TDDE41 Software Architectures
Architectural styles

Lena Buffoni
lena.buffoni@liu.se

LINKÖPING UNIVERSITY

Lecture plan

- Abstraction levels in architecture
- Domain specific design
- Architectural patterns
- Architectural styles

Architecture vs. Generic Components

A specific place for every component

Generic building blocks

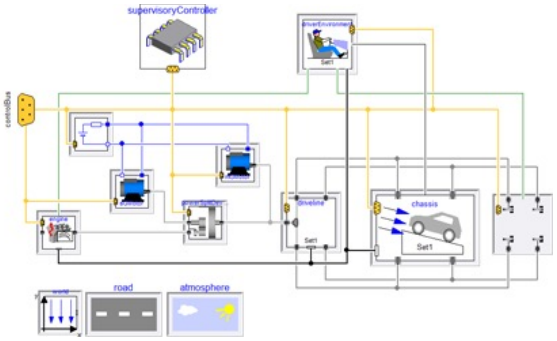
Comparable to top-down vs. bottom-up design strategy?

Reality?

At what abstraction level do we start?

- Solving a more general problem?
- Solving a specific problem?
- Separation of concerns

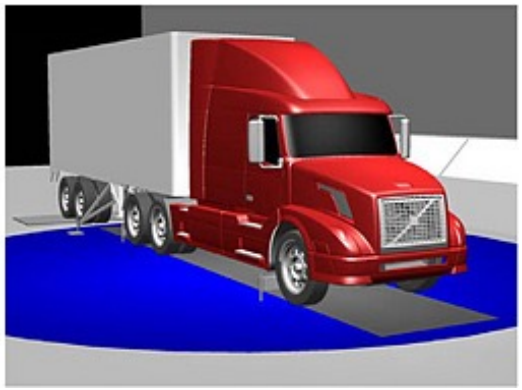
Example



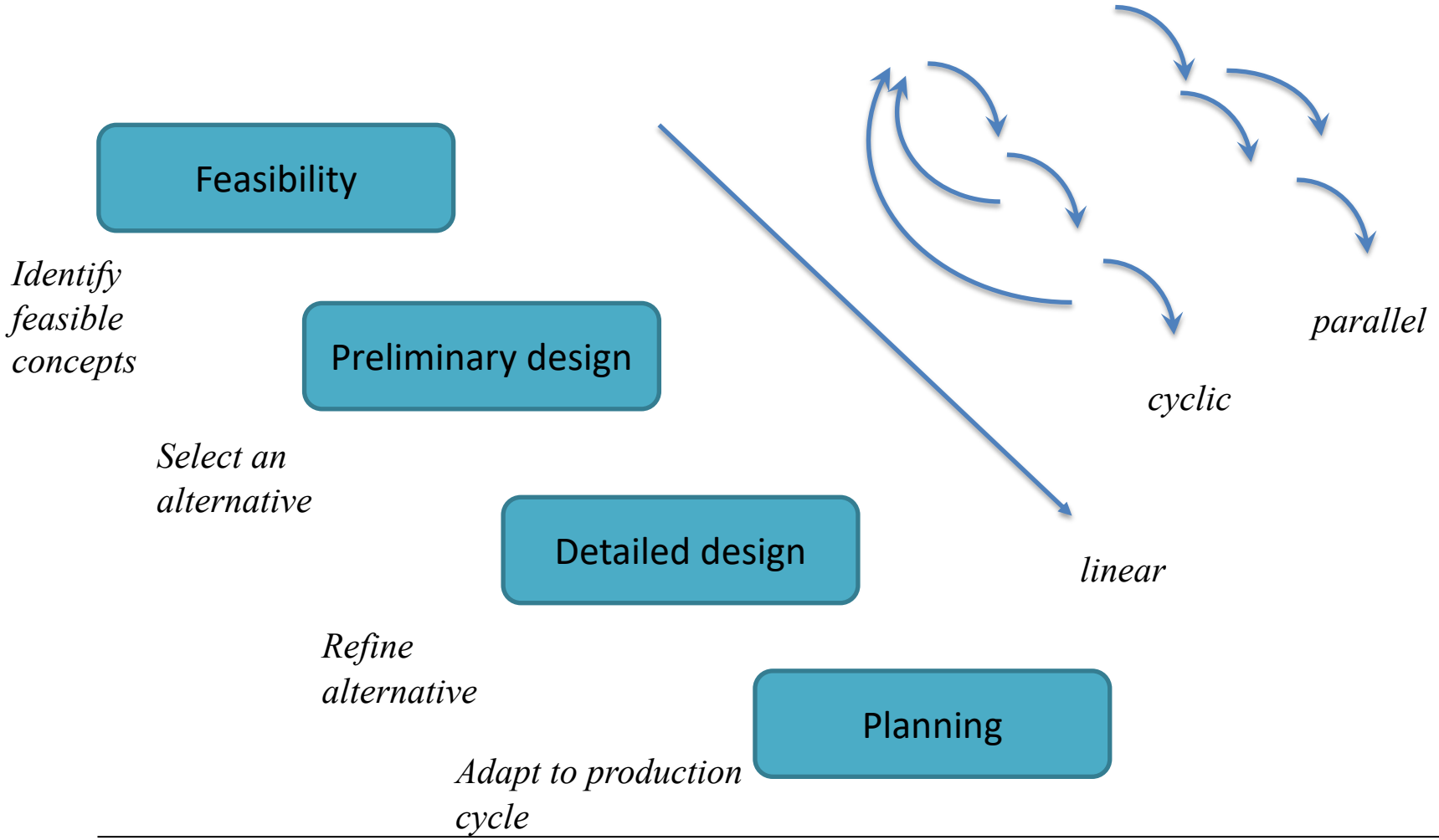
How can we generalize the concepts?

Car Model (in Modelica)

Truck model



Design process

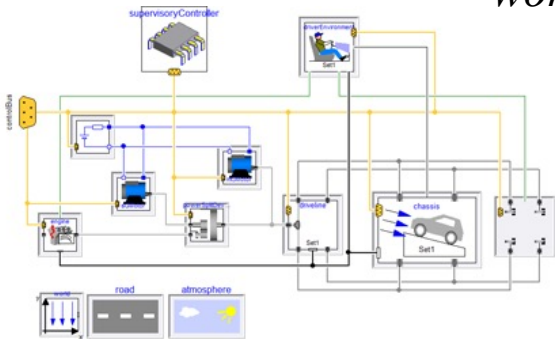


* Design Methods (Architecture), Jones, 1970

Domain specific software architectures

Rarely start from scratch – eg. New car model

Reference architecture	Library of components	Experience
A standard car decomposition layout	*A library of automotive components*	*Knowing how to combine thing so they work*



DSSA : representing domain knowledge

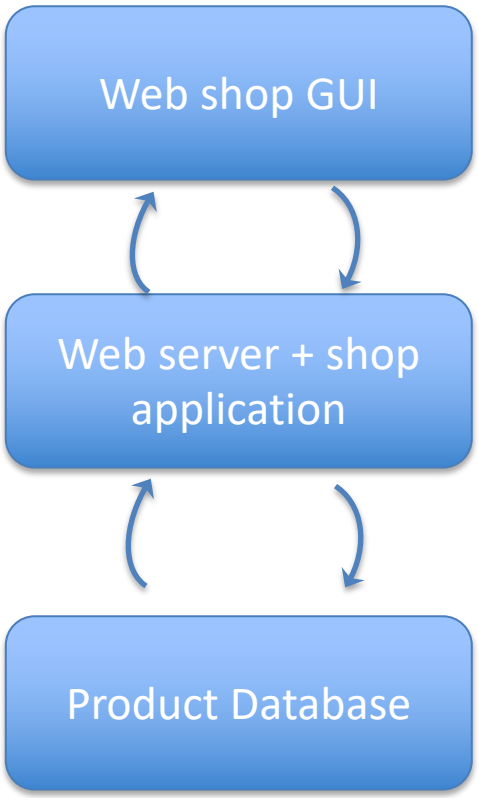
	Example	Represented
Domain dictionary	Powertrain, gear change, chassis	
Information model	A vehicle has a steering wheel and 1+ wheels	ER diagram, Context Information Diagrams
Feature model	User can start car, user can press accelerator pedal, user can get current speed	Use case diagrams, Feature relationship diagrams
Operational model	Car starts at rest -> Engine is started -> Gear is changed…	Data flow, control flow, state transition diagrams

Architectural patterns

Reminder:

- Are applicable in a given development context
- Constrain architectural design decisions that are specific to a particular system within that context
- Elicit beneficial qualities in each resulting system

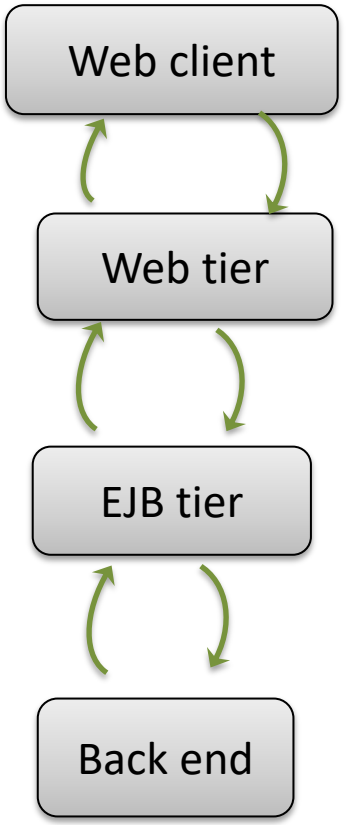
State-Logic-Display (Three-Tier)



UI

business logic

state



4 tier break down

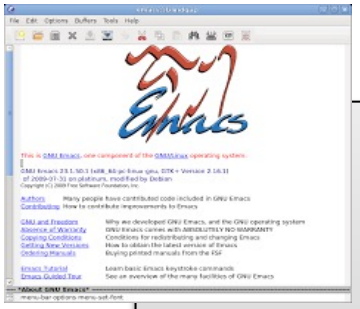
Tier architecture

- Structure the system as groups of components organized based on type of component, runtime-purpose, execution environment, …
- Communications with components in the same or adjacent tier – may restrict kinds of communication

+ simplify modifiability
+ easy to ensure security
+ good for performance management

- high cost, high complexity

Model-View-Controller



Keyboard input, mouse gestures, menu selections

Lisp-level events

View

Frames, windows, incremental display update logic

Window manipulation primitives

Buffer content rendering

Controller

Lisp interpreter, lisp code, lips objects

Model

Buffers, text properties, markers, overlays

Buffer access primitives

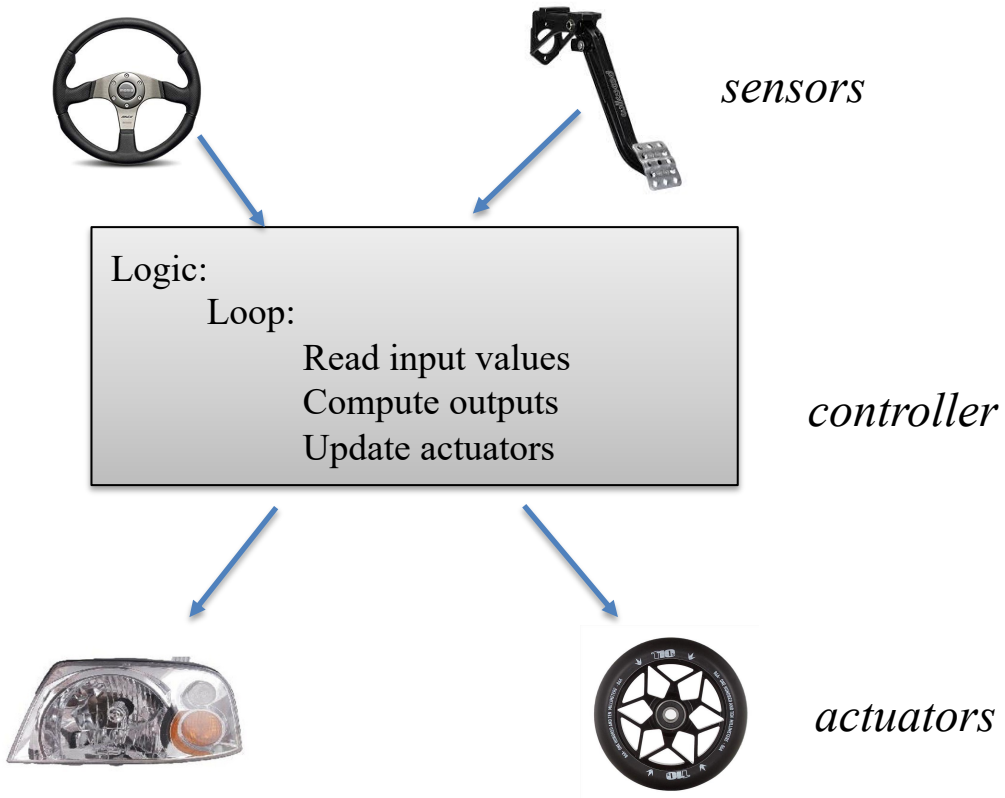
Emacs and feature driven development

- Easy to add a feature/extend the tool
- Interactive
- Safe – bugs will not crash the editor
- Everything is on the same level of abstraction (buffers, windows, your own extensions)

+ high cohesion, low coupling

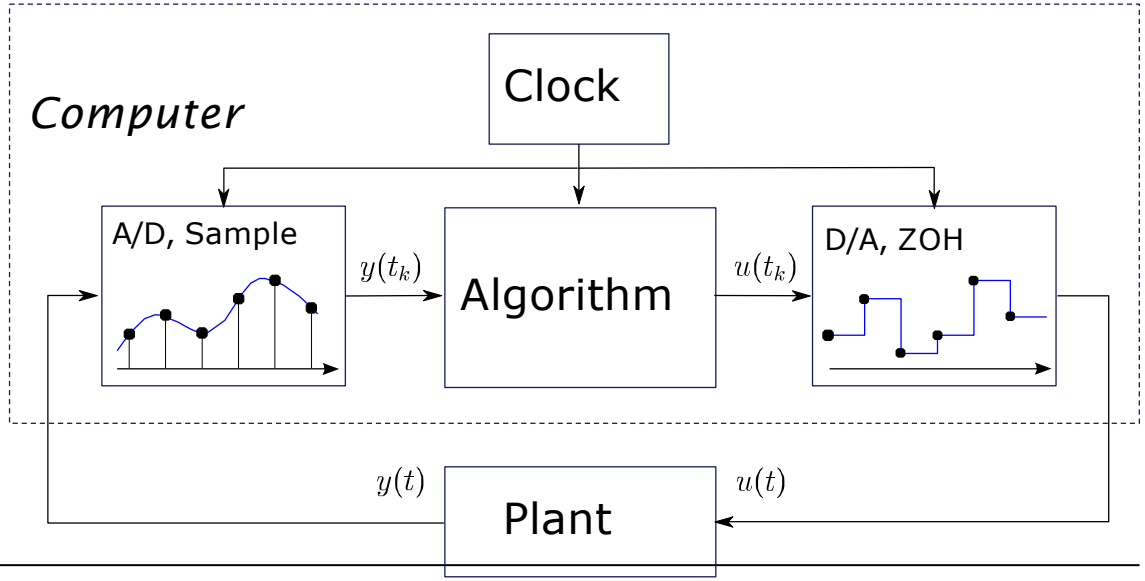
- Consistency and maintainability

Sensor-Controller-Actuator



Sensor-Controller-Actuator

- Typically scheduled on a clock
- Implicit interaction via the environment – a change on the actuators will result in a change sensors will detect

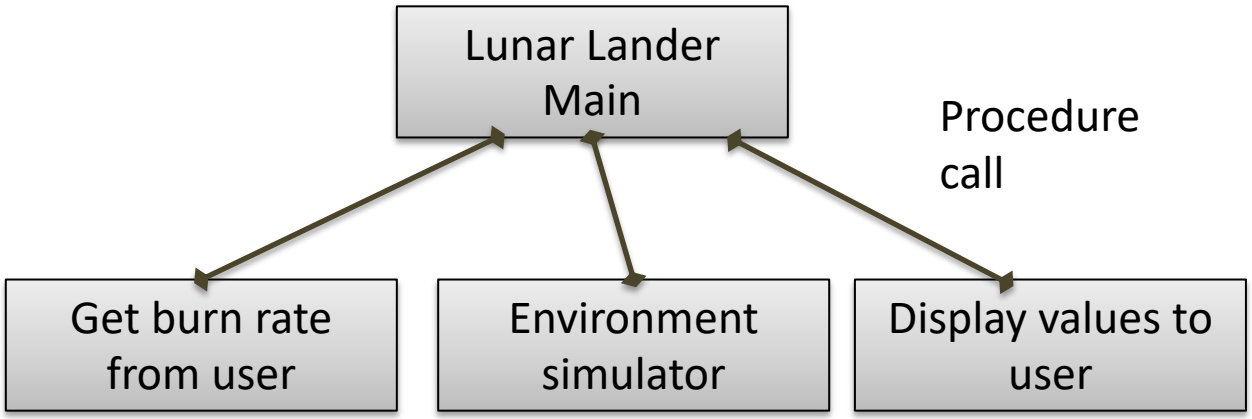


Architectural Styles

- An architectural style is a named collection of architectural decisions that
  - Applicable to recurring design problems
  - Parametrized for different contexts

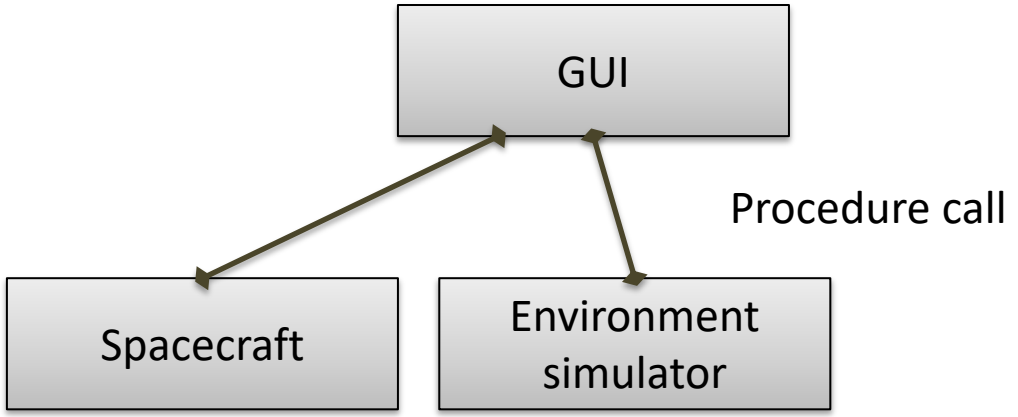
Program and Subroutines

- From classical imperative programming
- Small programs
- Does not scale or adapt well



Object-oriented

- Good for complex dynamic data structures
- Close coupling to real world entities
- Harder to distribute
- Needs additional structuring



Layered

- Design separated into layers
- Each layer obtains services from the layer beneath
  - Virtual Machines
  - Client-Server

Virtualization

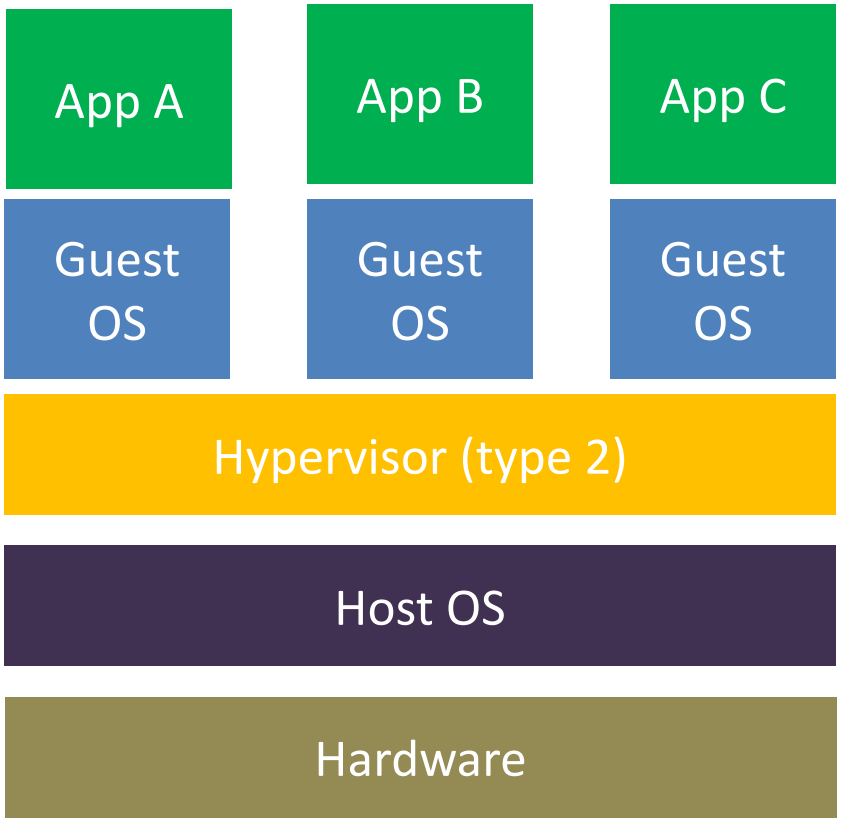
- Abstract the hardware and infrastructure
- Allow a unified user experience
- Energy saving
- Secure

But:

- Overhead
- Compatibility limitations
- Shared resources

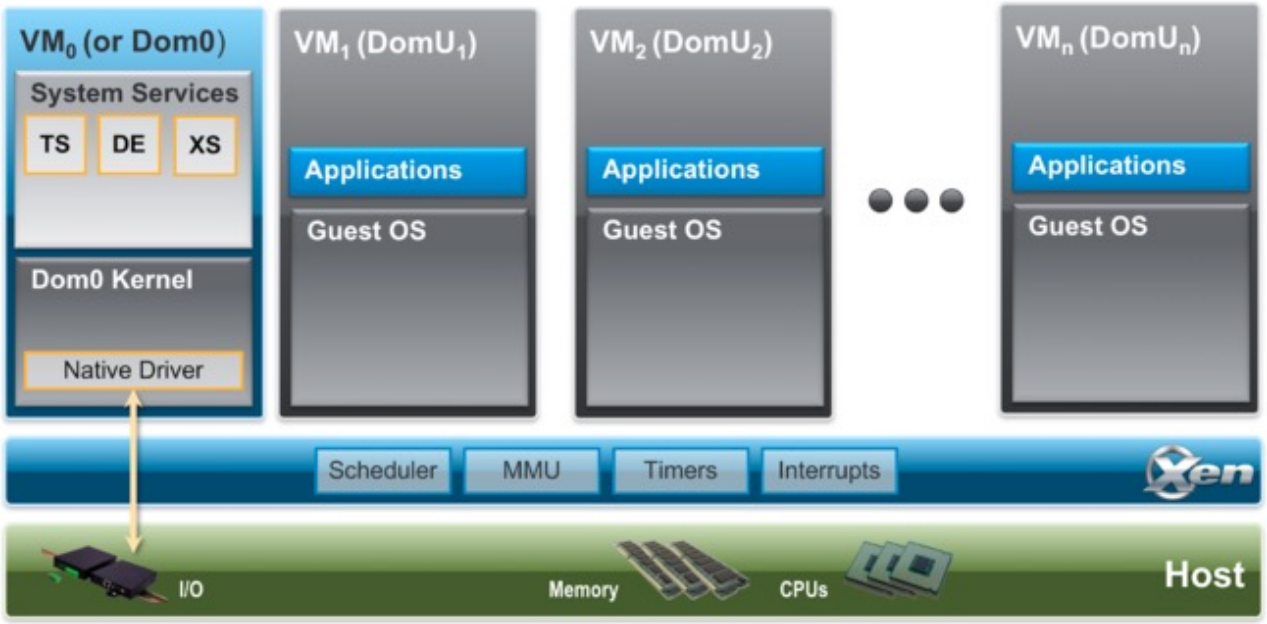
VMWare, VirtualBox, Xen …

Virtualization: Portability on machine level



Managing resource virtualization and distribution

Virtualization:
Xen Architecture for distributed computing

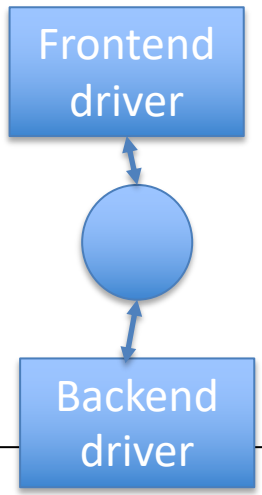


Highest privilege level, we separate the management logic, to reduce impact of errors

Paravirtualization

- Making changes to the program or operating system to optimize its performance
- Remove features that are difficult to virtualize and replace them with paravirtual features

- Eg: memory handling

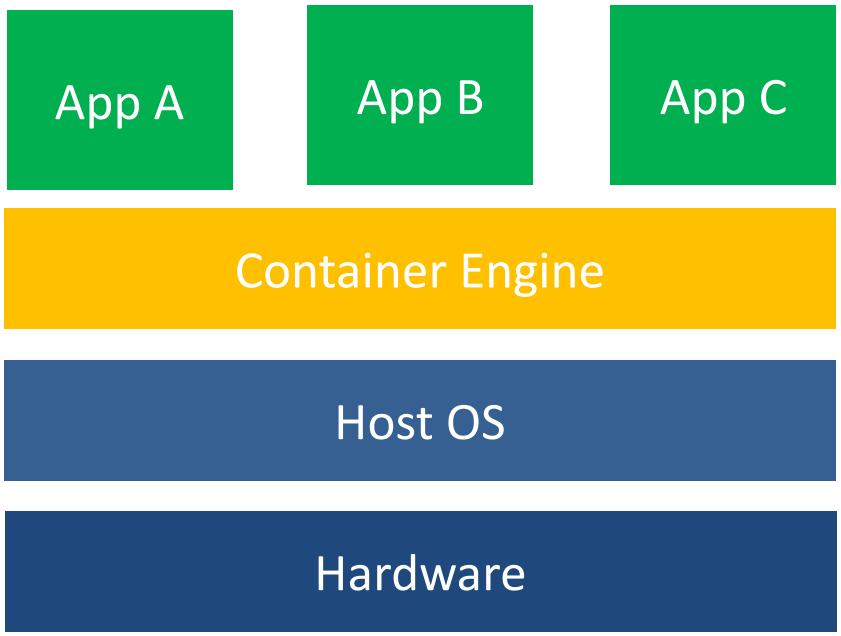


Implement familiar interface for operating system

Ring buffer

Connects to real hardware

Containerization:
Portability on OS/application level



Containerization

- Lightweight
- Portability
- Less overhead
- Breakdown into smaller chunks

But

- Less isolation
- No hardware virtualization
- Still… overhead

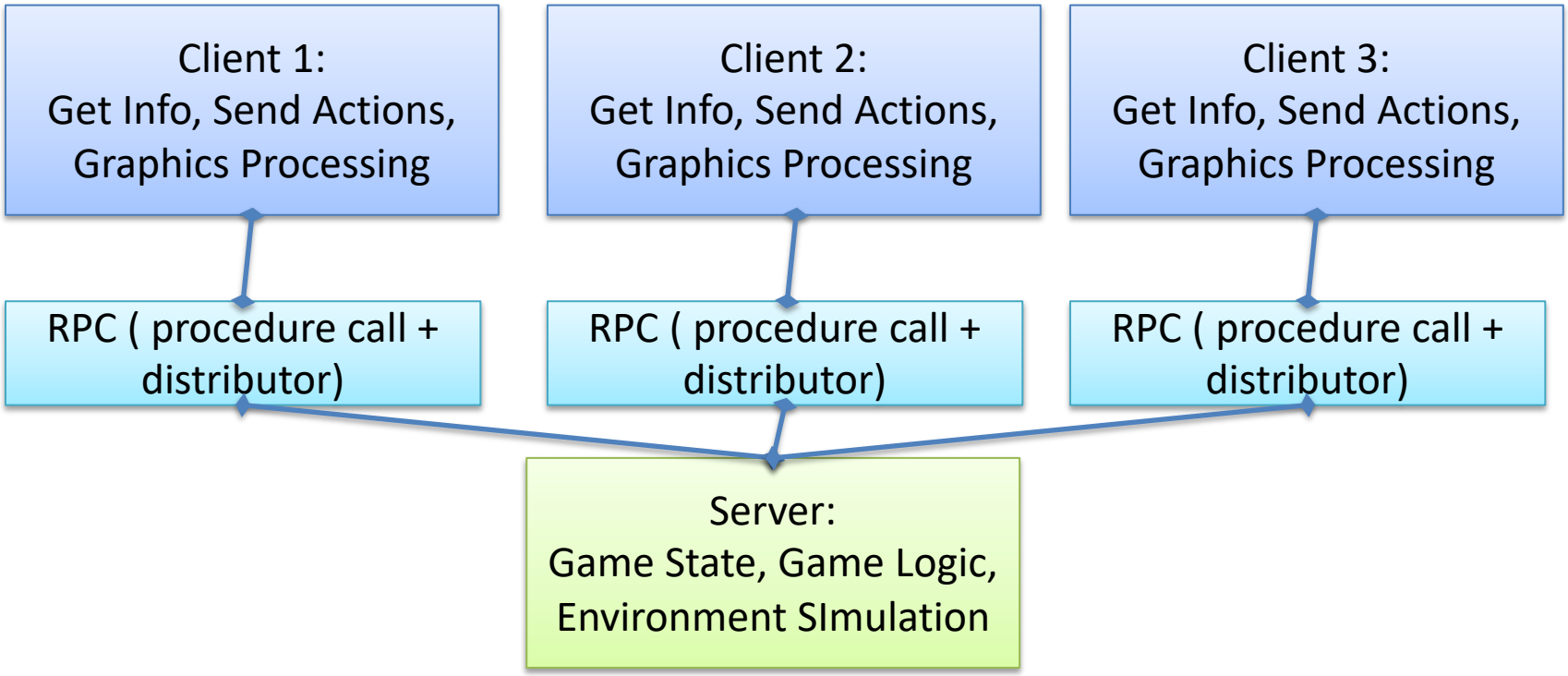
Complementary approaches?

Docker, Rocket…

Emulation vs virtualization

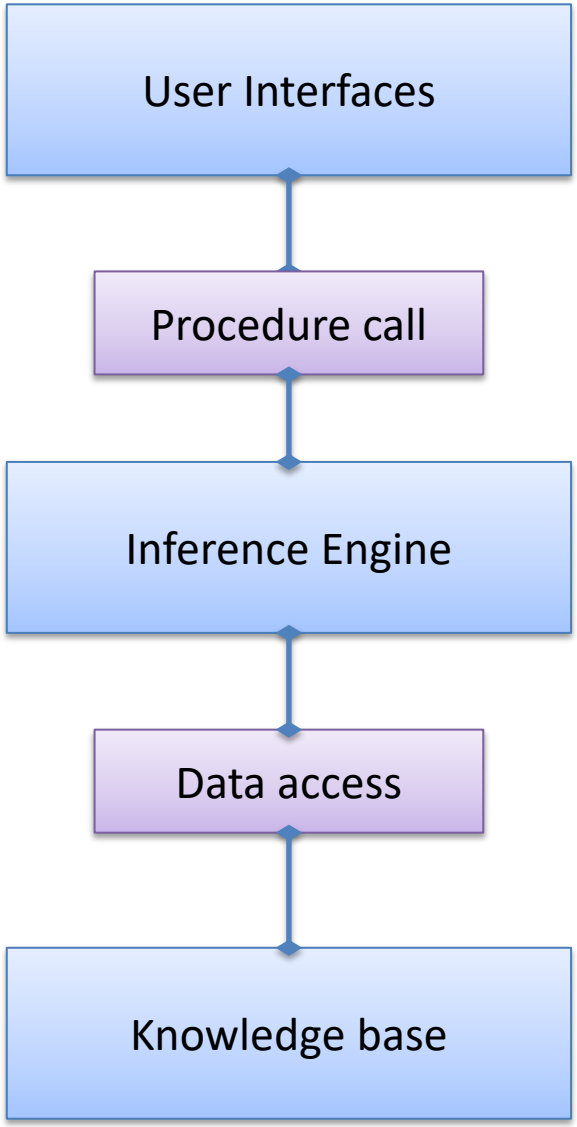
- Virtualized instructions run directly on the processor
- Emulated instructions are translated before execution

Client-server: a form of virtualization



Data-flow

- Batch-Sequential
- Pipe and filter
- Rule-based



Interpreter

Dynamic parsing and execution of commands (eg: Excel)

Mobile code

Used for distributed data processing

+ Dynamic

+ Evolutive

- Overhead

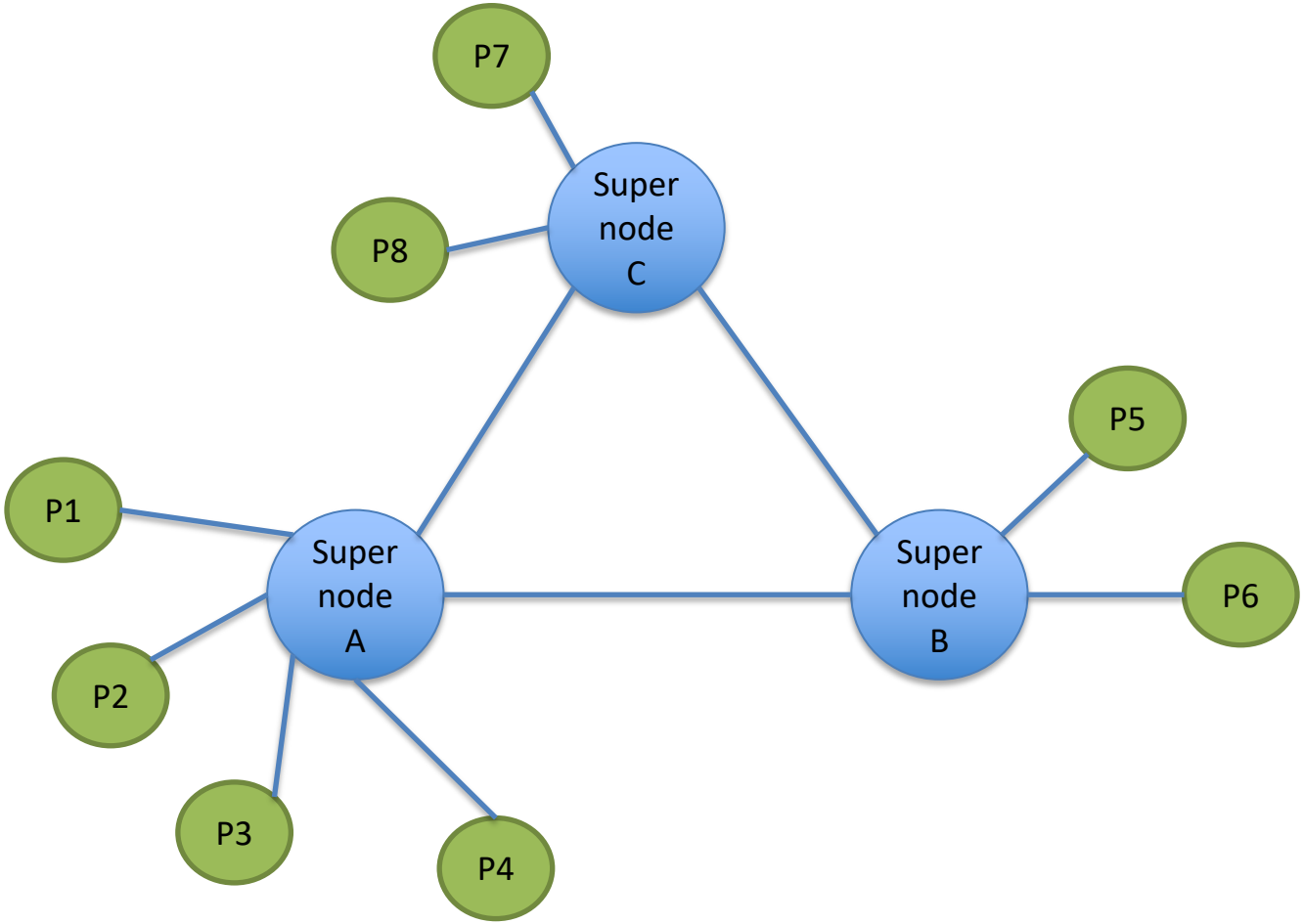
- Memory management

Peer-to-peer

- Network of peers
- Loosely coupled
- Autonomous
- Decentralized resources
- Requests propagate until information is discovered or threshold is reached

+ Scalable
+ Robust
- Latency
- Security

Peer-to-peer architecture



Skype - mixed client/server and P2P

- Download a Skype client
- Register and log in to server
- Get provided with a supernode address
- Queries and voice calls over the supernode
- Location of supernodes determined based on topology and machine characteristics
- Any peer can become a supernode

Skype – architecture properties

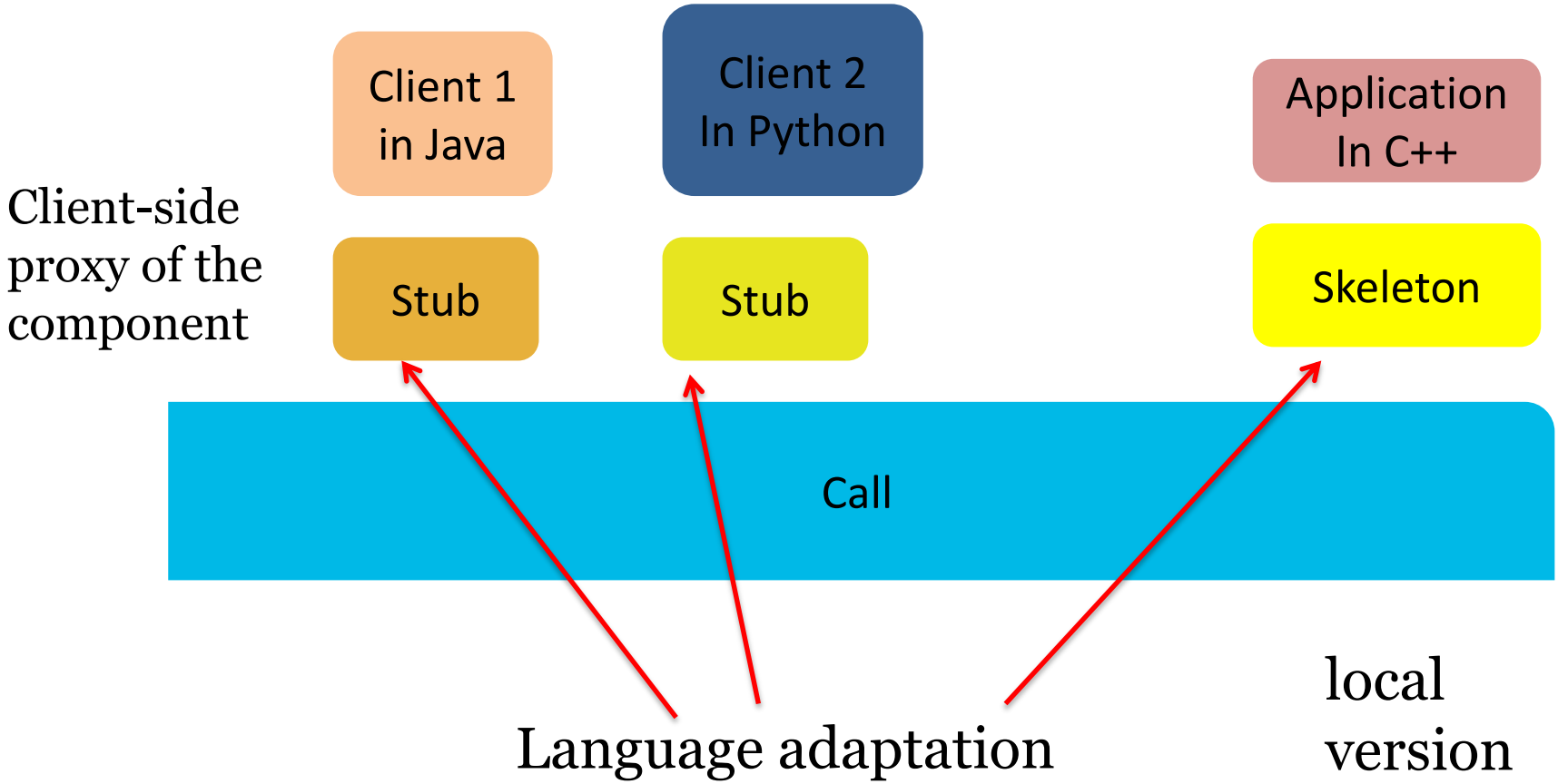
- Mixed model handles the discovery problem
- Scalable and robust
- Supernodes are chosen to maximize performance
- As many supernodes can be created as necessary
- Encryption protocol to ensure privacy
- Restriction to Skype controlled clients reduces risk for malaware

Distributed paradigms

- Distributed objects
- CORBA
- MOM

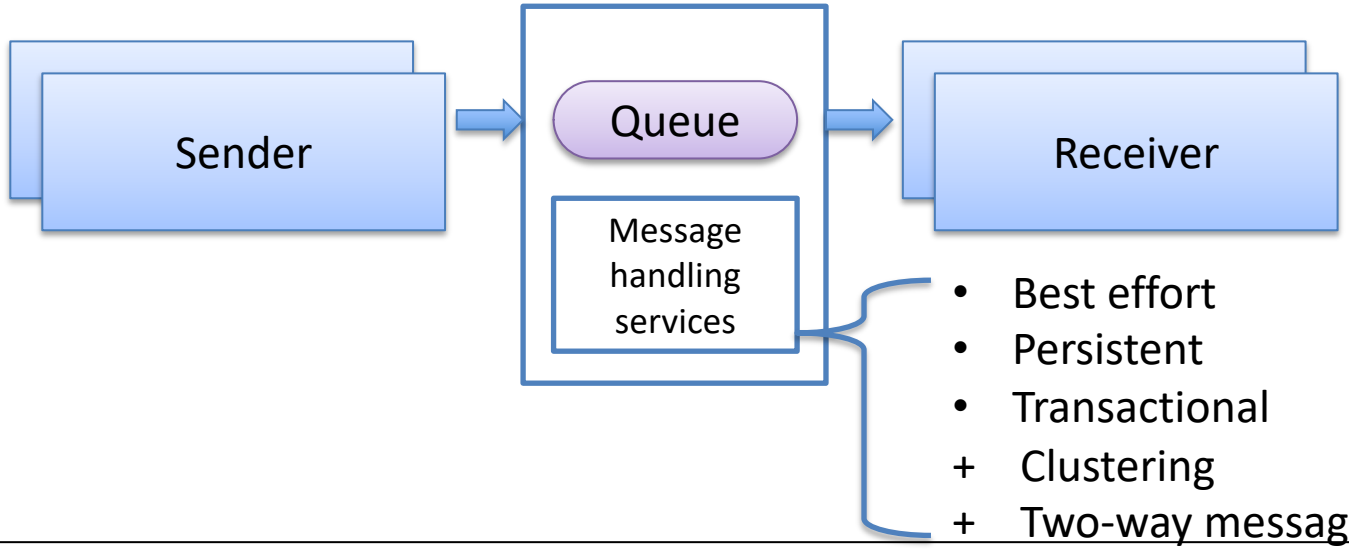
Not part of the lecture

The stub & skeleton proxy pattern



Message-Oriented Middleware (MOM)

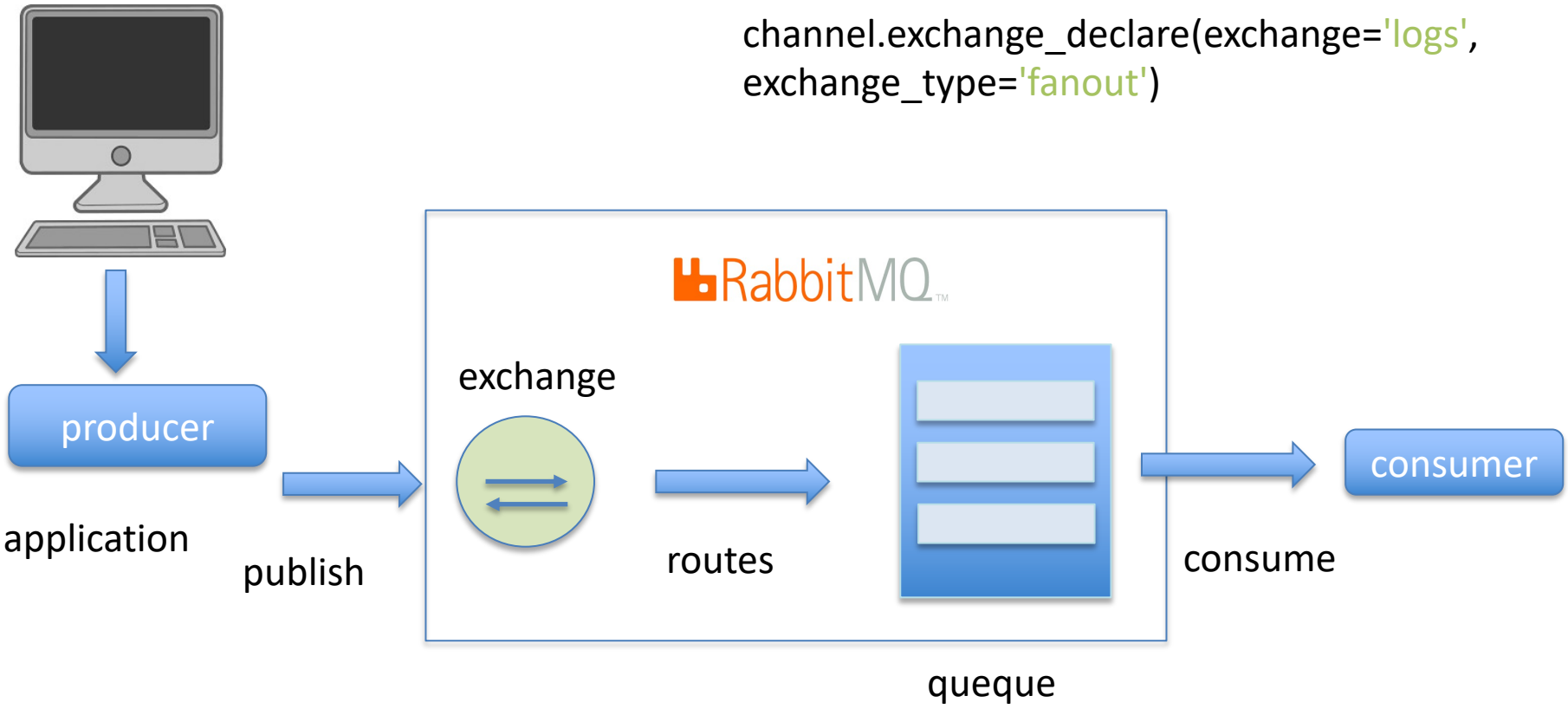
- Loosely coupled
- Asynchronous
- Used to connect independent applications



Rabbit MQ

- 4 communication models : direct, fanout, topic, headers
- producers send to the exchange then the queue
- multi-platform

```
channel.exchange_declare(exchange='logs',
exchange_type='fanout')
```

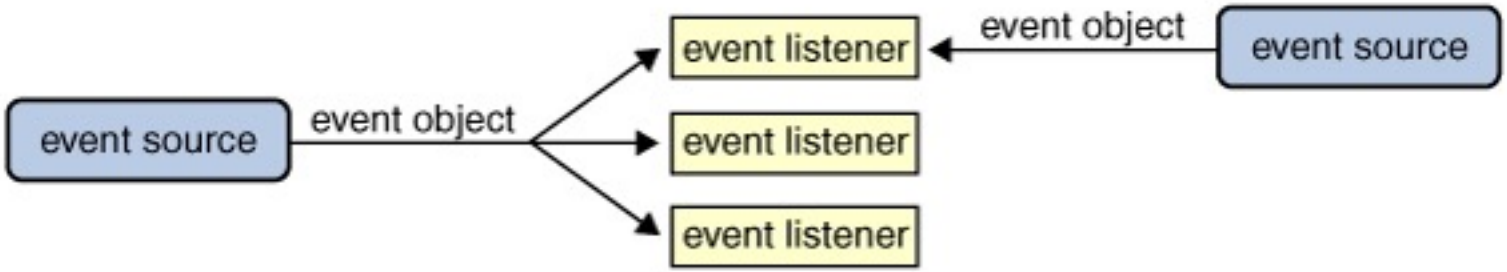


Event-based

- Independent
- Concurrent

+ scalable
+ distributed
+ supports heterogeneity

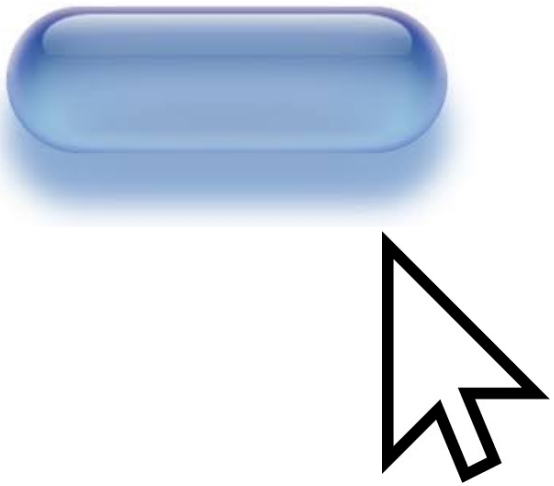
- no guarantee events will be processed



Java beans – main aspects

- *Events* Beans can announce that their instances are potential sources or listeners of specific types of events. An assembly tool can then connect listeners to sources.
- *Properties* Beans expose a set of instance properties by means of pairs of getter and setter methods.
- *Introspection* An assembly tool can inspect a bean to find out about the properties, events, and methods that a particular bean supports.
- *Customization* Using the assembly tool, a bean instance can be customized by setting its properties.
- *Persistence* Customized and connected bean instances need to be saved for reloading at the time of application use.

Events



- Event (MouseOver)
- Event source – generates an event (mouse hoovers over a button)
- Event listener - triggers some behavior when an event is detected (button is highlighted)

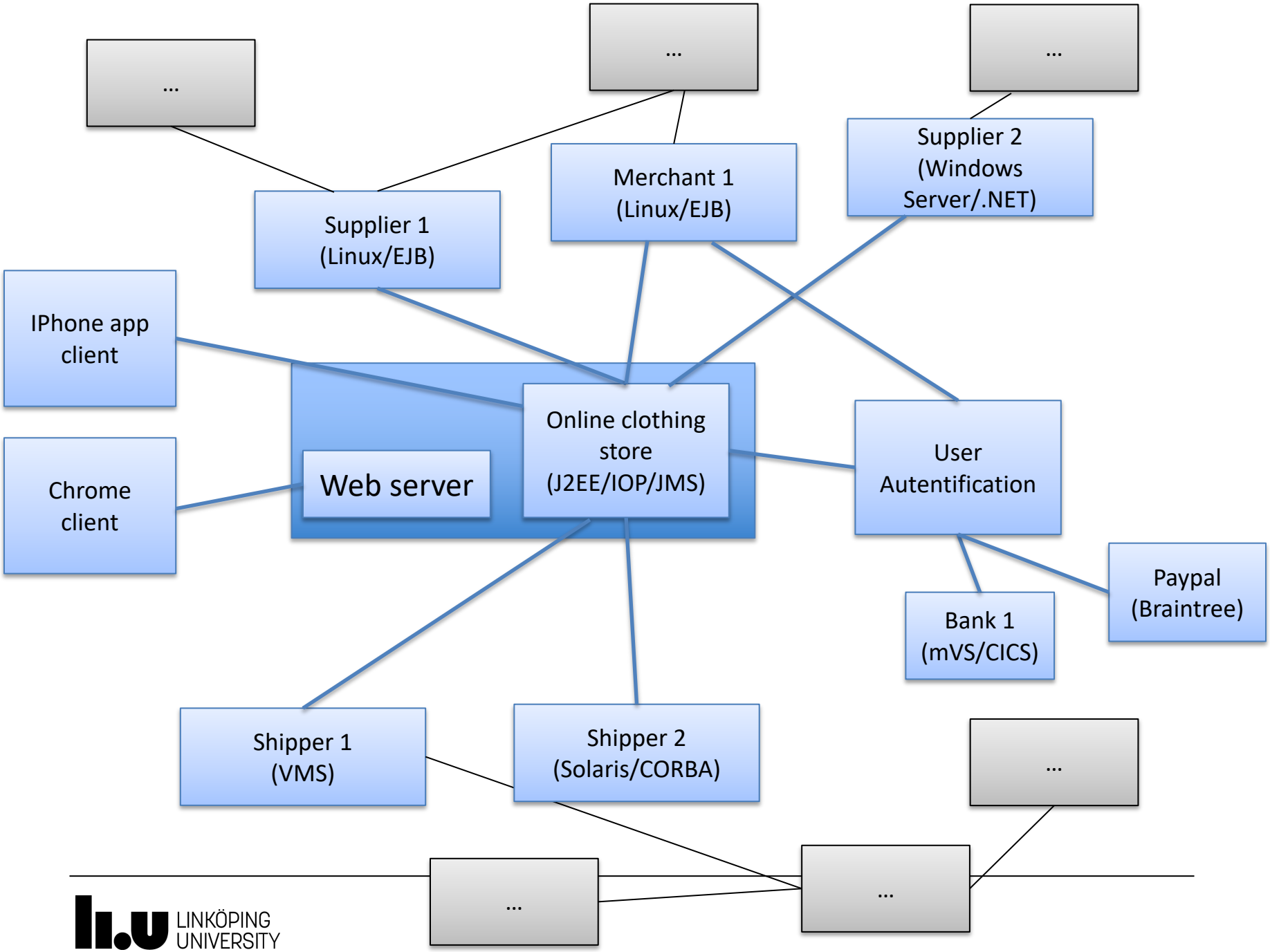
```
public void add<Event>Listener(<Event>Listener a)
```

Service Oriented Architectures

A form of application integration middleware

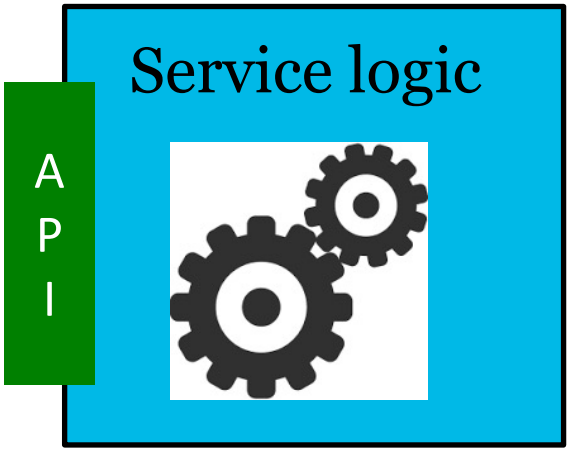
Goals:

- Interoperability
- Distributed systems
- Standardization (?)


...
...
...
Supplier 1
(Linux/EJB)
Merchant 1
(Linux/EJB)
Supplier 2
(Windows
Server/.NET)
IPhone app
client
Chrome
client
Web server
Online clothing
store
(J2EE/IOP/JMS)
User
Autentification
Paypal
(Braintree)
Bank 1
(mVS/CICS)
Shipper 1
(VMS)
Shipper 2
(Solaris/CORBA)
...
...
...
LINKÖPING
UNIVERSITY

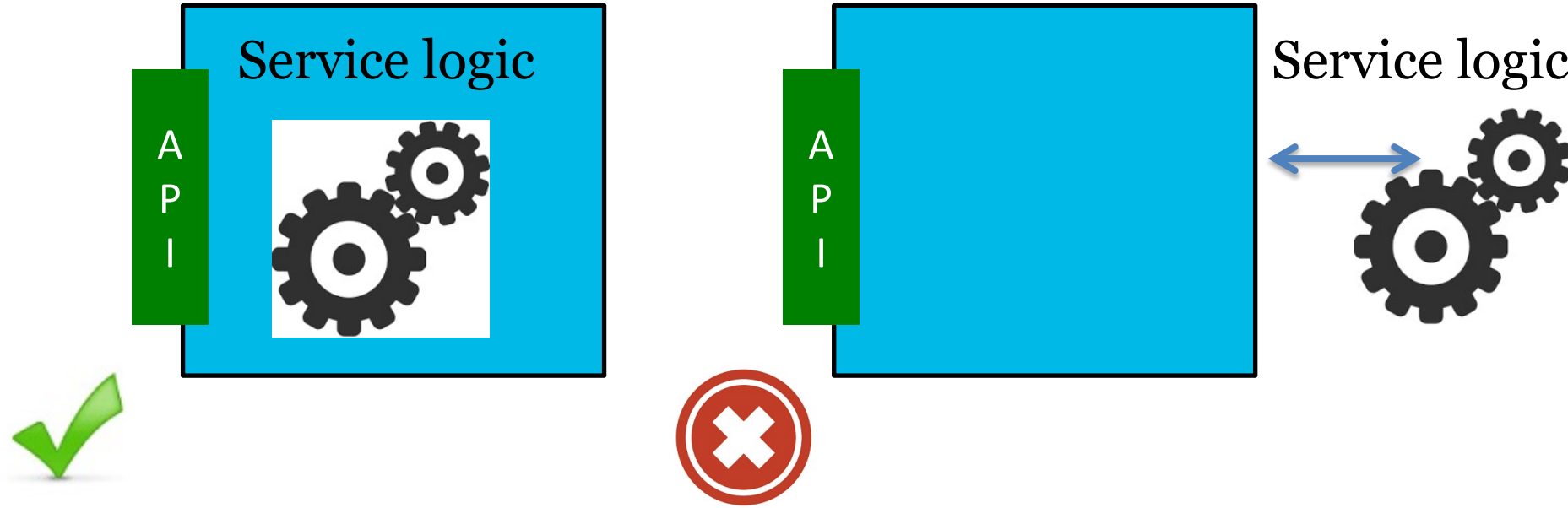
Encapsulation

- All access through API
- Explicit API definitions
- No hidden interactions
- Explicit data passing
- Context-free calls



Autonomy

- Replaced, managed and deployed independently
- Responsible for their own security



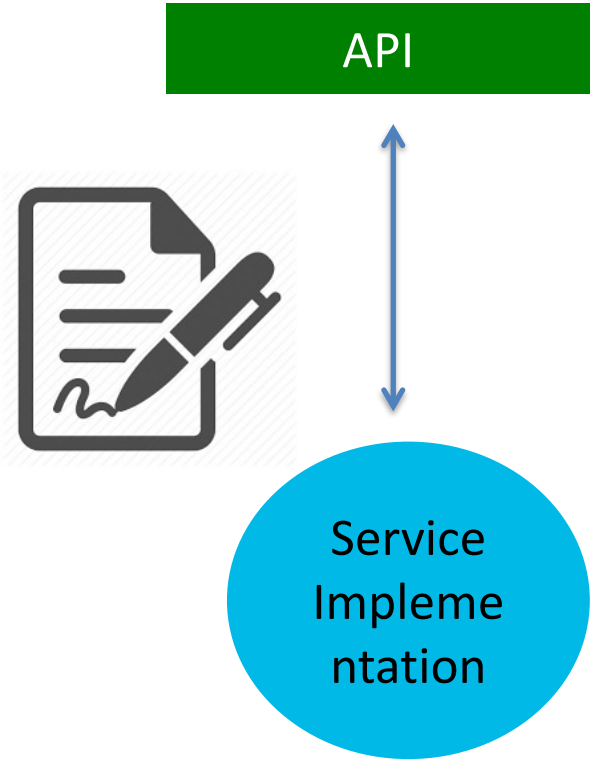
Loose coupling

Low dependency/connection level in terms of

- Time
- Location
- Type
- Version
- Cardinality
- Lookup
- Interface

Contract driven

- Not class driven
- Description based
- Loose coupling = few assumptions on implementation



ex: RMI is implementation dependent – not SO

Interoperability and standards

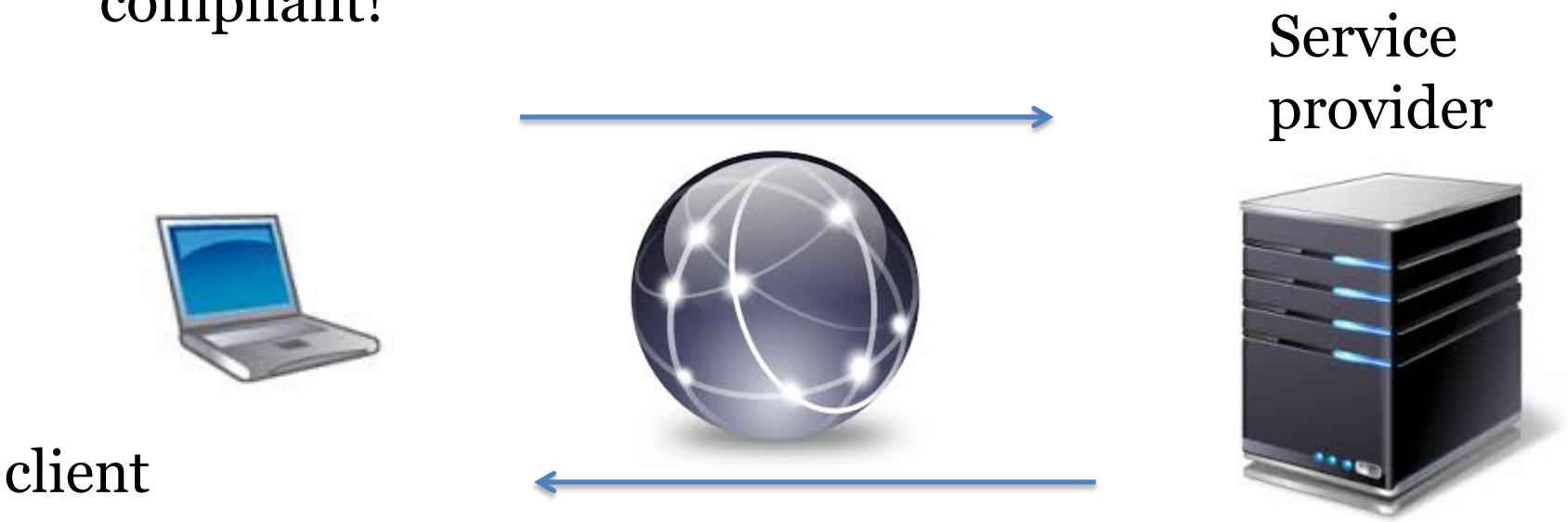
- Relies on open standards not proprietary APIs
- All message formats are described using an open standard, or a human readable description
- The semantics and syntax for additional information necessary for successful communication, such as headers for purposes such as security or reliability, follow a public specification or standard
- At least one of the transport (or transfer) protocols used to interact with the service is a (or is accessible via a) standard network protocol

Vendor & Technology independent

To ensure the utmost accessibility (and therefore, long-term usability), a service must be accessible from any platform that supports the exchange of messages adhering to the service interface as long as the interaction conforms to the policy defined for the service.

Web-services

- A realization of the SOA paradigm
- Relies on XML and the Web for implementation
- Attention: a web service is not necessarily SOA compliant!



XML format

- W3C standard
- Open & extensible
- Structured
- Readable

But

- Heavy
- No semantics

Example

```
<breakfast-menu>
  <food>
       <name>Belgian Waffles</name>
       <price>$5.95</price>
       <description>
          two of our famous Belgian
Waffles
    with plenty of real maple syrup
       </description>
       <calories>650</calories>
  </food>
  <food>
       <name>Strawberry Belgian
Waffles</name>
       <price>$7.95</price>
       <description>
    light Belgian waffles covered with
    strawberries and whipped cream
       </description>
       <calories>900</calories>
  </food>
</breakfast-menu>
```

We need some way to define XML structure

XML Schema : type definition

Item instance

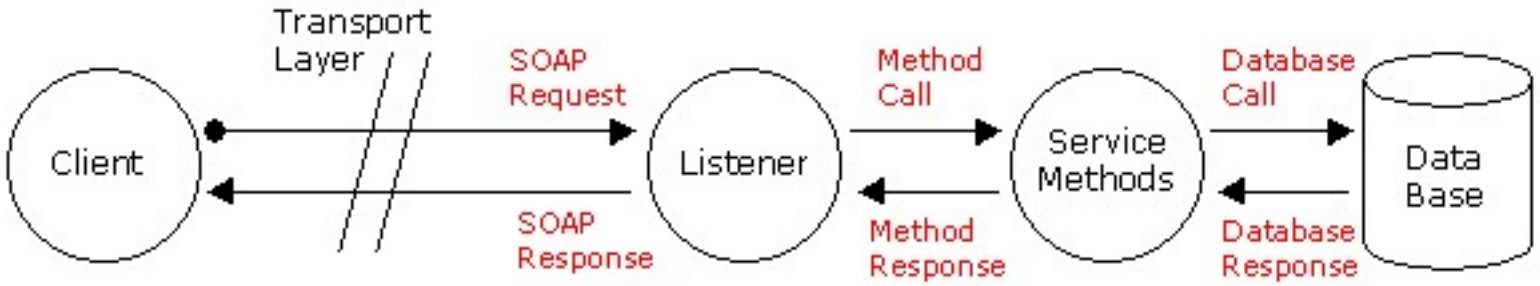
```
<item>
    <title>Empire Burlesque</title>
    <note>Special Edition</note>
    <quantity>1</quantity>
    <price>10.90</price>
  </item>
```

Item type definition

```
<xs:element name="item" maxOccurs="unbounded">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="title" type="xs:string"/>
      <xs:element name="note" type="xs:string"
minOccurs="0"/>
      <xs:element name="quantity"
type="xs:positiveInteger"/>
      <xs:element name="price" type="xs:decimal"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Simple Object Access Protocol (SOAP)

- XML based message exchange protocol
- Used for remote procedure calls (RPC)
- Platform and language independent
- Uses predefined channels (HTTP, SMTP, TPC)



SOAP message structure

```
<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
soap:encodingStyle="http://www.w3.org/2003/05/soap-
encoding">

<soap:Header>
...
</soap:Header>

<soap:Body>
...
  <soap:Fault>
  ...
  </soap:Fault>
</soap:Body>

</soap:Envelope>
```

Mandatory, define the message as a soap envelope

Optional, application specific information

Optional, contains the actual message, can contain error information

SOAP example

```
<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-
envelope/"
soap:encodingStyle="http://www.w3.org/2003/05/so
ap-encoding">

<soap:Body>
  <m:GetPrice
xmlns:m="http://www.w3schools.com/prices">
    <m:Item>Apples</m:Item>
  </m:GetPrice>
</soap:Body>

</soap:Envelope>
```

SOAP Pros & Cons

+W3C Recommendation (standard)

+ Implements RPC

+ Lightweight, extensible, neutral

- Untyped user data, types to encode in the message - Interpretation of SOAP messages required
- High overhead / low performance

- Serialization by value and not by reference

WSDL (Web Services Description Language)

- WSDL is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information.
- Used to describe:
  - a service for its clients
  - a standard service for WS implementers
- W3C standard

WSDL Interface Definition

- defines **services** as collections of network endpoints, or **ports**
- the abstract definition of endpoints and messages is separated from their concrete network deployment or data format bindings
- **messages**, which are abstract descriptions of the data being exchanged, and **port types** which are abstract collections of **operations**

WSDL interface structure

```
<definitions>

<types>
  data type definitions........
</types>

<message>
  definition of the data being
communicated....
</message>

<portType>
  set of operations......
</portType>

<binding>
  protocol and data format
specification....
</binding>

</definitions>
```

WSDL example : Glossary

```
<message name="getTermRequest">
  <part name="term" type="xs:string"/>
</message>

<message name="getTermResponse">
  <part name="value" type="xs:string"/>
</message>

<portType name="glossaryTerms">
  <operation name="getTerm">
    <input message="getTermRequest"/>
    <output message="getTermResponse"/>
  </operation>
</portType>
```

WSDL Binding to SOAP

- binding = associating protocol or data format information with an abstract entity like a message, operation, or portType
- SOAP specific elements include:
  - soap:binding
  - soap:operation
  - soap:body

Binding example

```
<binding type="glossaryTerms" name="b1">
   <soap:binding style="document"
   transport="http://schemas.xmlsoap.org/soap/
http" />
   <operation>
     <soap:operation
soapAction="http://example.com/getTerm"/>
     <input><soap:body use="literal"/></input>
     <output><soap:body
use="literal"/></output>
  </operation>
</binding>
```

binding of one-way operation over SMTP using a SOAP Header

WSDL Pros & Cons

WSDL abstracts from underlying

- Protocol (Binding to HTTP, SOAP, MIME, IIOP…)
- Component model (Mappings to CORBA, EJB, DCOM, .NET …)
- Supported by many tools (Visual Studio, Eclipse, …)

But:

- No inheritance on WSDL
- Not recursively composable

JAX-WS

- No need to manually format SOAP messages
- Converts API calls and responses from/to SOAP
- Provides WSDL mappings

```
@WebService
public class Hello {
private String message = new String("Hello, ");
@WebMethod
public String sayHello(String name) {
return message + name + ".";
}
}
```

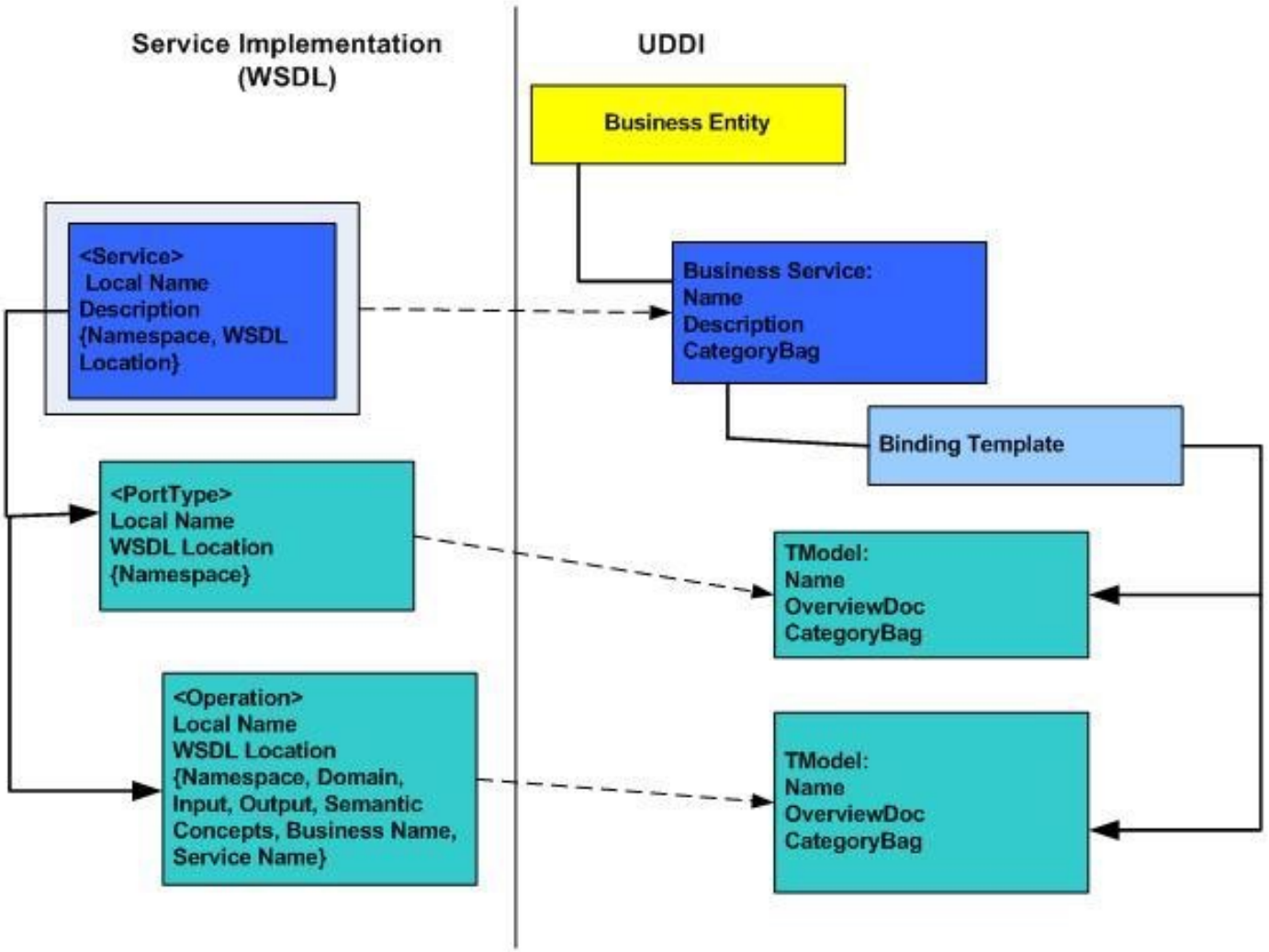
Categorization and discovery

- Providers need a way to propose their services
- Clients need a way to find available services

- UDDI (Universal Description, Discovery, and Integration) is an XML-based registry for businesses worldwide to list themselves on the Internet
- Like a telephone book for services

UDDI

- a specification for a distributed registry of web services.
- platform-independent, open framework.
- can communicate via SOAP, CORBA, Java RMI Protocol.
- Uses WSDL to describe interfaces to web services.


Service Implementation
(WSDL)
UDDI
Business Entity
<Service>
Local Name
Description
{Namespace, WSDL
Location}
Business Service:
Name
Description
CategoryBag
Binding Template
<PortType>
Local Name
WSDL Location
{Namespace}
TModel:
Name
OverviewDoc
CategoryBag
<Operation>
Local Name
WSDL Location
{Namespace, Domain,
Input, Output, Semantic
Concepts, Business Name,
Service Name}
TModel:
Name
OverviewDoc
CategoryBag

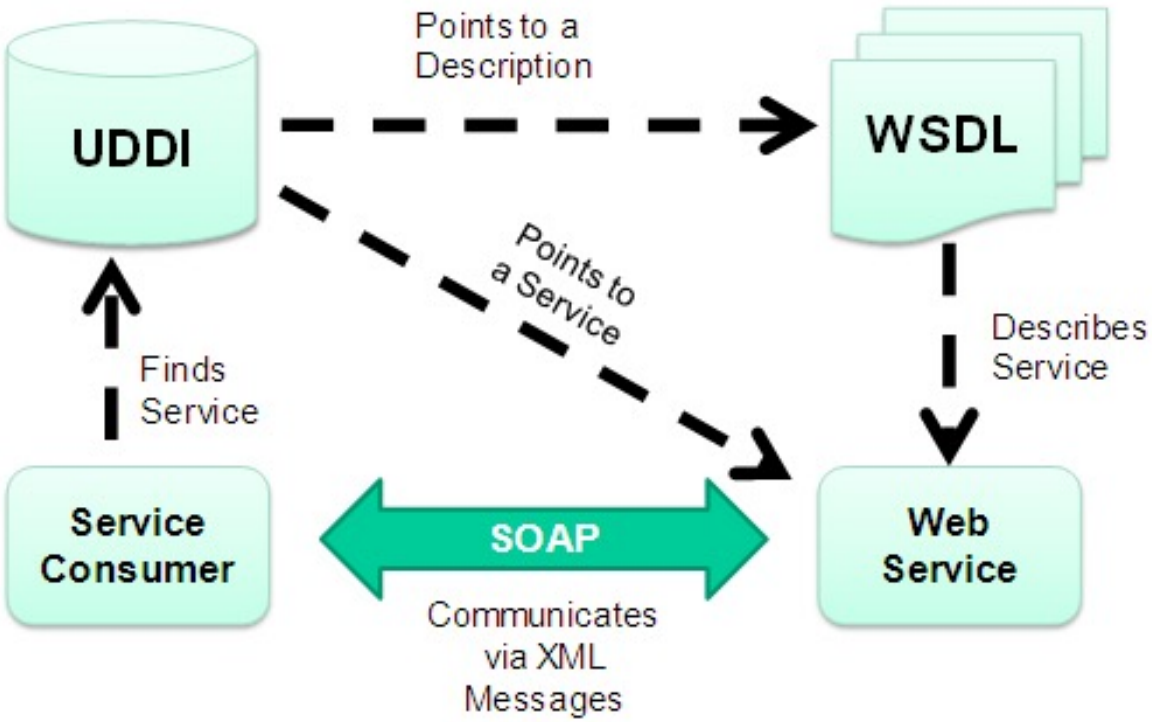
UDDI Example

```
<tModel authorizedName="..." operator="..." tModelKey="...">
   <name>HertzReserveService</name>
   <description xml:lang="en">
      WSDL description of the Hertz reservation service interface
   </description>

   <overviewDoc>
      <description xml:lang="en">
         WSDL source document.
      </description>
      <overviewURL>
         http://mach3.ebphost.net/wsdl/hertz_reserve.wsdl
      </overviewURL>
   </overviewDoc>

   <categoryBag>
      <keyedReference tModelKey="uuid:C1ACF26D-9672-4404-9D70-
39B756E62AB4"
         keyName="uddi-org:types" keyValue="wsdlSpec"/>
   </categoryBag>
</tModel>
```

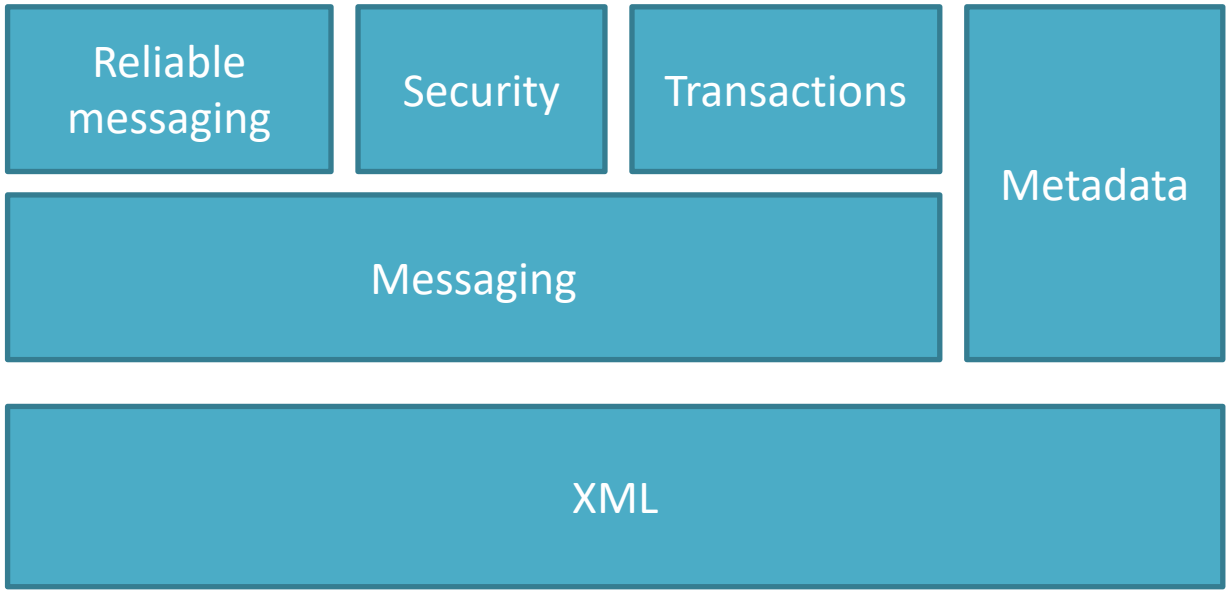
SOA and Security

- Confidentiality, Integrity, Authenticity: XML Encryption, XML Signature.
- Message-Level Security: WS-Security.
- Secure Message Delivery: WS-Addressing, WS-ReliableMessaging.
- Metadata: WS-Policy, WS-SecurityPolicy.
- Trust Management: SAML, WS-Trust, WS-SecureConversation, WS- Federation.
- Public Key Infrastructure: PKCS, PKIX, XKMS

Summary and Context



Summary of Web service standards



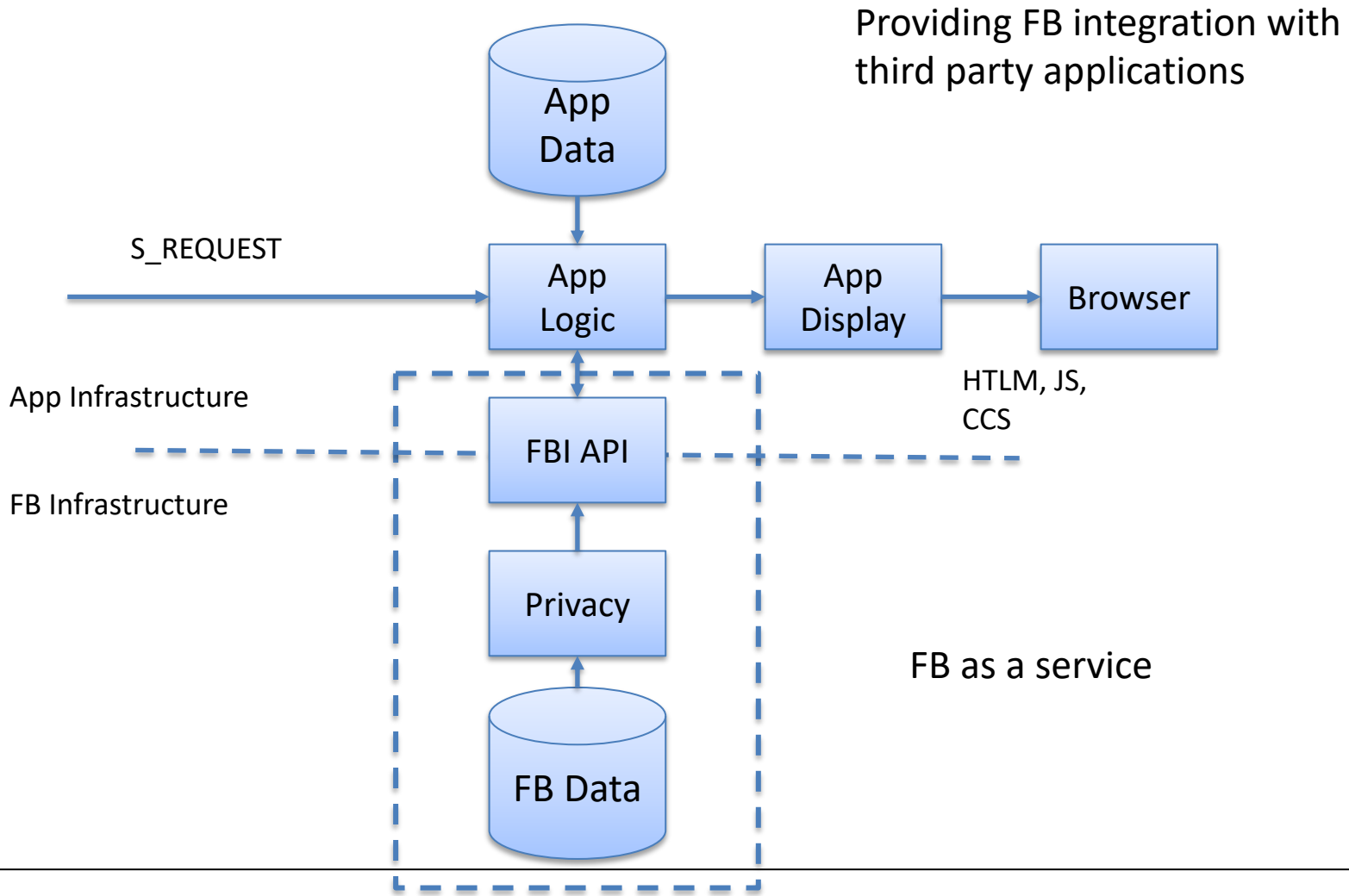
Use of styles and patterns

- Styles restrict focus -> reduce areas of concern
- Easier to use code generation and frameworks (eg: IDL)
- Communication is more efficient
- Combining patterns - multiple benefits of several patterns
  eg: RESTful services

SOAP vs RESTful web services

- Key abstraction of information is a resource, named by a URL
- Resources = sequence of bytes + metadata to interpret the bytes
- Context-free
- Components perform only a small set of well defined methods
- Replication and caching
- Intermediaries

Facebook case-study

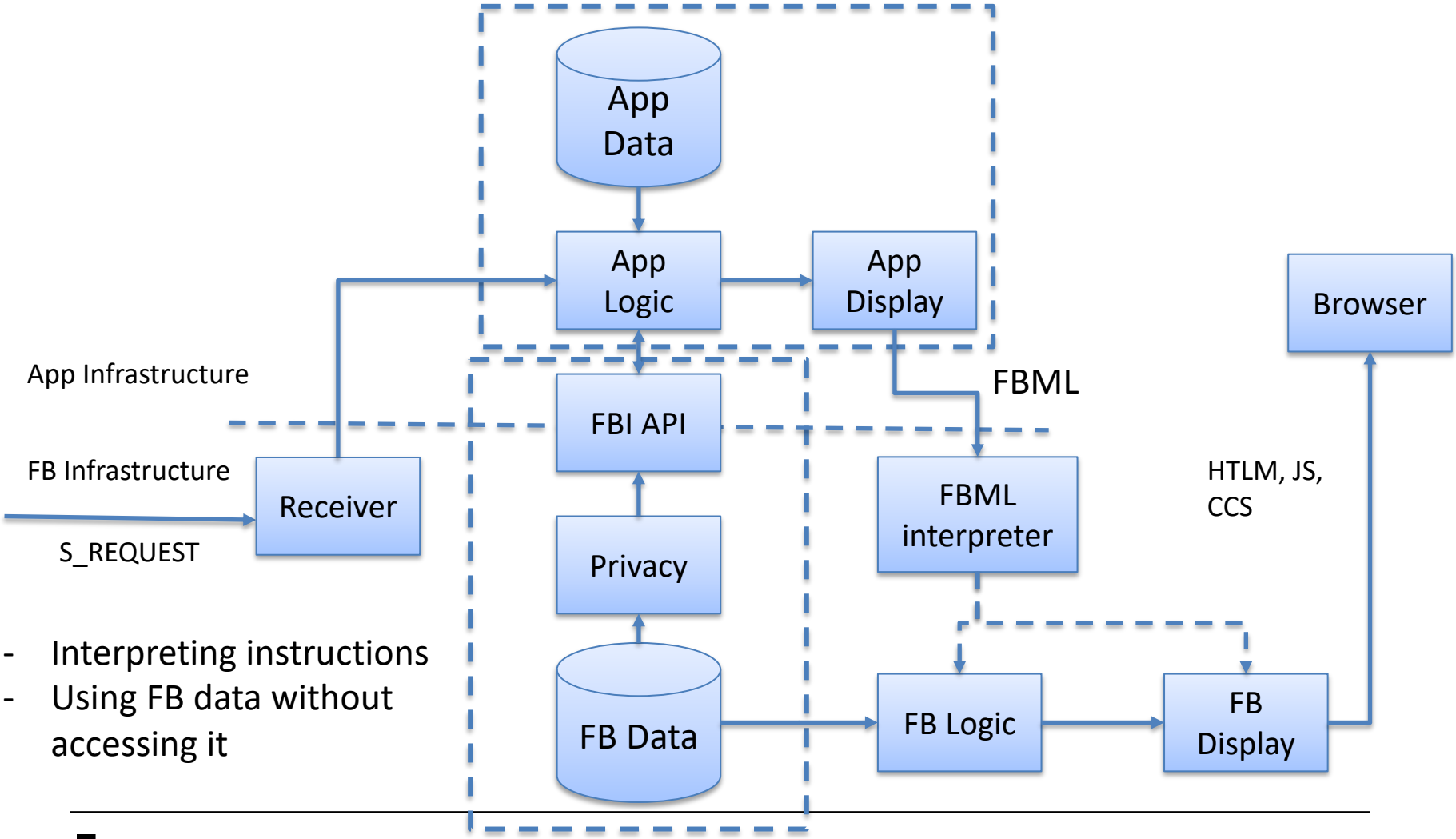


Facebook case-study

- Data-centric application
- FB provides a standard API
- Accessing data -> multiple requests -> overhead
- FQL : a Facebook query language

Facebook case-study

Integrating 3rd party app into the FB platform

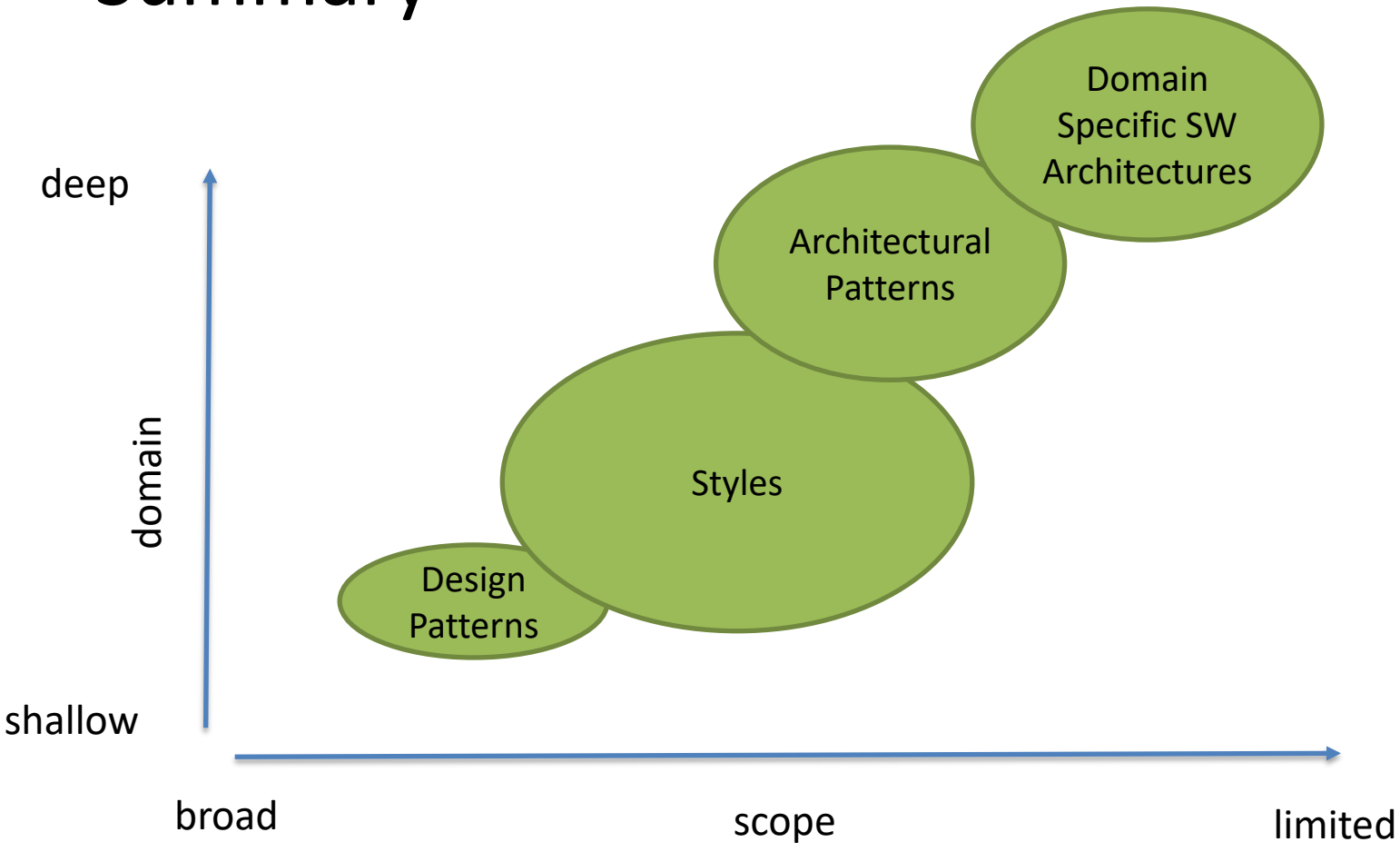


- Interpreting instructions
- Using FB data without accessing it

Unprecedented design

How do we model in new contexts?

Summary


Domain
Specific SW
Architectures
Architectural
Patterns
Styles
Design
Patterns
deep
domain
shallow
broad
scope
limited