business capability, automated deployment, intelligence in the endpoints, and decentralized control of languages and data.

[Microservices. Martin Fowler (2015)]

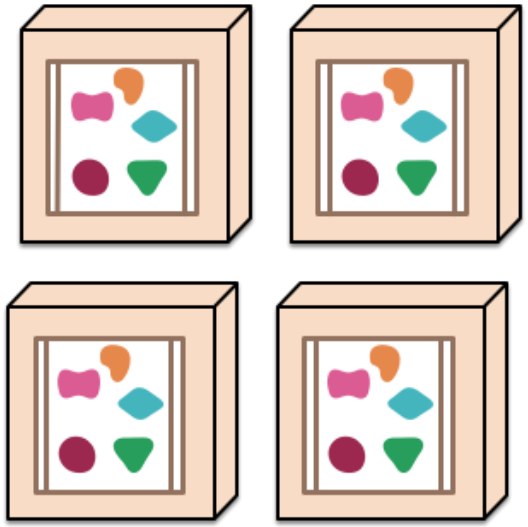
“is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.”

[M. Fowler and J. Lewis, 2014]

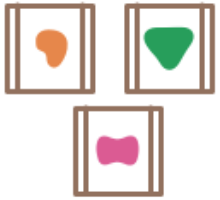
Monoliths and microservices

A monolithic application puts all its functionality into a single process...

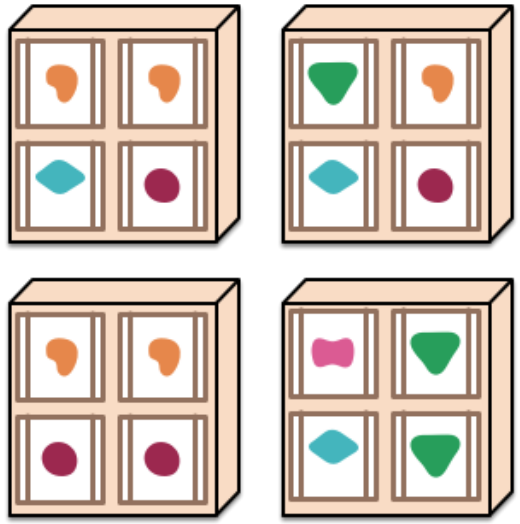
... and scales by replicating the monolith on multiple servers



A microservices architecture puts each element of functionality into a separate service...



... and scales by distributing these services across servers, replicating as needed.



[Microservices. Martin Fowler (2015)]

Characteristics of microservices

- **Componentization via services**
  - A **component** is a unit of software that is independently replaceable and upgradeable. Organized around Business Capabilities
  - The primary way of componentizing is by breaking down software into services
  - **Services** are out-of-process components who communicate with a mechanism such as a web service request, or remote procedure call.
  - Services are independently deployable
  - More explicit component interface
  - A service may consist of multiple processes that will always be developed and deployed together, such as an application process and a database that's only used by that service.

Problem: Remote calls are more expensive than in-process calls, and thus remote APIs need to be coarser-grained

[Microservices. Martin Fowler (2015)]

Characteristics of microservices

Products not projects

- A team should own a product over its full lifetime
- A common inspiration for this is Amazon's notion of "you build, you run it" where a development team takes full responsibility for the software in production: DevOps
- Day-to-Day contact with how software behaves in production and increases contact with users

Smart endpoints and dumb pipes

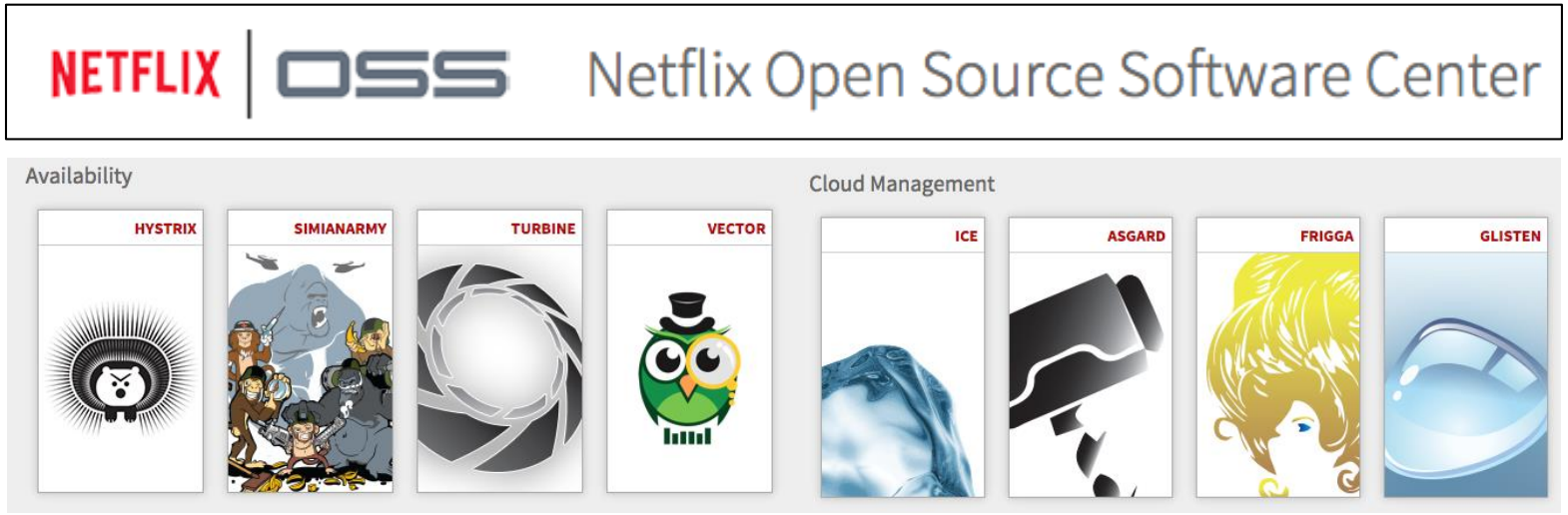
- As decoupled and as cohesive as possible – MS own their own domain logic and act more as filters in the classical Unix sense
- The two protocols used most commonly are HTTP request-response with resource API's and lightweight messaging
- Another approach: messaging over a lightweight message bus. The infrastructure chosen is typically dumb (dumb as in acts as a message router only) - simple implementations such as RabbitMQ or ZeroMQ

[Microservices. Martin Fowler (2015)]

Characteristics of microservices

Decentralized governance

- Centralized governance leads to standardization on single technology platforms. Experience shows that this approach is constraining - not every problem is a nail and not every solution a hammer.
- Rather than use a set of defined standards written down somewhere on paper; with microservices, we prefer the idea of producing useful tools that other developers can use to solve similar problems to the ones they are facing.

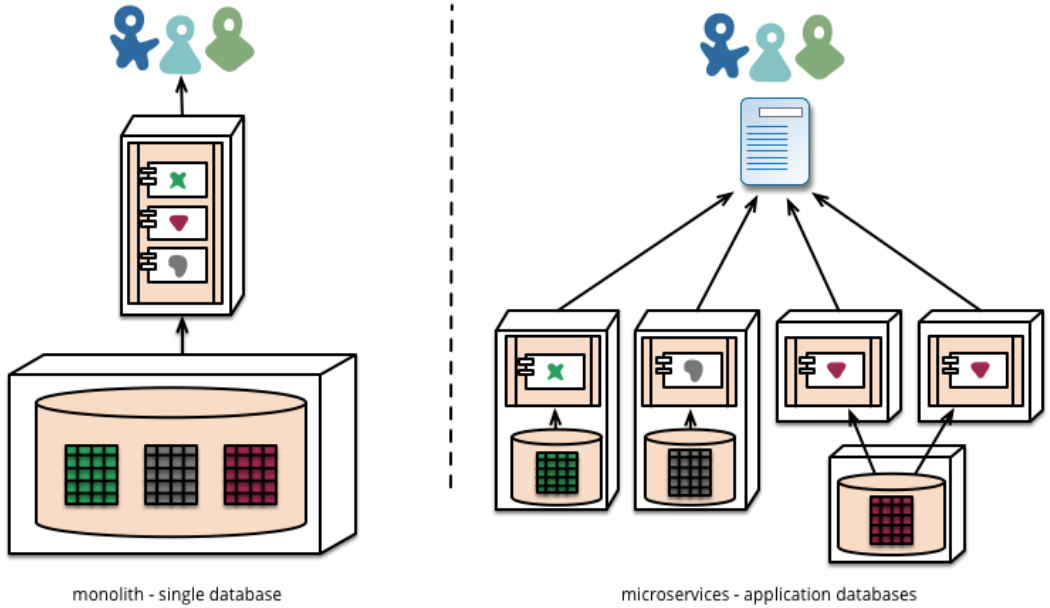


[Microservices. Martin Fowler (2015)]

Characteristics of microservices

Decentralized data management

- Conceptual model of the world will differ between **systems (common issue when integrating across a large enterprise)**
- Decentralize data storage decisions
- Each service manages its own database, either different instances of the same database technology, or entirely different database systems
- Dealing with updates: Use transactions to guarantee consistency when updating multiple resources
- **Problem: significant temporal coupling**



[Microservices. Martin Fowler (2015)]

Characteristics of microservices

Infrastructure automation

- Lots of **automated tests**
- **Automate deployment** to each new environment
- There isn't that much difference between monoliths and microservices
- Extensive infrastructure automation in production

Design for failure

- Any service call could fail due to unavailability of the supplier
- Choreography and event collaboration key. This leads to emergent behavior.
- Constantly reflect on how service failures affect the user experience

> Example: Netflix's Simian Army induces failures of services and even datacenters during the working day to test both the application's resilience and monitoring.

- Since services can fail at any time, it's important to be able to detect the failures quickly and, if possible, automatically restore services.

[Microservices. Martin Fowler (2015)]

Characteristics of microservices

Evolutionary design

- Independent replacement and upgradeability
- Sometimes we even expect many services to be scrapped rather than evolved in the longer term.
- Putting components into services adds an opportunity for more granular release planning.

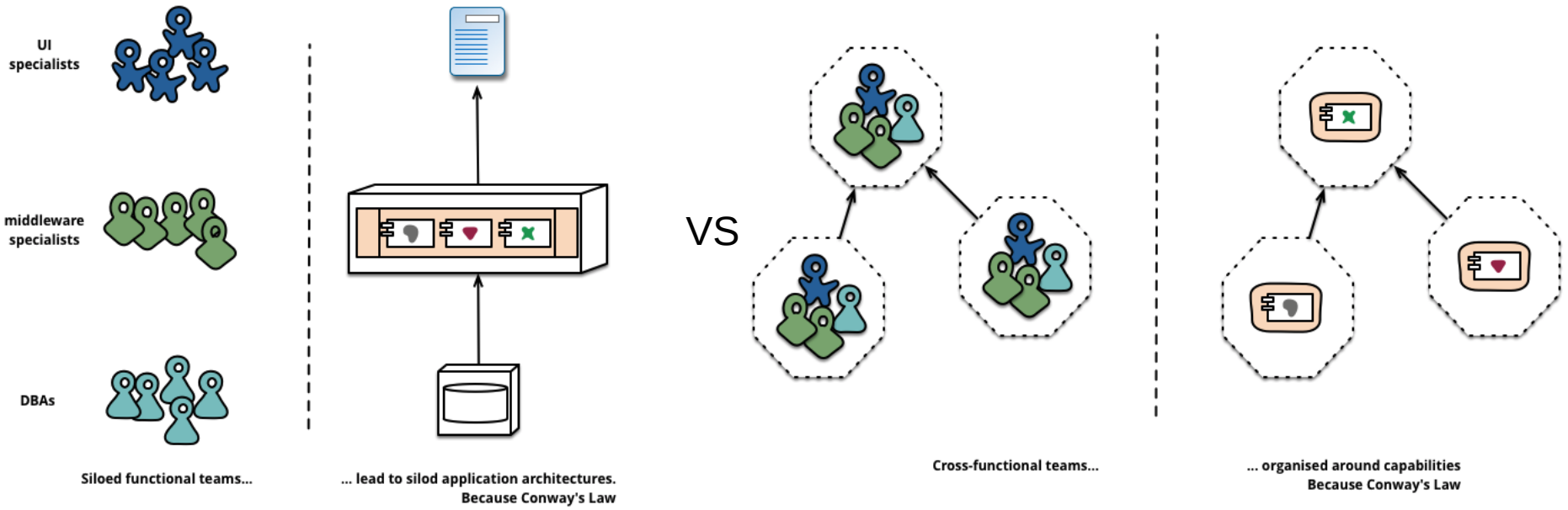
[Microservices. Martin Fowler (2015)]

Characteristics of microservices

Most importantly…

Organized around business capabilities

- When looking to split a large application into parts, often management focuses on the technology layer, leading to UI teams, server-side logic teams, and database teams.
- The microservice approach to division is different, splitting up into services organized around **business capabilities**
- Services take a broad-stack implementation of software for that business area, including user-interface, persistent storage, and any external collaborations
- Teams are cross-functional



[Microservices. Martin Fowler (2015)]

Modeling microservices

Domain driven design and bounded context

- Think of business capabilities, not data models
- An application **domain** *consists of* multiple **bounded contexts**.
- Residing within each bounded context are models that do not need to be shared outside, as well as other models that are shared externally.

- Strategical design
  - divide the system into modular and manageable parts which work together (divide the problem **domain** into **subdomains** – we have heard that before!)
  - identify core business functionalities and map them into subdomains
  - identify supporting and generic subdomains
  - divide the resulting domains along multiple strategies

> “In your organization, you should be thinking not in terms of data that is shared, but about the capabilities those contexts provide the rest of the domain.”

[“Building Microservices: Designing Fine-Grained Systems”. Sam Newman (2015)]

Modeling microservices

Domain driven design and bounded context

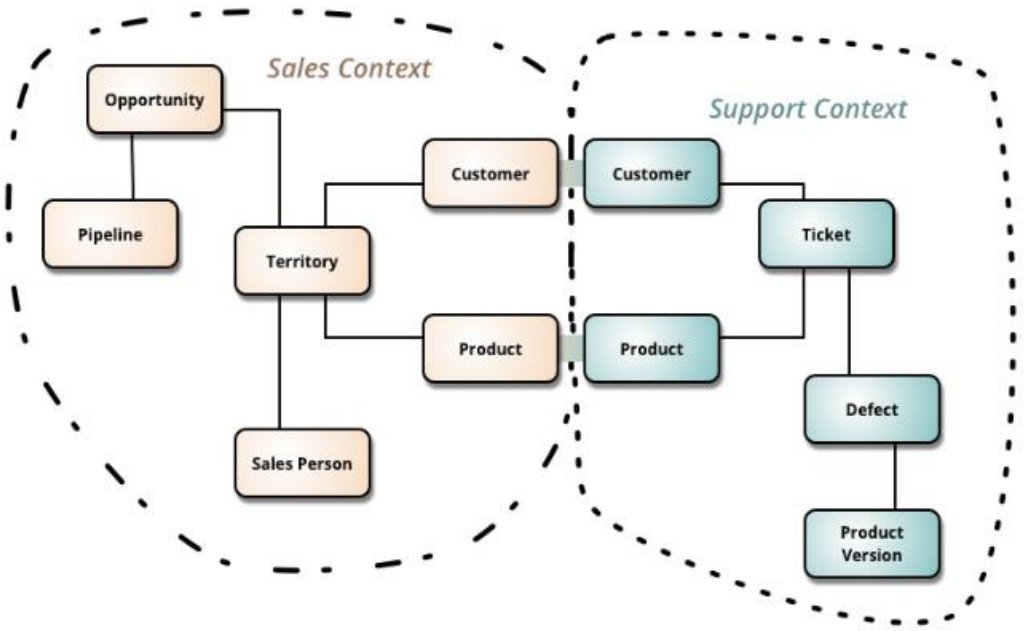
Next, identify bounded contexts

> “A bounded context delimits the applicability of a particular model so that team members have a clear and shared understanding of what has to be consistent and how it relates to other contexts. A bounded context has specific functionality and explicitly defines boundary around team organization, code bases and database schemas.”

[“Domain-driven Design: Tackling Complexity in the Heart of Software”. Eric Evans (2004)]

Some hints:

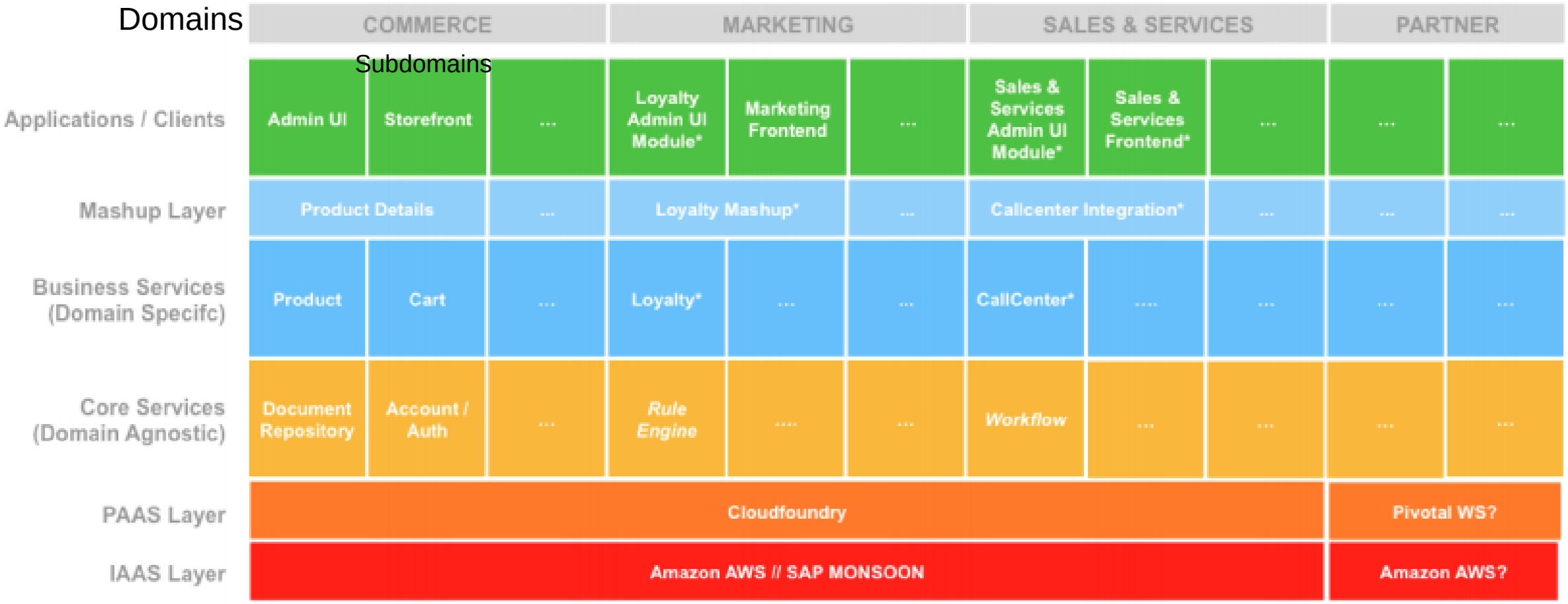
- assign individual bounded context to the subdomains identified during strategical design (see previous slide)
- identify polysemy
- identify independent generic functionality supporting the subdomain



[“Bounded context.” M. Fowler (2014)]
https://martinfowler.com/bliki/BoundedContext.html

Modeling microservices

An example from a case study – SAP Hybris



["Designing a business platform using microservices." R. Kharbuja (2015)]

Challenges in Microservices

- Microservices approach means for the developers the additional complexity of creating a distributed system
  - How to handle communication between the microservices?
- Testing is more difficult for distributed systems
- How to manage distributed transactions?
- How to split (partition) the system into micro-services?
  - What should the size of a microservice be?
  - How can one measure the size?
- How to manage microservice orchestration?

- Security: large attack surface, malicious insiders, integrity

Recommended books: