Intro

There are several organizations/consortia which influence Web Engineering by producing (quasi-) standards and keeping catalogues.

- W3C, IETF, IANA, ICANN

Top level domains (TLDs) are handled by **ICANN. ICANN** coordinates the **IANA** concerning the DNS.

In 1994 Tim Berners-Lee constituted the **World Wide Web Consortium (W3C)** to regulate the further development of Web technologies through a standardization process.

Well-known **W3C recommendations** are, e.g.,

- HTTP
- HTML, XML, DOM, XSLT, Xpath, XQuery (see Ch. 2)
- CSS
- OWL, RDF, SPARQL (see Semantic Web)
- SOAP, WSDL (for Web Services)
- SVG (for Graphics)

The goal of **Open Standards** is the respectful cooperation between standards organizations. Standards are voluntarily adopted and success is determined by the **market**. Principles – Due process (equity and fairness among participants), Broad consensus, Transparency, Balance, Openness.

The Internet Engineering Task Force (IETF) (1986) is an open standards organization (no formal membership) dealing with the engineering aspects of the Web.

Web applications

Architectural essence of web applications:

- **Client** (a person using a **browser** on a remote node) and
- **Server** (a system **supplying services** for the client).

The service is materialized in **documents** (**web pages**).

The cooperation is controlled by **protocols** for exchanging information over the **Internet**.

A **protocol** is a formal description of message formats and rules for exchanging those messages.

Stages of WA

1. Document Centric Web Sites
- using HTML, static documents,
- updated manually

Examples: static homepages

2. Interactive Web Applications
- using CGI, HTML forms, with pages generated automatically
- in response to user input (read-only)

Examples: news sites, timetable information

3. Transactional Web Applications
- allow user to update underlying content, i.e. data base (read-write)

Examples: conference registration, room booking

4. Workflow-Based Web Applications
- using Web Service, complicated interactions involving several partners (users, companies, public authorities), in a structured collaboration

Examples: e-commerce, e-government

5. Collaborative Web Applications
    - intense cooperation of many persons in collaborative work,
    - generate/edit/manage shared information

   Examples: groupware, wiki, bscw, …

6. Social Web
    - collaboration as part of non-anonymous personal cooperation

   Examples: weblogs, Facebook, etc.

7. Portal-Oriented Web Applications
    - supply information from and interaction with heterogeneous sources;
    - single point of access to handle separate, potentially heterogeneous sources of information

   Examples: search engines, marketplace portals (shopping malls, community portals)

8. Ubiquitous Web Applications
    - customized services anywhere for any device,
    - personalized, location aware, multi-platform

   Examples: device-independent apps

9. Semantic Web

- offer information not only for humans, but in a machine-readable form,
- enabling automatic knowledge management

Examples: ontology-based knowledge sources

Software (particularly web applications!) needs to have high quality, meaning:

- User-friendliness
- Performance
- Reliability
- Scalability
- Security

rSoftware (particularly web applications!) is subject to permanent changes requiring

- Maintainability, evolvability
- Portability
- Interoperability

Web Applications must in particular

- be continuously available (24/7) world-wide
- have short response times
- be easy to use for heterogeneous user groups
- have a pleasant look and feel
- be highly secure
- run on a large number of different platforms (e.g., browsers)
- offer up-to-date data from heterogeneous sources
- support exploratory access
- provide correctly implemented functionality

A discipline for Web engineering thus needs to have

- Clearly defined goals,
- A systematic process,
- A coherent and comprehensive set of tools
- Well-defined methods,
- Reliable quality assurance

WWW and Internet

Basically, the Web is an implementation of Hypertext over the Internet.

Computers (hosts) communicate via local and wide area networks (LANs and WANs) containing also other kind of nodes. For communication, conventions (protocols) are needed.

Protocol: set of rules for communication (e.g., how to establish/destroy a connection or how to exchange data over a connection).

“The Internet”: Network communication technology based on the internet protocol (IP)

Internet distinguishes **two roles** for active components: **Requester** (**client**) and **provider** (**server**).

To enable clients to **find servers**, every server-node has an **address**.

The **address** of a server consists of

- the node’s **IP-address**
- a **port** number
- a transport **protocol** identifier

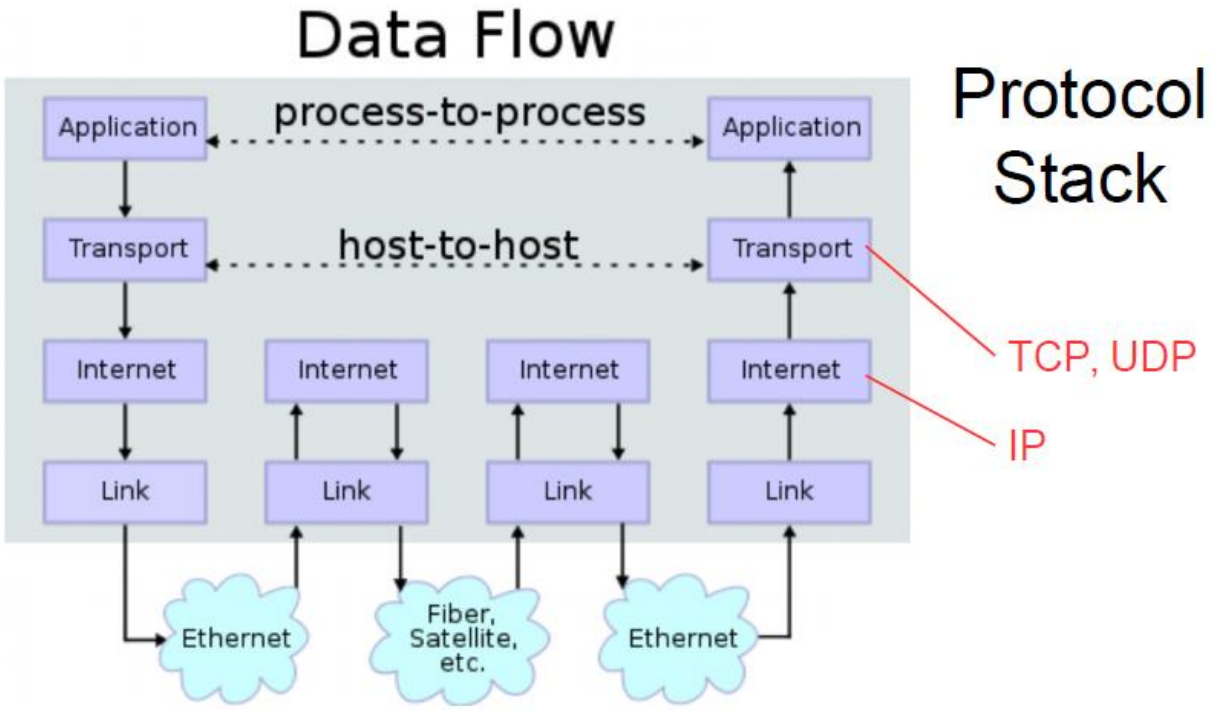
Every node in the Internet has a unique address (called IP address). Formats for this address are **IPv4** (32 bits) and **IPv6** (128 bits).

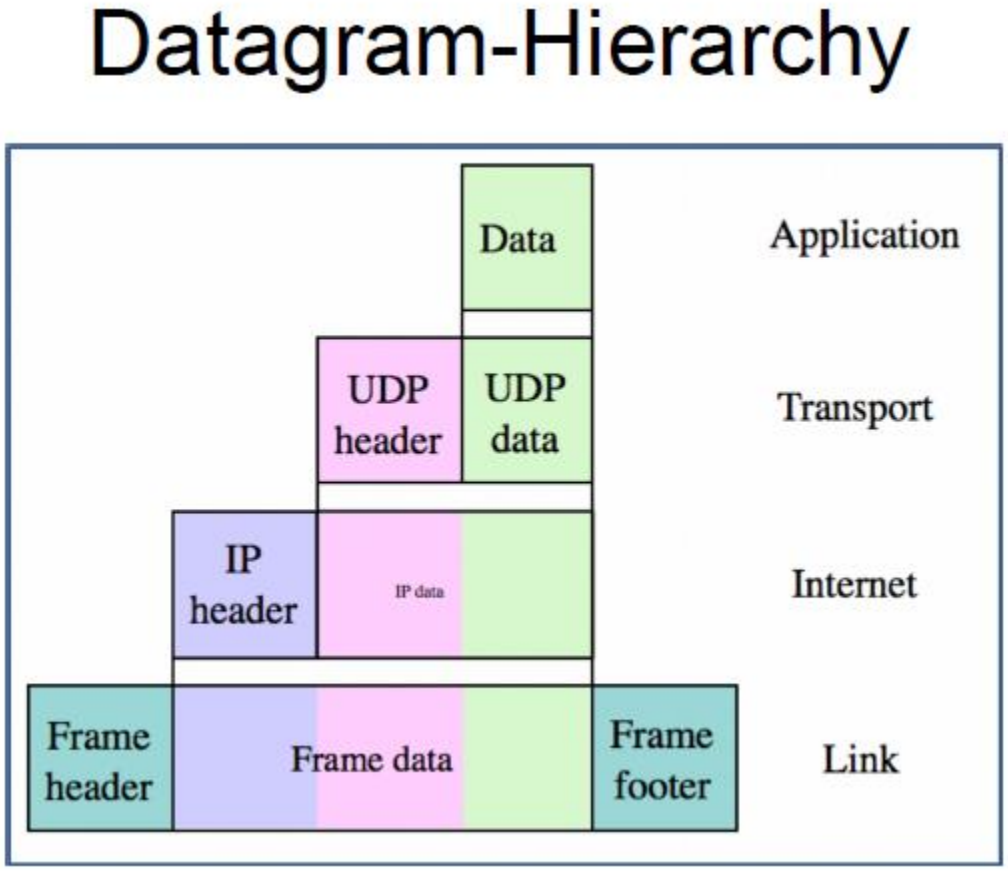
A **domain name server (DNS)** maps known domain name to its corresponding **IP address**.

In general, several server applications run simultaneously on one host. They are distinguished by their port numbers (16 Bits):

- 0-1023: **well-known** ports, used by well-known protocols or services assigned by IANA
- 1024-49151: **registered** ports can be registered with IANA
- 49152-65535: **dynamic**/private ports can be used freely (on one‘s own risk)

Protocol Stack





Internet Protocol (IP)

Internet Protocol (IP) supports exchange of data between nodes.

Properties

- **connection-less**: routes data solely based on address
- **packet-based**: data are split into fixed-size datagrams
- **not reliable**: several kinds of faults may occur

User Datagram Protocol (UDP), Transmission Control Protocol (TCP)

Documents

Hypertext

Definition "**Hypertext**": Network of resources (documents or pages) linked to each other by references ("hyperlinks" or "links") inside the documents.

Hypertext may be

- Static (documents are prepared in advance and exist persistently) or
- Dynamic (documents are computed on the fly - depending on the state of the server - in response to user input).

Web Browsers are programs that are able to **render (visualize) web resources** (provided by a web server) and support navigation via web links.

Web browsers support the use of hypertext over the Internet.

Browsers and Web Servers should deal **gracefully** with non-existent documents or errors in documents.

Markups

Graphical display of text in browsers has led to the need of **augmenting** the pure text for **structuring** the content and **rendering** it for presentation.

Ideally, **structuring** information is **included** in the text, while **rendering** information should be **separated** to allow multiple versions for different platforms.

Markups may have different meanings:

- simple markup (marks a point of the text for navigation)
- descriptive markup (describes the type or special attributes of a content)
- semantic markup (gives meaning to the content)
- procedural markup (controls the way a content is processed)

Markup inside the text:

Text is **annotated** in a way that is distinguishable from the plain text (usually using escape character sequences).

There are several markup languages used in the Web, especially:

- HTML, XHTML, HTML5
- XML

They all have a common ancestor: **SGML**

The hypertext markup language (**HTML**) is a markup language for the Web that allows the markup of text documents using tags. With HTML also **structuring** and **rendering** information for browsers may be **added to the text**.

The Standard Generalized Markup Language (**SGML**)

SGML extends text by elements and entities. SGML uses generalized markups which are

- Declarative (describing structure and text attributes)
- Rigorous (being processable by automated tools).

Elements: text parts marked by opening and closing tags:

<tag> … </tag>

Elements may have **attributes** in their starting tag:

<tag **attr1=value1 … attrk=valuek** > … </tag>

Elements without content may be **self-closing** (or empty):

<tag attr1=value1 … attrk=valuek **/>**

Forbidden characters are encoded as entities:

& < > **(for &, <, >)**

Elements may be nested. Thus, they have a tree-like structure. (Including parent, child, sibling, root, leaf).

Subsets of SGML are described in a **DTD** (document type definition).

A DTD is a **schema describing the structure** of a class of SGML documents.

Conformance to this schema may be **validated**.

XML

Serialization transforms objects and/or data structures into a sequential (**byte stream**) format that can be **stored or exchanged** and reconstructed by another program.

A serialization can be used to **create a semantically equivalent clone** somewhere else.

The standard way of serialization in the Web is **XML**. XML is a format for **storing structured and semi-structured text data.**

XML **does not come with pre-defined markups** or implicit semantics (in contrast to HTML).

XML contains:

- **Elements**, which are the main building blocks.
- **Attributes**, which provide extra information about elements. They are placed inside the starting tag of an element.

  Attributes are added as **name-value pairs**. Each name **may appear only once** in a tag, i.e., they form a mapping map: **name → value**

  All values are **atomic** (strings).

- **Entities**, which are variables used to define common text. Entities are referenced using '& ... ;'.(<)

There are **only five entities** predefined, which are referenced as:

&lt;	&gt;	&amp;	&apos;	&quot;
<	>	&	‘	“

Unicode-characters are also referenced as entities:

&#nnnn; (where n is a hexadecimal number)

Example: &#103B; stands for Greek 'α'

XML-documents start with a declaration and contain **one single outer root** element.

```
<?xml version="1.0" encoding="utf-8"?>
```

Everything between tags that is not markup is called the **content** of the document.

XML-Text should be

- **well formed**, i.e., fulfill **XML syntax**
- **valid,** i.e., conform to a **DTD or an XML Schema**

Well-Formedness Rules

- XML tags are **case sensitive**
- All XML elements must be **balanced** (have closing tag)
- XML elements must be **properly nested**
- XML documents must have **one root element**
- XML **attribute values must be quoted**
- **Comments** are enclosed in <!-- ... -->
- There are **five entity references**: < > & ' "

Validity rules may be specified by

- a **DTD** (from SGML) or
- an **XML Schema** file

A **name space** is a named abstract container for items with different names (i.e., a set, list). Name spaces are used to **avoid collisions** of equally named elements. In **XML** name spaces are used to **disambiguate tags in different vocabularies**.

Name spaces are added to XML elements by the reserved attribute `xmlns` in two forms

- `xmlns` applies to the whole element (incl. children) and denotes the „default namespace“

- `xmlns:prefix` applies only to those (sub)elements, which are explicitly tagged with prefix. (o:order)

XML Schemas

The **validity rules** are given with **schemas**, the most important kinds of schemas being:

- **Document Type Definitions (DTDs)**
- **XML Schema Definitions (XSDs).**

An **XML schema should define:**

- the **tags**
- the **attributes** per element
- the **nesting** of elements
- the **entities**

A **Document Type Definition (DTD)** is a meta-description that defines a document type, i.e., the structure of a class of XML documents.

- DTDs are based on regular expressions.
- This DTD language is inherited from SGML.
- Though it is deprecated, it is still quite often being used.

DTDs define

- the tags by **Element-declarations**

  Elements are declared in the form: **<! Element name content>** The content may be:

  - **EMPTY** no content
  - **ANY** any content without restrictions
  - **#PCDATA** one single text content
  - (**#PCDATA**|elementName1|…|elementNamek)* - text and elements are mixed
  - – or a regular expression – concatenation(a,b,c), choice(a|b|c), option (?), one or more (+), zero or more (*)
- the attributes by **Attlist-declarations**
  - Attributelist Declarations describe the attributes for a given element and have the form:
  - <!Attlist elementName **listOfTriples** >
  - The triples are constructed along: attName type default
  - Defaults are **#REQUIRED, #IMPLIED**, **#FIXED**, or a **value**
  - Example: **<!Attlist cdrom title CDATA #REQUIRED>**
- the nesting of elements by **regular expressions**
- the entities by **Entity declarations**
  - <!Entity name value>
  - Declaration: <!Entity course "Web Engineering">
  - Use: &course;

There are two kinds of text data:

- **CDATA** (character data), which are taken as they are
- **PCDATA** (parsed character data), where tags and entities inside the text are (recursively) also treated as markups

```
<?xml version="1.0"?>
<!DOCTYPE order [
<!ELEMENT order (item+,OrderDate,price)>
<!ATTLIST order OrderID ID #REQUIRED>
<!ELEMENT item (book,cdrom)+>
<!ELEMENT book EMPTY>
<!ATTLIST book isbn CDATA #REQUIRED>
<!ELEMENT cdrom EMPTY>
<!ATTLIST cdrom title CDATA #REQUIRED>
<!ELEMENT OrderDate EMPTY>
<!ATTLIST OrderDate ts CDATA '2003-06-30T00:00:00'>
<!ELEMENT price (#PCDATA)>
]>
```

[after: Kappel2006WE]

Usage - A DTD is associated with a document via the DOCTYPE-declaration, either by a URL or directly as a text:

<!DOCTYPE rootElementName [dtdText]>

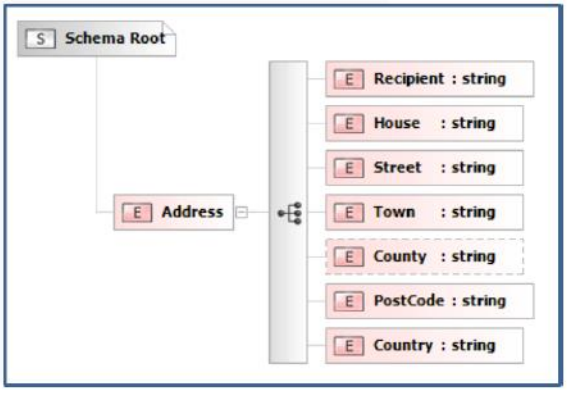
The XML-Schema Language (also called **XSD** = XML Schema Definition) is an alternative to DTDs for defining the structure of XML texts.

Unlike DTD, XSD **allows typing of elements and of attributes**. Typing of elements also defines the nesting structure. XSD **uses XML Syntax** (as opposed to DTDs based on **regular expressions**)

XSD-text contains:

- **element** declarations
- **attribute** declarations
- simple and complex **type definitions**

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Address">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Recipient" type="xs:string" />
        <xs:element name="House" type="xs:string" />
        <xs:element name="Street" type="xs:string" />
        <xs:element name="Town" type="xs:string" />
        <xs:element name="County" type="xs:string" minOccurs="0" />
        <xs:element name="PostCode" type="xs:string" />
        <xs:element name="Country">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:enumeration value="IN" />
              <xs:enumeration value="DE" />
              <xs:enumeration value="ES" />
              <xs:enumeration value="UK" />
              <xs:enumeration value="US" />
            </xs:restriction>
          </xs:simpleType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```



Attributes are added using additional `xs:attribute` elements:

```
<xs:attribute name="orderid" type="xs:string"/>
```

XSD Types - The concept of "**Type**" is used for the declaration of

- the structure of the content of elements and
- the set of the possible values of attributes

- XSD distinguishes **19 simple types** (string, decimal, integer, boolean, date, time, …).
- XSD supplies **25 predefined complex types**.
- XSD allows **construction of new types** using restriction, sequence and union of existing types.

The XSD-text is supplied

- **either as parameter** to a validation engine
- or is **referenced inside the document**.

XSD defines

- the **tags** by `xs:element-Elements`
- the **attributes** per element by `xs:attribute-Elements`
- the **nesting** of elements by types
- XSD is **not meant to define xml-entities**.

XML Processors

To process XML-documents, processors are needed. Processors **must report any violation** of well-formedness rules („**draconian discipline**", not „gracefulness").

XML-processors (**XML-parsers**) are tools that **analyze the markups** and **pass structured information** to an application.

There are 2 main XML parsers:

- **Document Object Model (DOM),** tree-based

DOM is a language-independent convention for representing and interacting with the constituents of XML-documents.

  - The Document Object Model (DOM) is an **application programming interface (API)** for well-formed XML documents.
  - It is a standard for representing XML-documents as **object trees**
  - (DOM-trees) in object-oriented software (e.g. inside browsers).
  - It offers an object-oriented **(tree-like) model of an XML document**.
  - With DOM, programmers can **build documents, traverse their structure, and add, modify, or delete elements and content**.

DOM comes with a parser. Its use is batch-sequential.

Since the DOM tree is a fine-granular model of the document, it may become quite large (O(size)). To process an XML document as a whole may lead to memory size problems, so DOM has problems with scalability.

- **Simple API for XML (SAX),** event-based

While **DOM produces a whole model**, the Simple API for XML **(SAX) is an event-based** parser API. A SAX-parser emits a sequence of events while reading the document. (**It does not create a model**.)

The **memory required** by SAX is only proportional to the **depth of the element nesting** (O(depth)).

Other XML Technologies

- **XQuery** (XML Query Language) is designed to query XML data.
- **XForms** (XML Forms) uses XML to define form data.
- **SVG** (Scalable Vector Graphics) defines graphics in XML format.
- **SMIL** (Synchronized Multimedia Integration Language) is a language for describing audiovisual presentations.
- **MathML** (Mathematical Markup Language) describes mathematical expressions.
- **RDF** (Resource Description Framework) is an XML-based language for describing web resources.

XSL

The **Extensible Stylesheet Language (XSL)** is a suite of three languages intended for transforming and rendering XML-documents:

- the **XML Path Language (XPath)** is used for addressing parts of XML-documents
- the **XSL Transformation (XSL-T)** is used for transforming XML documents
- the **XSL Formatting Objects (XSL-FO)** is used for specifying visual rendering of XML-documents.

XPATH

The XML Path Language (**XPath**) supports **finding items by a traversal of XML-documents**.

XPATH is a language to formally define concrete sets of items (elements, strings, numbers, …) in a given document. **How does it work?**

- XPATH views **XML-documents as trees**.
- In general, the **search goes top-down** from the root to its descendants.
- **Expressions are evaluated with respect to a set of context nodes**. The semantics of an XPath expression is a set of selected items.
- **Predicates may be used** for additional selections.

Operators

operator	meaning
.	self (current item)
/	child
//	descendant
..	parent
@	attribute
[…]	predicate

XSL-T

XSL-T is the **transformation language** in the XMLware technological space. Extensible Stylesheet Language Transformations (XSL-T) transform XML-documents to arbitrary textual representations.

- XSL-T uses a pattern matching approach.

- The XSL-T processor searches the input for pattern occurrences and replaces them by the respective result.
- XPath is used for locating patterns.

Some very cool pictures follow this text in slides. Go look at them (or don't?)

XSL-T transformations may be **referenced inside a document** using processing instructions.

HTML

The Hypertext Markup Language (HTML) is the standard markup language in the Web. Web browsers use it to interpret and render text, images and other material into web pages.

Furthermore, there is an inherent conflict between HTML (graceful) and XML (draconian).

In HTML, there are

- **presentational** markups, which describe the appearance of text
- **hypertext** markups, which describe information for links
- **descriptive** markups, which describe the purpose of text

Browsers interpret HTML gracefully („tag soup") and usually ignore markup elements they don't know.

Different browsers may have also their own extra HTML tags. => HTML pages do not look identically in different browsers using pure HTML.

XHTML

XML version of **HTML**

As an XML-dialect XHTML is **more rigorous** than HTML. (Balanced tags, case-sensitivity, no shorthand features) XHTML documents **can be parsed by standard XML processors** and handled by XML tools. The use of namespaces allows easy extensibility by other XML-based languages.

HTML5

HTML5 is not an SGML dialect. HTML5 is downward compatible to HTML 4.01. But some tags and tag attributes are deprecated (<font>,).

Properties:

- more semantic elements like <section>, <article>, <header> …
- (plug-in independent) embedding of multimedia data by <video>, <audio>, <canvas> … elements
- integration of vector graphics (SVG)
- support of mathematical formulae (MathML)
- barrier freedom
- handling of syntax errors

CSS

In HTML, **markups are also used for describing rendering information**. This is considered bad style today. **Rendering information should be supplied separately** from semantics (to support different (maybe unexpected) rendering contexts).

Cascading Style Sheets (CSS) is a language to define the presentation of a document written in a markup language like HTML, XHTML or any XML language in general.

CSS supports the **separation of content and presentation** and allows usage of special presentation facilities on special platforms. CSS allows to **define the presentation** of the document on **different devices.**

CSS supports rendering of the **same page in different styles** as well as of **several pages in same style**.

CSS refers to typographic characteristics like **font, size, color, emphasis, borders, spacing, and text alignment**.

A style sheet is a list of rules. Each rule consists of some selectors and a declaration block containing a list of declarations. A declaration consists of a property, a colon, and a value, terminated by a semicolon.

Priority scheme for CSS sources:

– Author styles

- Inline styles, style information on a single element, specified using the style attribute
- Embedded styles, blocks of CSS information inside the HTML-text (<style>… </style>)
- External style sheets, i.e., a referenced separate CSS file

– User styles, a local CSS file in a browser, which overrides all documents styles

Communication

Addresses

The client has to

- locate the server using its address (via the domain name service) and to
- Identify the resources (e.g. a file) in its file system.

A **Uniform Resource Name (URN)** unambiguously **identifies** a given resource (e.g., document).

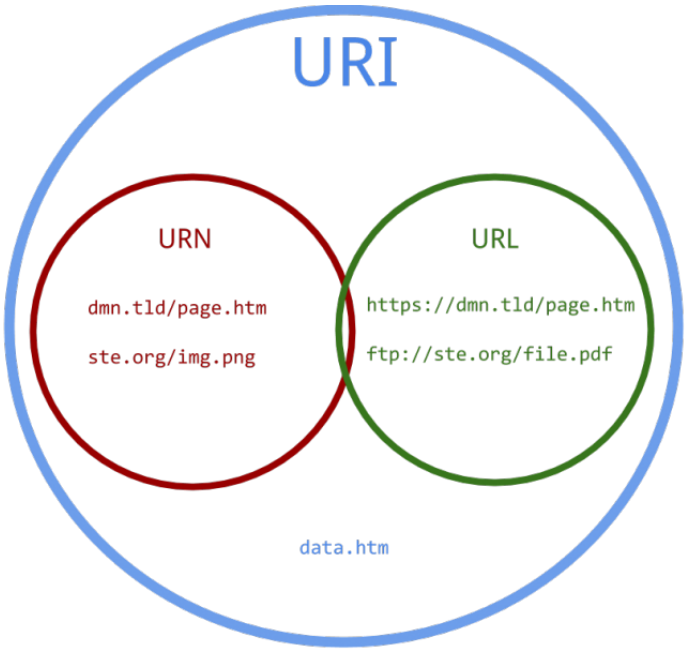
A **Uniform Resource Locator (URL)** unambiguously **describes the access** to a given resource (e.g., document) using the domain name system (DNS).

A **Uniform Resource Identifier (URI)** may be a **URL** or a **URN**.

Examples (URLs):

– http://penguin2.uni-koblenz.de/ebert/memo

–http://localhost/tmp/memo



A URL consists of

- the protocol
- the node's name
- the port number (optional, default:80)
- the document path (name and location of the resource on the host)

- Parameters (optional, following '?')
- an anchor (optional, e.g., #anch1)

Example:

http://www.uni-koblenz-landau.de:80/koblenz/fb4/institute/IST/we#content?fail=true

Document formats

With the shift towards hypermedia, many non-text types of media are transferred.

MIME (Multipurpose Internet Mail Extensions) is an approach to define specific types of media.

A (full) MIME type has two parts:

- a type
- a subtype

IANA (www.iana.org) keeps a registry of media types and encodings. MIME types – application, audio, example, image, message, model, multipart, text, video.

HTTP

Communication in the Web is done using the **Hypertext Transfer Protocol (HTTP)**. HTTP is a **simple protocol** on the **application layer** of the Internet supporting the exchange of documents between the **client and the server**.

- HTTP is a **request-response protocol**. Thus it is alternating and asymmetric.
- HTTP is **state-less**. It delivers data as byte streams.
- HTTP/1.1 usually works **on top of TCP** and **reuses a connection multiple times** (to download further elements for a page).

The HTTP client initiates a request, which establishes a TCP connection to port 80 on a host given by a URL. The request consists of:

- a request line
- headers
- an empty line
- message body

(all ending with CR/LF).

GET Request, Format:

GET <location of the document> HTTP/1.1

GET Request, Example:

GET uni-koblenz.de/fb4/index.html HTTP/1.1

Main request verbs (should be supported):

- **GET** (requests a representation of the specified resource)
- **HEAD** (like response, but requests only the header, i.e., the meta information)
- **OPTIONS** (returns the HTTP methods supported for this URL)
- **POST** (submits data, e.g., from a form. This results in an update of the resource on the server)

Further request verbs (optional):

- **PUT** (uploads a representation of a specified resource)

- **DELETE** (deletes a resource)
- **TRACE** (echos a request)
- **CONNECT** (converts connection to support HTTPS)
- **PATCH** (applies partial modifications to resource)

The use of HTTP commands is **restricted by a Guidance Rule:**

HEAD, GET, OPTIONS and TRACE should be **safe**, and are intended only for **information retrieval** and **should not change the state of the server**.

This rule is not enforced.

The response consists of:

- a status information
- response headers
- an empty line
- message body

(All ending with CR/LF)

To allow Java (or JavaScript) programs to use **HTTP for data transfer**, an API is provided: **XMLHttpRequest (XHR).** XHR supports HTTP requests to the server, and it supports integration of response data into the DOM-tree.

Same origin policy applies: a web browser permits scripts contained in a web page to access data in another web page only if both web pages have the same origin (defined as a combination of URI scheme, hostname, and port number).

Sessions

The HTTP-protocol keeps no state. Though TCP is **connection-oriented**, **HTTP is not aware of it.**

Interactive Web Applications must be able to distinguish requests by multiple users.

A **session** is a coherent sequence of related HTTP requests between a specific user and a server within a specific time frame.

Approaches to keep track of sessions:

- URL rewriting
- Cookies

URL rewriting transmits all session-relevant data as **parameters in the URL** (XYZ should be hard to guess (*)):

```
  http://host/application/page.ext?SessionID=XYZ
or
  http://host/application/XYZ/page.ext
```

Drawbacks of URL Rewriting

- the **URL can become messy and error-prone**
- the approach can be **unusable due to length-limits of URLs** on some systems
- the **URLs inside the documents have to be adapted dynamically** to include the session ID

- **It is inherently insecure** (unless using https, or unless no content at web-space should ever be protected).

Cookies are small text files used to **store server information** (e.g., a session ID) **on the client node** as name-value pairs. They allow to make the **session information at the server's side accessible by an ID**.

The **server transmits the cookie to the client in the heading of the HTTP response**.

The browser transmits the cookie to the respective server with every request.

There are

- session cookies (kept in browser)
- permanent cookies (stored on disk)

Client

Since the higher stages of the Web require dynamic, **interactive pages**, new technologies are needed to "**bring life into HTML**". Technologies:

- Helper programs / plugins
- Applets
- **Scripting languages**

Helper programs: applications that add functionality to browsers.

Plug-in: helper program permanently installed in the browser.

Scripting languages

Scripting language: programming language that **controls the execution of other applications**.

Examples:

- Mainframe job control languages (**JCLs**)
- **Unix / Linux-shells**
- **JavaScript** (for **client side** scripting in the Web)
- **PHP** (for **server side** scripting in the Web)

Properties of Scripts:

- (Usually) **interpreted** whereas their controlled applications are (usually) **compiled**.
- May be **embedded** in the applications they control.

Client-side scripts: embedded in HTML-page using tag **<script>.**

Browser interprets and executes the code. Allows to make the **content alive** depending on, for example: **user input**, **environment**, **special conditions**.

Server-side scripts: used to produce HTML-text (which may again contain client-side scripts). Executed by script interpreter on the server.

JavaScript

JavaScript is used to support (for example):

- user interaction
- browser control
- asynchronous communication with the server
- document manipulation
- form validation
- dynamic changes of content

JavaScript is

- **weakly typed** (dynamically typed) (unlike Java),
- **prototype-based** (unlike object-oriented languages), and
- Supporting **first-class functions** (like functional languages).

JavaScript fragments are included in HTML using **<script>** element. Script text may be **embedded** or **imported**.

JavaScript fragments are executed by the browser **immediately when they occur** during parsing of the HTML-page. The browser acts as a **JavaScript interpreter**, (using a special JavaScript engine, e.g. SpiderMonkey), i.e., the statements are commands to the browser. Function definitions **may be called anywhere**.

JavaScript is:

- inspired by **C/C++/Java** imperative style and influenced by Self and Scheme
- **object-based (not object-oriented)** with some **functional features** (including closures)

No built-in IO-support in JavaScript. Simple interaction support is given by boxes:

- **alert-boxes** expect an ”OK“ by the user
- **confirm-boxes** expect an ”OK“ vs. ”cancel“ decision
- **prompt boxes** expect a string

Typing is dynamic in JavaScript, so the **variables don’t have a type**, but the values. Variables may get **reassigned with different types**.

This leads to **more flexibility** at the expense of **less quality assurance** at compile time (catch typing errors).

JavaScript types are

- **primitive types** (numbers, Booleans)
- **Strings**
- **Arrays** (zero-based)
- **Objects**
- **Functions**
- **Undefined** (null)

An **object** is a collection of properties as an “**associative array**“. Thus, **properties** are **name-value pairs**, whose values **may be of any type**.

Objects may be created by a **constructor**

```
var obj = new Object();
```

Objects may **refer to another object** with their own **prototype** property.

Thereby they get access to the other object's properties:

```
obj.prototype = person;
```

The prototype-property mimics **"inheritance" by delegation**.

The concept of objects in JavaScript differs from that of object-oriented languages.

JavaScript is **prototype-based**. There are no classes. **Differentiation**:

- **class-based**: classes define structure and behavior, **objects are instances having a state**
- **prototype-based**: instances maybe **constructed by cloning**

Object categories in JavaScript are

- **Native** (built-in) objects, supplied by JavaScript: **String, Array, Image, Date, etc.**
- **Host** objects, supplied by the browser environment: **window, document, form, etc.**
- **User-defined** objects: **person, library, book, etc.**

Functions are also objects.

- They have **data properties and methods**.
- Functions **may be nested**, i.e., they may have functions as properties which are called methods.
- Functions **may be returned as values**. They are handled as **closures**.

Functions are closures, i.e., they **carry the environment they were created in**. The closure of a function is an extension of the pure function by its **environment**. The environment contains all referenced non-local variables.

Since JavaScript is an interpreted language, **it is possible to create/read JavaScript text at runtime** and execute it on-the-fly. Consequently, **static security checks are hampered**.

In the Web, the runtime **environment of JavaScript is the browser**. Predefined environment variables allow access to:

- the window and its frames (**window**)
- the browser (**navigator**)
- the screen (**screen**)
- the url history (**history**)
- the current url (**location**)

HTML-text of current page is accessible as DOM tree in **window.document.** Can read and modify the web page using DOM objects and their respective methods. Thereby, can use JavaScript e.g. to load new content, change layout and style, introduce interactive content, validate inputs.

JSON

JavaScript supplies a nice short-hand **syntax for objects and arrays: JSON** (JavaScript Object Notation).

JavaScript program can use built-in eval()-function to produce native JavaScript objects from JSON-Text.

Advantage of JSON: For JavaScript-applications JSON is **faster and easier than XML.** Often used as light-weight notation for structured data, as a **substitute for XML**.

Structure of JSON:

- data are in **name / value pairs**
- data are **separated by commas**
- curly braces hold **objects**
- square brackets hold **arrays**

AJAX

The following technologies are usually **used together to support dynamic communication** between client and web server:

- **HTML** (or XHTML) and CSS for presentation
- the Document Object Model (**DOM**) for dynamic display of and interaction with the page
- **XML** for the interchange of data (**or JSON**)
- **XSLT** for data manipulation
- the **XMLHttpRequest** object for asynchronous communication
- **JavaScript** to bring these technologies together

This ”technological subspace“ has been coined **AJAX = Asynchronous JavaScript and XML**.

JAVA Applets

Applets: Small applications for a specific task, started from the web page and loaded and executed by the browser. Written in a compiled language.

Execution of Java GUI Applications by web browser JVM via Java-Plugin. Embedding in (X)HTML by <applet> tag

- Lifecycle: Constructor, init, (start, paint+, stop)+ destroy
- Parameter passing via <param> tags
- Utilizes the AWT or Swing frameworks

Properties of Java Applets:

- they are not programs (no main-method)
- input and output via the applet’s GUI
- event-driven
- executed in a sandbox

Java applets are **very obsolete** nowadays…

Server

Web server (HTTP server): program that **delivers web pages** to web clients - usually as HTML documents with additional files like style sheets, JavaScript texts, and/or images.

Web servers host web sites. May also receive data from client via HTTP.

Web server translates URL to

- either a file path (in case of static pages)
- or a program path (in case of dynamic pages)

The Apache HTTP Server

Apache: component-based structure. May be further extended by so-called modules.

Role of the modules in Apache: implement / override / extend the functionality of Apache web server.

All modules have the same interface to the core of the server. Modules do not interact directly with one another.

Server side scripting

Server side scripting: behavior of server may be programmed (scripted) for a given application. Can support e.g.:

- dynamic web pages (dependent on database)

- database access
- interaction support

Server-Side Solutions:

- SSI (**Server Side Includes**)
  - (few) SSI statements in HTML text
  - Construction of result page by combination of different parts
- CGI (**Common Gateway Interface**)
  - Calls programs via web server upon requests to specific URLs
  - Query parameters are passed to the program
  - Program calculates result page as standard output
- Script Languages (**PHP**, Perl, Python, Ruby, JavaScript)
  - Script interpreters contained in web server (server module)
  - Mixture of script code, HTML, layout, and content in script programs

Server-Side on JAVA

– (CGI programs in Java)

– **Java Servlets**

– **JSP** - Java Server Pages

– **JSF** - Java Server Faces

Server-Side Include (SSI): simple preprocessor mechanism to create HTML page on the fly.

The preprocessed page is sent as a response. SSI is activated in case the extension of the file is shtml.

SSI: controlled by pseudo comments, i.e., comment statements that contain additional instructions inside.

Includes help to insert into the page e.g.:

- files
- outputs of a program (e.g. CGI-programs, cf later)
- values of system variables

Includes support separation of concerns by separating common parts (page headers, page footers, navigation) from varying content.

Simple commands are e.g.:

- include (to insert text; parameter virtual specifies file to be included relative to domain root: e.g. html page, text file, or CGI script which produces the text when executed)
- exec (to execute CGI-scripts or commands)
- echo (to insert values of environment variables)

Structuring commands (control directives) are e.g.:

- if
- else
- endif

Common Gateway Interface (CGI): convention for cooperation between web servers and application programs (on the server side). Delegates generation of web content to executable files (called CGI-programs or CGI-scripts). CGI-script usually runs on the same host and is kept in a directory called cgi-bin.

Control flow:

- Server sets environment variables
- Server starts script (possibly with parameters)
- Server supplies data (e.g., POST content) via stdin

- Server reads data via stdout until script terminates

A CGI-script is addressed by (the path suffix of) a URL. It reads parameters via stdin, processes them, and creates an HTML document via stdout.

Additional information is passed via environment variables. Thus, the ”Interface“ is primarily a list of environment variables and two stream ports.

CGI does not scale well, since an independent process is started for each incoming request.

PHP

The PHP Hypertext PreProcessor (PHP) is a scripting language on the server side.

The script fragments are embedded in HTML and executed by a script interpreter on the server.

Any text outside of PHP-elements is sent to the client via stdout without being interpreted by PHP.

But it can be controlled with PHP constructs like loops.

An interpreted PHP-File is (usually) plain HTML. One can either use the processing instruction:

```
<?php ... ?>
```

… or a script-element...

```
<script language="php"> ... </script>
```

… to embed PHP-fragments into HTML.

PHP:

- **a scripting language** (usually interpreted)
- inspired by C / C++ / Java
- **object-oriented**
- **has closures** since 5.3 (2009)
- supports the use of **databases** by function calls
- owns an **extensive API** (Standard PHP Library)

PHP: object-oriented language. Supports

- **classes** (with constructors, attributes, methods),
- **specialization**,
- **interfaces**, and
- public / private / protected modifiers for **visibility control**

Objects of the same class may access each other‘s members.

Functions are **first-class entities**. **Closures** are supported.

PHP supports **easy access to relational databases** by API-functions.

JAVA Servlets

Servlets are a **Java-based alternative to PHP** programs.

Servlets are Java objects that run in a special runtime environment (servlet container) inside the web server. The container can be run as a sandbox.

A servlet serves a client’s request and returns a response. The generated content is usually HTML.

Servlets **can maintain their semantic state in session variables** across many server transactions (by using HTTP cookies or URL rewriting).

Activation by web server (servlet container) via specific URLs definition of URLs in container configuration

The servlet container

- **maps a URL to a particular servlet**,
- ensures that the URL requester has the **correct access rights**, and
- **manages the lifecycle** of the servlet.

Each request is handled by a **separate Java thread** within the web server.

JAVA Server Pages

Drawbacks of Servlets

- Intermixture of program code, content, and layout
- HTML only contained in Java Strings (no syntax checks)

→ **Java Server Pages (JSP)** as extension of the servlet technology

- Layout in HTML
- Special tags (JSP actions) for dynamic parts
- Java code in so-called „scriptlets“

Java Server Pages (JSP) are **similar to PHP pages**, in that they add server-side code to an HTML page, but they use Java as programming language.

JSPs are compiled into Java servlets at runtime. It is recompiled only in case a modification occurred.

A scriptlet is a piece of Java-code embedded in the HTML-page:

<% `...` **%>** encloses a JSP scriptlet.

Java Server Faces

Reevaluate how useful the next text is. As it probably isn't.

Drawbacks of JSPs:

- too much „infrastructure“ in code
- no event model
- rather complex code

→ extension **of JSP: Java Server Faces**

UI framework for web applications

- Standardized UI components
- Input validation
- Event handling
- Control via Bean components
- „Rendering“ of components into XHTML and JavaScript
- JSP-Code is simplified by „**facelets**“

Lifecycle:

– Sequence of actions while handling a Java Server Faces request

Coarse separation in

- Execute phase
  - construction of component tree
  - validation of input values
  - event handling
- Render phase - transformation of component tree into (X)HTML

Initial request of a JSF page slightly different to subsequent requests (postback requests)

Event sources:

– JSF standard events, e.g. valueChangeEvent or application specific events

- Organization in Event queue
- Insertion of user defined events possible
- Processing after/between specific phases

– Phase-ID of an event determines execution, e.g.

- PhaseId.INVOKE_APPLICATION, Phaseid.ANY_PHASE
- Possible interruption/termination of lifecycle by event handler
- Phase can be declared in to observe phase changes

Restore View

– Construction or Recall of the component tree for the page

– Tree structure determined by XHTML elements

Apply Request Values

– Evaluation of request parameters (URL or POST parameters, cookies, values of input components …)

– Call setters of component attributes

– Call of methods to parse input Strings into Java objects

– Optional:

Queueing of ActionEvents by ActionSource Components

Premature validation and execution of ValueChangeEvents by components with flag

Process Validations

– Validation of attribute values

– Call of method

– Registration of specific validators via XHTML page

Alternatively: Bean Validation (later...)

• Update Model Values

– All request parameters were processed, component attributes in

component tree were validated syntactically and semantically

– Call of processUpdates() method for all components

– Call of setter methods of connected backing beans by

Components

Invoke Application

– Process of events with PhaseId

INVOKE_APPLICATION

– Execution of application specific behavior

– Make decision on next page

– Transition to the Render Phase

Render Response

– Construction of component tree of the following page (only when

page is changed)

– Transformation of the component tree by a RenderKit

– Standard JSF component behavior:

Transformation into HTML by call of the encode() method

– Composition of the output

• Static parts and templates

• Dynamic parts

• Call of getter methods of backing beans

• Formatting of values by Converter

Processes and Requirements

Main Activities (Software Life Cycle)

- requirements engineering
- modeling
- architectural design
- detailed design (technology aware)
- implementation
- operation and maintenance

Parallel activities

- configuration management
- quality assurance (especially testing)
- project management
- process management

Challenges in Web Engineering:

- **Multidisciplinary** – Many disciplines work together
- **Unavailability of Stakeholders** - difficult to find suitable representatives with realistic requirements
- **Volatility of Requirements and Constraints** – highly dynamic environment
- **Unpredictable Operational Environment** - execution environment props (bandwidth, browsers, …) are hard to predict.
- **Impact of Legacy Systems** - Integration of existing software may be necessary for economic reasons, but possibly not compliant to the architecture.
- **Significance of Quality Aspects** - Besides the standard quality properties, performance, security, availability and usability are indispensable
- **Quality of Content** - Content has to be accurate, objective, credible, relevant, up-to-date, complete, and clear.
- **Developer Inexperience** - Many of the underlying technologies are still fairly new.
- **Firm Delivery Dates** - There usually are fixed schedules that have to be kept.

Processes

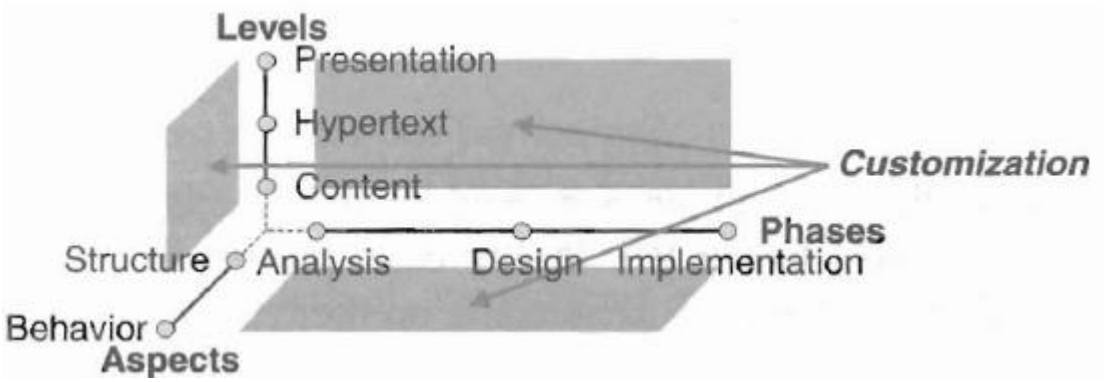
Web Application development is usually a process that is executed by many persons cooperatively. Thus the process has to be organized and controlled.

The process should (at least) be defined to such an extent that it is manageable and repeatable (CMM-2).

Development (and maintenance) processes have

- a **logical structure**, which defines the activities to be done and
- an **organizational structure**, which defines the coordination of the cooperative work

LOGICAL STRUCTURE



The development activities for WAs can be structured according to

- the levels (**concerns**) to be described
- the aspects (**viewpoints**) they subsume
- the phases (**abstraction levels**) they belong to

Levels (Concerns)

- **content**, the information and application logic underneath
- **hypertext**, the structuring of the content into nodes and links
- **presentation**, the user interface and page layout

The levels have to be mapped to each other.

Aspects (Viewpoints)

Like for classical applications, we have to define and develop the **structure and the dynamic behavior**.

- **structural** description subsume object-relationship views,

e.g., class / object / package diagrams

- **behavioral** descriptions subsume controlflow / dataflow and state-transitions views,

e.g., activity diagrams, state machines.

Phases (abstraction levels)

The main **phases** for software development are

- **requirements engineering**
- **design**, i.e., modeling and architecture engineering
- **implementation and testing**

Along these phases the descriptions become more and more concrete (less abstract).

ORGANIZATIONAL STRUCTURE

The **metaphor of engineering** implies:

- definition of **requirements**
- building of **abstract models**
- **implementation** along these descriptions
- **quality assurance**

To handle the development of the three dimensions, effective processes have to be defined, to:

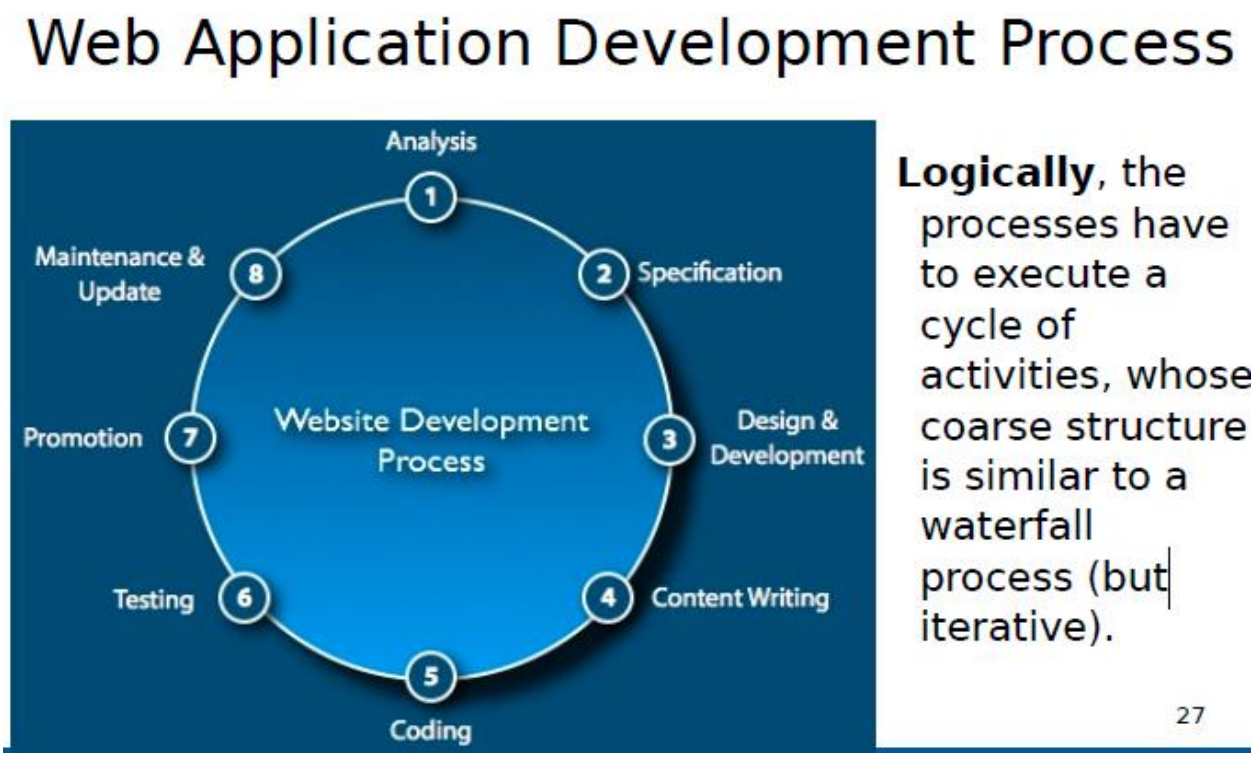
- help the team understand what it has to do
- coordinate the work of the different developers
- support the manager to control the actions.

Process Models (software processes) describe the **relevant tasks for software development** and their organization. A good process model helps to:

- **make experience transferrable**
- **reduce risks**, since the process becomes planable

Needed: **good balance between standardization and room for creativity**. Examples are:

- Waterfall models
- V-models (e.g., V-model XT)
- incremental models (e.g., Rational Unified Model)
- agile models



Requirements for web applications

Requirements engineering is the most important phase of any software development process.

The **more requirements** are defined and documented up front, the more likely the web site will be **built properly**.

Web applications require an even more extensive requirements engineering process due to the number of stakeholders involved and the diversity of requirements.

A **requirement** describes a service to be provided or a property to be met by a system. The requirements list summarizes all requirements agreed between the contractor and the customer.

There are **several categories of requirements** needed for Web Applications besides:

- functional requirements - specify the services expected
- non-functional requirements - deal with quality, constraints, and technology aspects

Namely:

- **business** requirements
- **content and navigation** requirements
- **technical** requirements

FUNCTIONAL REQUIREMENTS

The functions the web site must fulfill to support the business objectives need to be clearly defined:

- **Personalization Functions**:
  - Registration and sign-in, welcome back, enabling a user to sign up for newsletters, etc…
- **Transactional Functions**:
  - shopping cart, integration with backend systems like financial software, reviewing and selecting products
- **Security Functions**:
  - creating a secure registration page, ensuring passwords are secure, secure shopping cart payment information

NONFUNCTIONAL REQUIREMENTS

More general non-functional requirements are added:

- **Look and feel** standards
- **Usability** guidelines
- **Legal** and security guidelines / issues
- **Internationalization**
- **Accessibility** (for handicapped)

BUSINESS REQUIREMENTS

Business requirements document the business functions that the web site will support, e.g.,

- The process to **create product** information for the web site
- The process to **sell products** from start to finish
- The process to **order products** or materials from suppliers
- The **customer service** processes

CONTENT AND NAVIGATION REQUIREMENTS

It is necessary to capture all the types of content that will be used on the site:

- What is the purpose of the content?
- How is it related to other content?
- In what format is it available (e.g., Word, Excel, PDF, graphic)?
- Who owns it?
- Who maintains and updates it?
- Is there any workflow associated with it?
- How often is it updated?
- Is this content static or dynamic?

TEHNICAL REQUIREMENTS

Technical requirements relevant to the stability of a web site include:

- Expected volumes of users
- Expected peak periods of use
- Types of content that will create high load (like video and audio files)
- Security requirements (e.g. access restrictions)
- Performance requirements (e.g., page load time)
- Operating availability (24/7 availability standard for WAs; further requirements e.g. < 1 hour downtime / year); maintenance periods
- Support requirements
- Database sizes (for storing content)
- Types of browsers to support, including browser resolutions

Activities in Requirements Engineering

Requirements Engineering (RE) is the process of

- eliciting,
- specifying,
- validating,
- prototyping, and
- Managing requirements for software.

It is an iterative and cooperative process. Generally, RE can be done as in classical Software Engineering, but there are WA-specific issues.

The set of needed stakeholders depends on the **business case**. At least one person has to be included for every stakeholder role.

Stakeholders having direct or indirect interest in a WA include:

- customers,
- users,
- content authors,
- web page designers
- domain experts,
- usability experts,
- marketing professionals,
- developers

For WAs, the objectives and **expectations** of stakeholders are often **quite diverse and possibly conflicting**:

- **capabilities vs. budget**
- **project schedule vs. quality**
- **development technology vs. developers skills**

Challenges for WAs:

- find representative success-critical stakeholders
- reconcile conflicting requirements
- do it under time pressure

Since WAs are highly interactive, means for concretizing expectations of the UI and the interaction are needed.

Requirements must be accompanied by sketches and prototypes (simulations).

IKIWISI =”I know it when I see it. “

Mockups and prototypes help to get early response from the users (**participative development**).

A mockup is an **almost original model** of the **user interface**. It is a prototype if it also provides at least part of the functionality of the system and enables validation of the design.

During the project the **requirements specifications evolve** due to the permanent changes in agile processes.

A **prototype** is a (usually preliminary) **light and incomplete version** of the application that is used to ease discussion about requirements.

Prototypes may also be used for validation.

Most important **Requirements Artifacts** are

- the vision - text
- the requirements lists - structured sentence lists
- glossary - dictionary

– a (conceptual) domain model– UML (class diagram)
– use-case diagrams - UML

The **agreed requirements** have to be documented in a degree of detail and formality that is appropriate for the project context. In **agile processes** requirements are often **derived dynamically** from user stories.

Requirements elicitation

At least **one representative of each user class** should be interviewed.

This affords

– identification of the right persons
– preparation of the interview
– interview execution
– result documentation by a protocol

The **requirements are the result of an iterative learning** and consensus-building process.

So **communication** with all stakeholder (roles) **is essential**.

Requirements must be **validated and verified**.

Modeling

Models represent a solid starting point for the implementation of Web Applications.

The set of models can be structured according to

– the aspect they model
– the phase they belong to
– the level to be described

Main models needed:

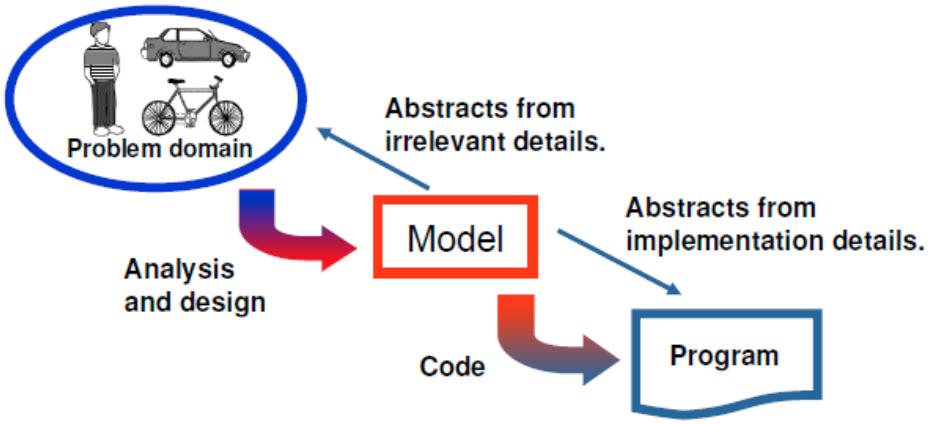
– content model (information)
– hypertext model (navigation)
– presentation model (look-and-feel)
– application logic model (like for non-web applications)

Furthermore models are a basis for model-based development and code generation.

For all three dimensions, the inclusion of context information leads to customization (tailoring).

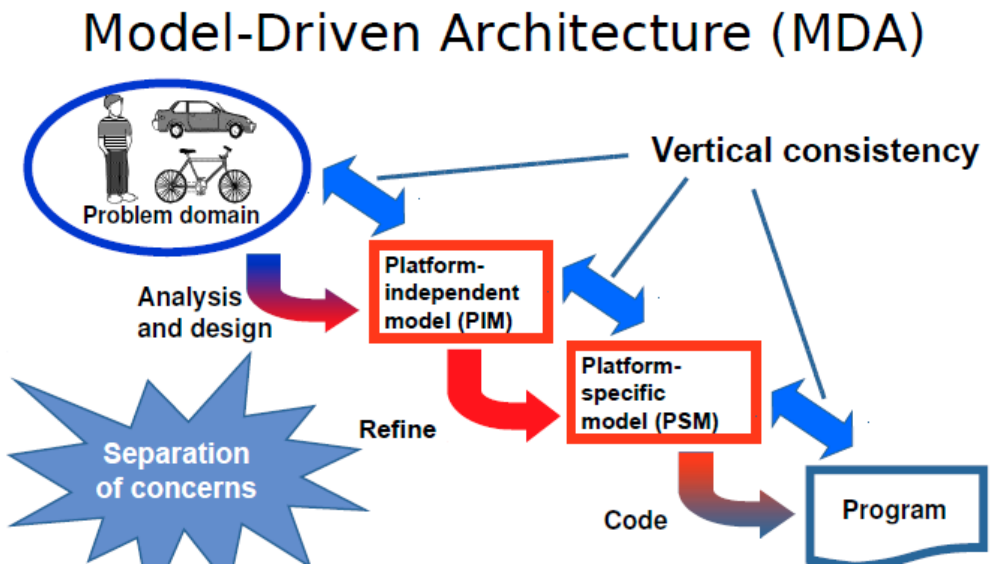
Customization considers the context in which the web application is executed, including:

– user's preferences
– device characteristics
– bandwidth restrictions
– used software (e.g. browser)



Modeling Objectives:

- Detailed specification
  - as base for automatic model transformation
  - as input for realization / coding
- Reducing the complexity
- Documentation of design decisions
- Readable description of system structure and functionality
- Visualization of relevant system aspects



Layered model stack

– Different hypertext structure on top of the same content
– Different presentation models on top of the same hypertext model
• Different modeling objectives
– Content: no redundancy
– Hypertext: planned redundancy, i.e., information may be retrieved via different navigation paths

Modeling for Web Apps

For the domain-specific and functional properties of web applications, all general modeling languages are used, e.g.,

- **UML-based**: e.g. class diagrams, state machines, activity diagrams, use case diagrams, sequence diagrams, component diagrams
- **Variants**: e.g. BPMN diagrams, feature trees, block diagrams, Matlab/Simulink diagrams
- **Domain-specific**: e.g. UWE (cf. this chapter); could also be newly invented

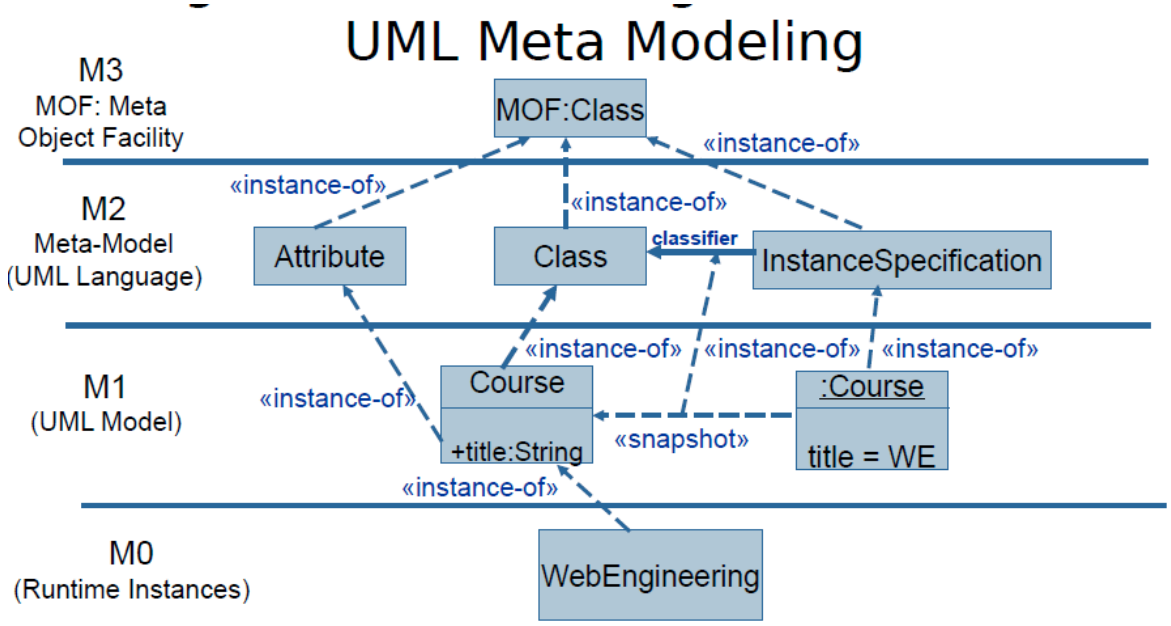
UWE: UML-based Web Engineering, 2002 object-oriented, UML-based

UWE consists of:

- UML-based domain specific modeling language,
- model-driven methodology,
- tool support for design and code-generation

UWE supplies

- a **UML profile** (version 2.1): lightweight extension using stereotypes
- a **meta model**
- a **model-driven development** process
- **tool support** for editing UWE models (magicUWE, plugin for magicDraw) and
- **(semi-)automatic generation of web applications** (based on ATL and graph transformations)

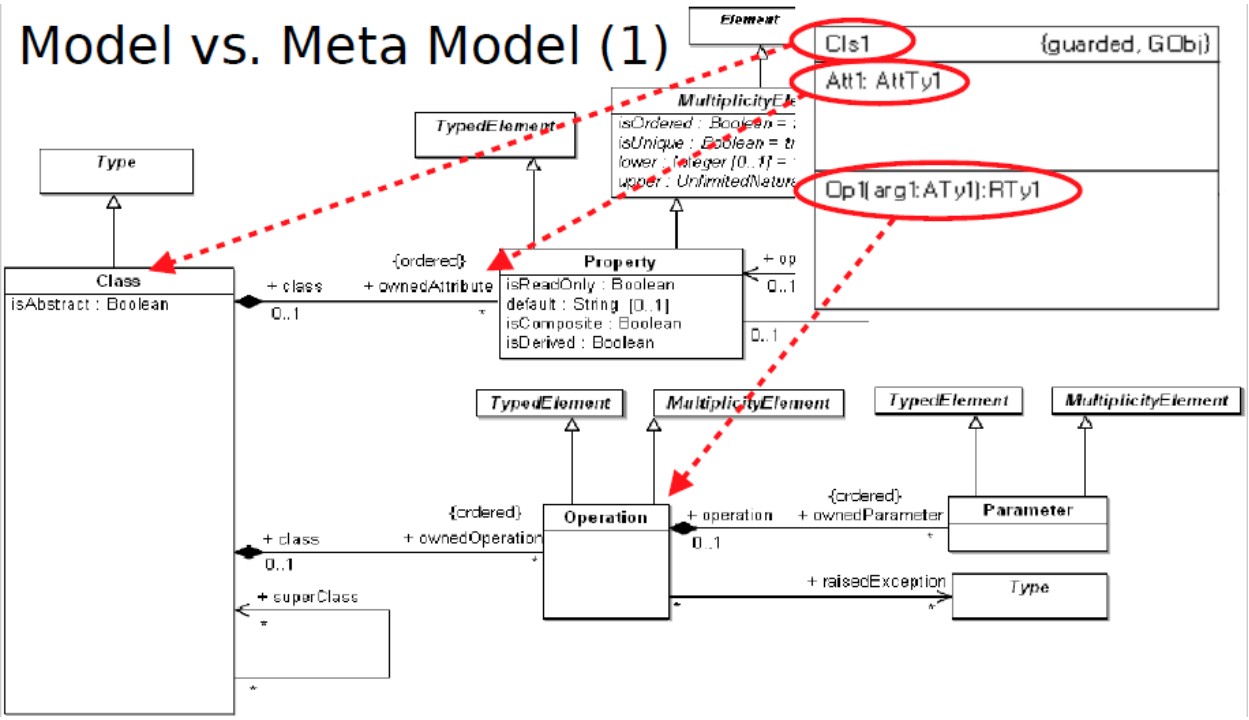


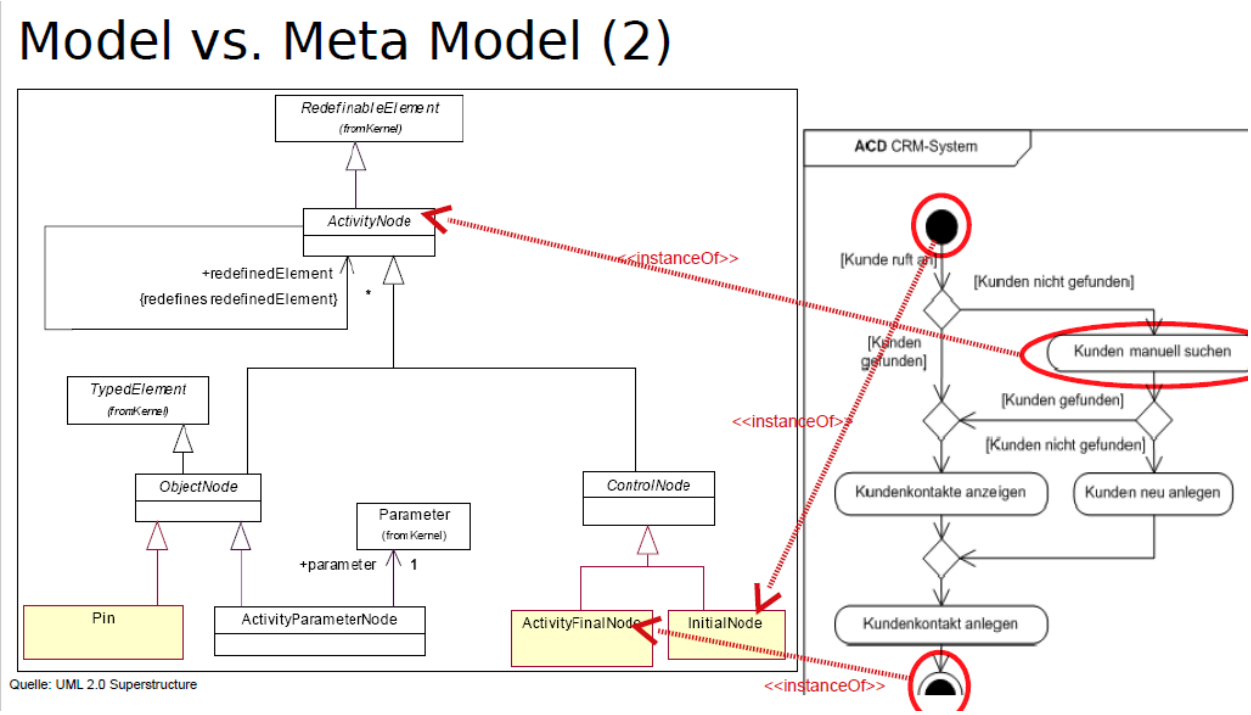
UML/MOF: 4-layered Language Architecture

- **Meta-Meta Model (M3)**
  - Defines language to define languages
  - MOF: Meta Object Facility
  - Examples: MOF:Class, MOF:Attribute, MOF:Association
- **Meta Model (M2)**
  - Meta models are instances of M3
  - Define (abstract) syntax of modeling languages
  - Examples: UML:Class, UML:State, UML: Attribute
- **Model (M1)**

- Concrete UML Model
- Models are instances of M2
- Examples: Class Person with attributes Name, Birthday, …

- **Runtime Instances (M0)**
- Concrete instances: person Gregor





How to Define a UML Extension

The following options exist for **defining modelling languages within MOF**:

- Definition of **new meta model** (not necessarily extension of UML, but could be)
- **Heavy-weight** extension: uncontrolled extension of UML meta model

- **Light-weight** extension: controlled extension of UML meta model with stereotypes, called UML profile

UML Profiles

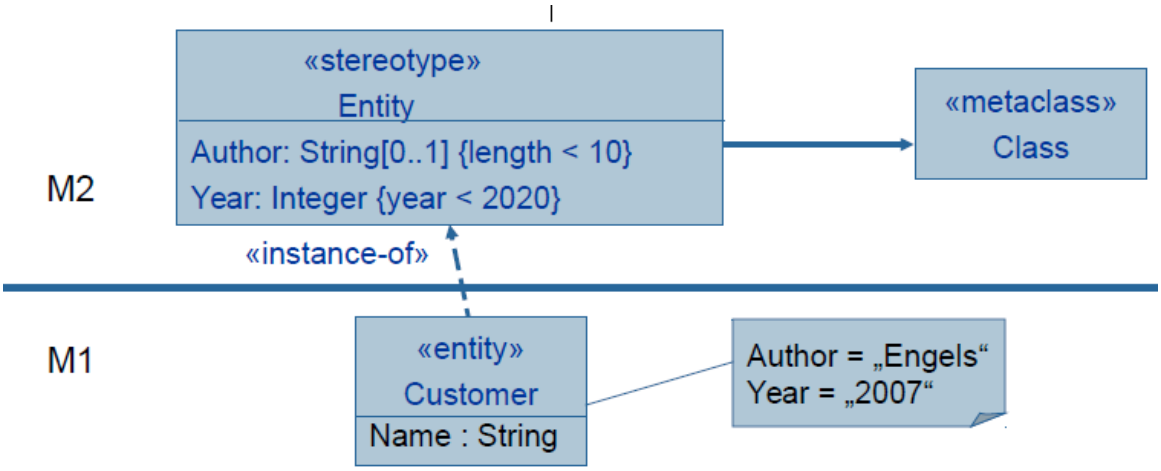
- UML is a **general-purpose modeling language**
- UML can be adapted to **become a domain-specific language** (DSL)
- Domain-specific adaption of UML is called:

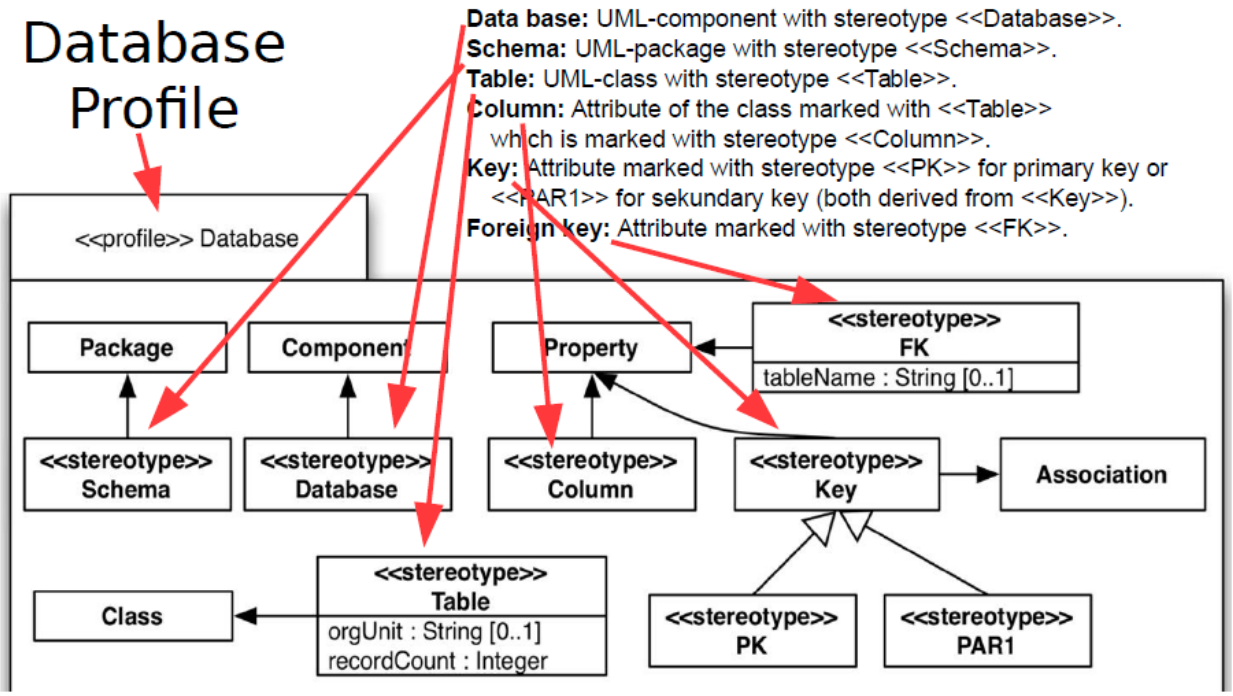
UML Profile

- Domain-specific extensions are indicated by stereotypes
- Notation: `«stereotype»`

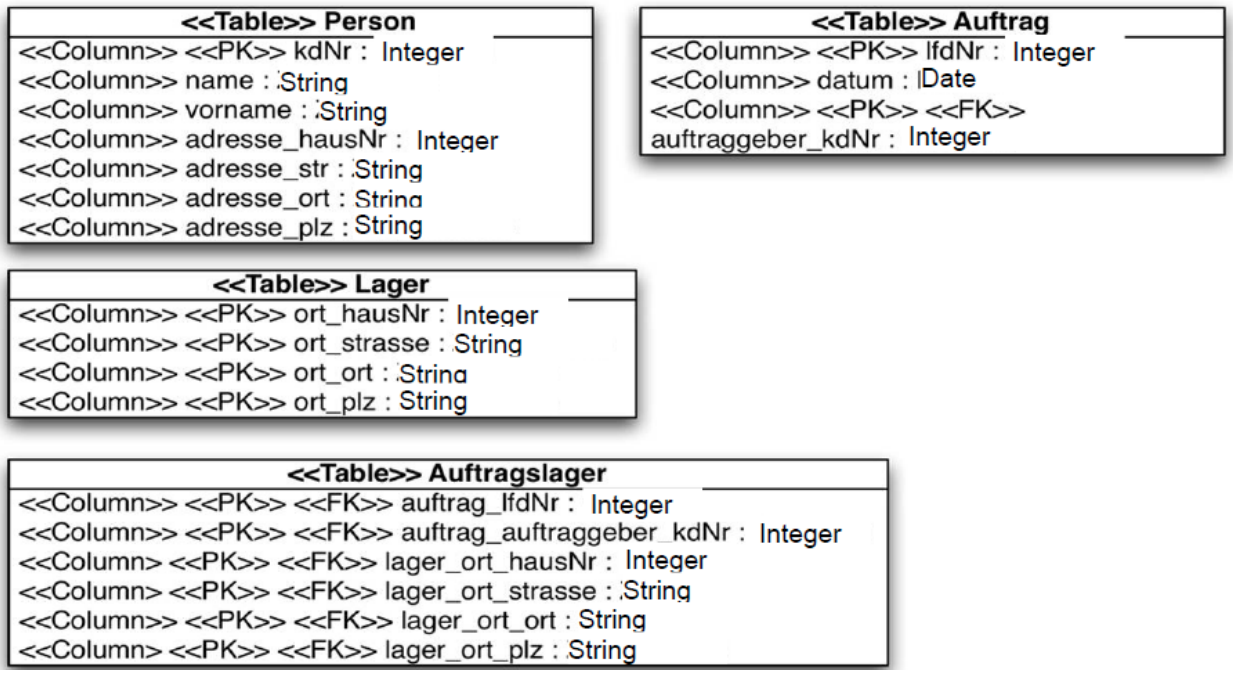
Extension of UML-metaclasses

- + Additional meta attributes (“tags”)
- + Additional constraints





Example: Using Database Profile



MODELING WEB APPLICATION REQUIREMENTS WITH UWE

Modeling Functional Requirements

- Overall functionality modeled by **UML use cases**
- Based on the view of actors
- Two types of requirements
- **Functional** (to be found in all software systems)
- Annotated with the stereotype **«processing»**
- **Login, register**
- **Navigational** (typical for web applications)
- Annotated with the stereotype **«browsing»**
- **Search, list**

Each use case should be detailed by several scenarios.

A scenario is a single intended run through a use case.

Descriptions of scenarios may use

- natural language
- sequence diagrams

Sometimes a few closely related scenarios are grouped using behavioral descriptions such as:

- activity diagrams
- decision tables?

Scenarios help to **understand the intended system** and support the development of the design model.

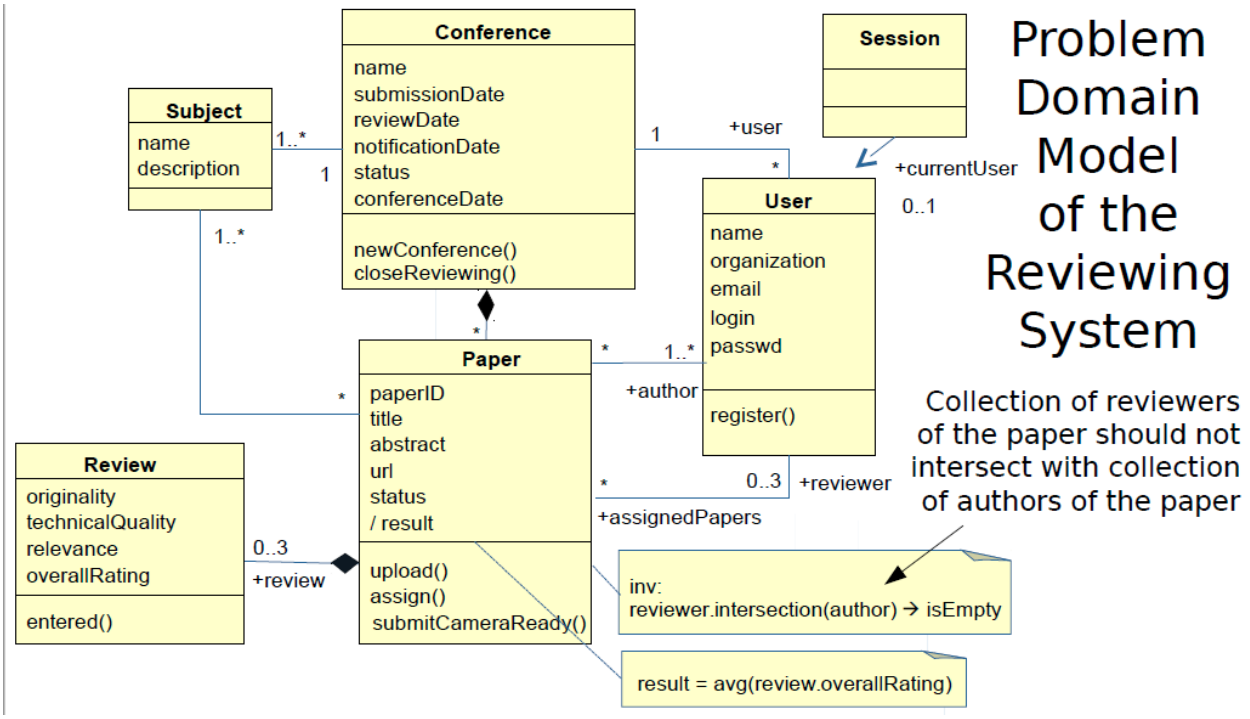
Mockups and prototypes help to develop and test the **<<browsing>> use cases**.

Use cases and scenarios structure and refine the requirements on the systems functionality.

Modeling the content

The information inside the pages is the **content of the hypertext.**

Modeling of content corresponds to conceptual modeling (domain modeling), which can be done using **UML class diagrams.** They build on a **(conceptual) domain model**.



Persistence

For **transactional pages**, the content model is the first conceptual model for the database.

If a **legacy database** is to be used, the content model and the database model have to be mapped, i.e., they should be as similar as possible.

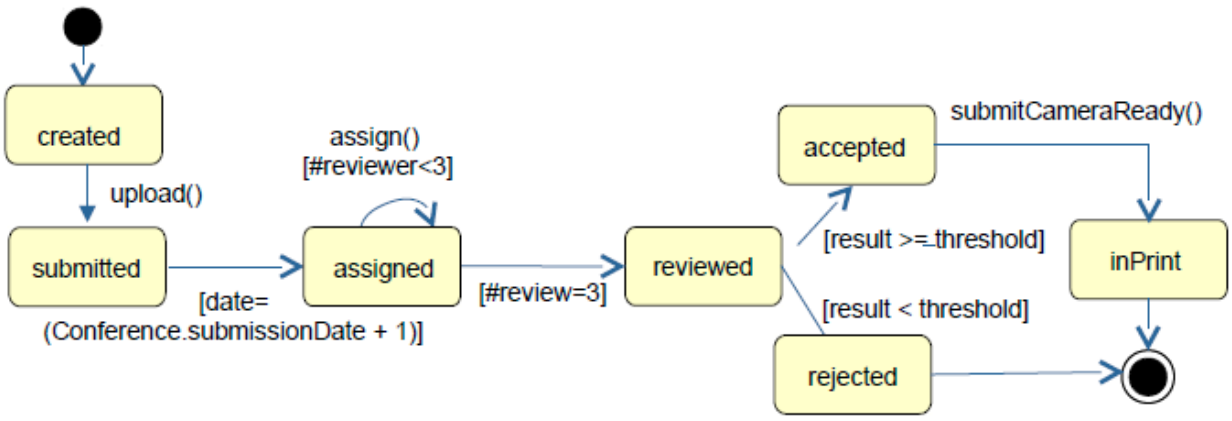
Behavior

More behavior of the content may be specified in the requirements phase.

E.g., possible interactions with the content on the client side may be defined using **state machines** for object life cycles, i.e., the description of its different states of a content unit.

Usually, **domain models do not yet have methods**. But, for dynamic web pages also the content-specific behavioral part (methods) of the application may be modeled, as far as it is needed on the client side.

Life Cycle of a Paper



Operations in the state machines life cycle are defined in the conceptual domain model.

Modeling the Hypertext

Navigation modeling means to specify the possible paths through the content.

Navigation models are **built on top of the content model.**

Models:

- **navigation structure model (NSM)** defines the structure of the hypertext, i.e., which classes of the content model can be visited by navigation.
- **access model (AM)** refines the NSM by access elements.

Navigation Modeling Concepts in UWE

Navigation structure

- «navigation class» : for navigation nodes
- «navigation link» : for navigation links

Access structure

- «menu» : access to nodes of different classes
- «index» : access to single nodes of one class
- «guidedTour» : sequential access to a list of nodes
- «query» : search for a node and direct access

NSM – Navigation Structure Models

Navigation structure models (NSMs) are **based on the concepts (classes) of the content** model.

The NSM is a **view for a given stakeholder** on the content model.

NSM Elements

<<navigation class>> represents a visible node in the hypertext

<<navigation link>> is an allowed navigation between these nodes.

NSM Links

UWE distinguishes

- **navigation links**, which connect nodes in the hypertext
- **process links**, which point to the start node of an interaction process
- **external links**, which point to nodes not directly belonging to the application

How to Find (Make?) a Navigation Model?

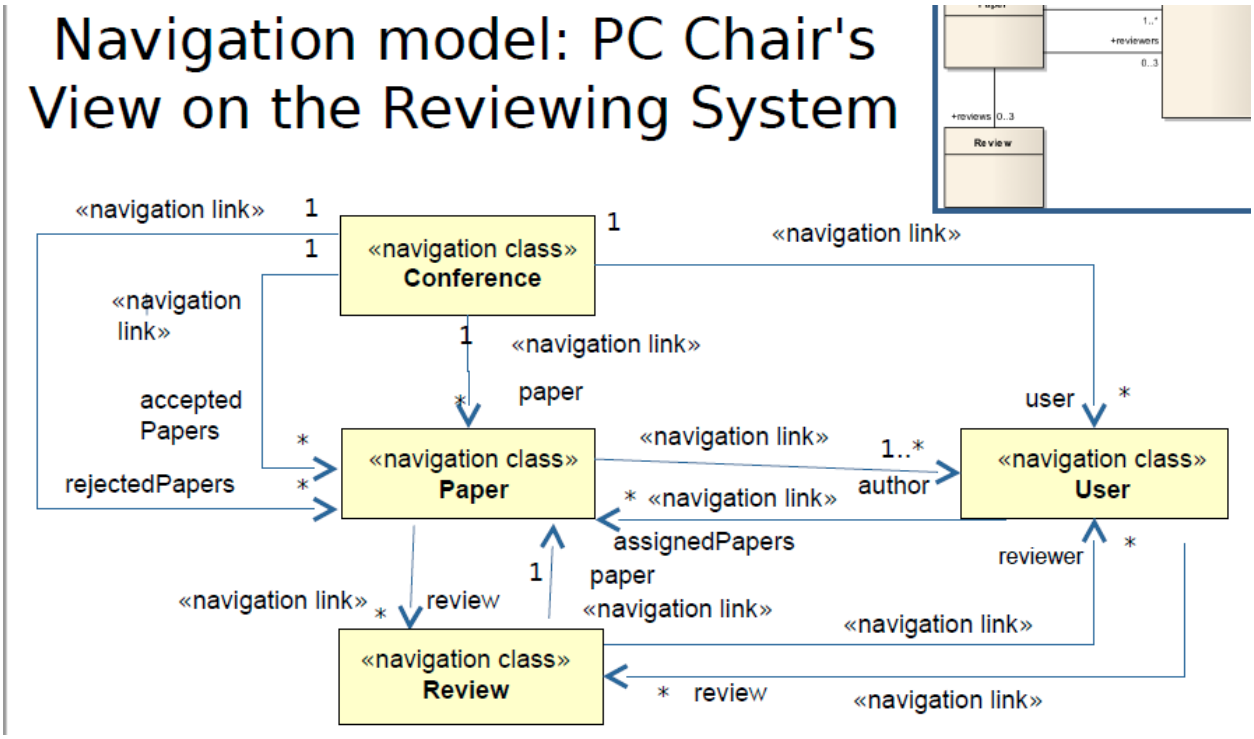
1. Define **navigation class** for each navigation-relevant **content class**

2. Define **navigation links** for relevant **associations**, **aggregations** and **compositions** of content model

3. Add **multiplicities** and **role names** existing in content model

4. Add additional **navigation links** due to the **scenarios** of requirements analysis

5. Add **additional navigation** links as **shortcuts**



Other Links in NSM

There may be more kinds of links, such as:

- **structural** links, leading to parts of a node
- **perspective** links, leading to other views of the node
- **contextual** links, leading to more detailed information
- **traversal** links, leading to the next sibling wrt. to a parent node

AM – Access Models

The access model (AM) refines the NSM by defining **the kind of presentation** of and access to the content in more detail.

The navigation classes are extended by **concrete interaction elements**, called **navigation patterns**.

Navigation Patterns define interaction widgets:

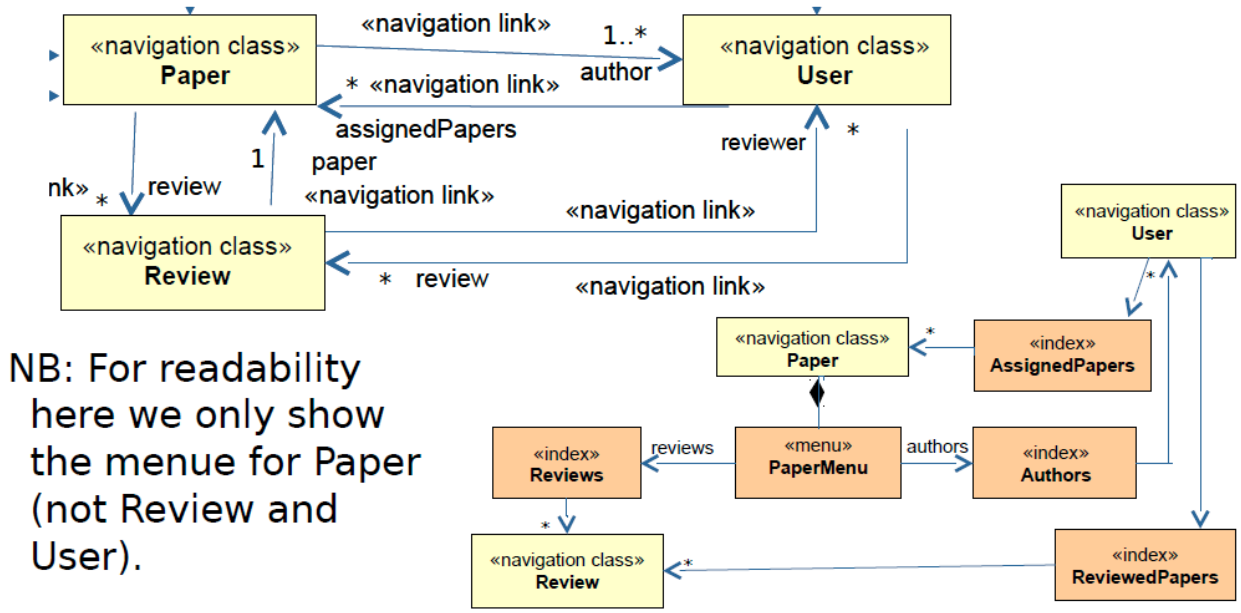
- an **index** is an access structure, which allows to select **a single object out of a list**
- a **menu** allows the user to choose the **node to access next**
- a **guided tour** allows the user to **sequentially walk** through a number of nodes
- a **query** allows the user to **search for nodes** and special links:
- **home** points to the **homepage**
- **landmark** points to a page **reachable from every node**.

Adding Access Structures

(Automatic) method to add access structures to the navigation model

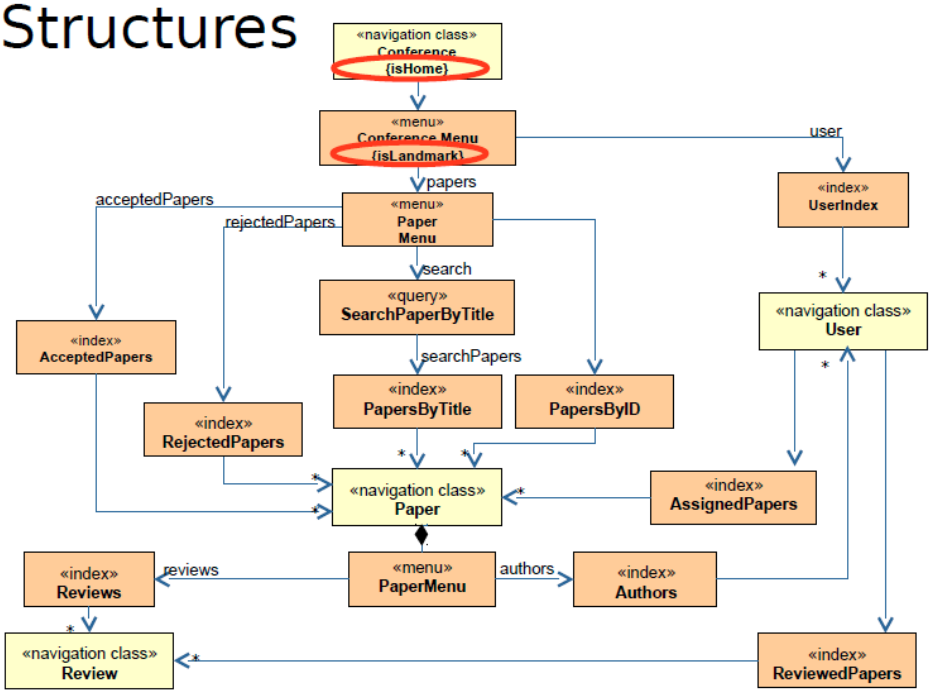
- Introduce index for all navigation links with multiplicity >1
- Introduce menu for each class with more than one outgoing navigation link
- Use role names of outgoing navigation links as menu items

Adding Access Structures to Navigation Model



NB: For readability here we only show the menue for Paper (not Review and User).

Extended Navigation Model with Further Access Structures

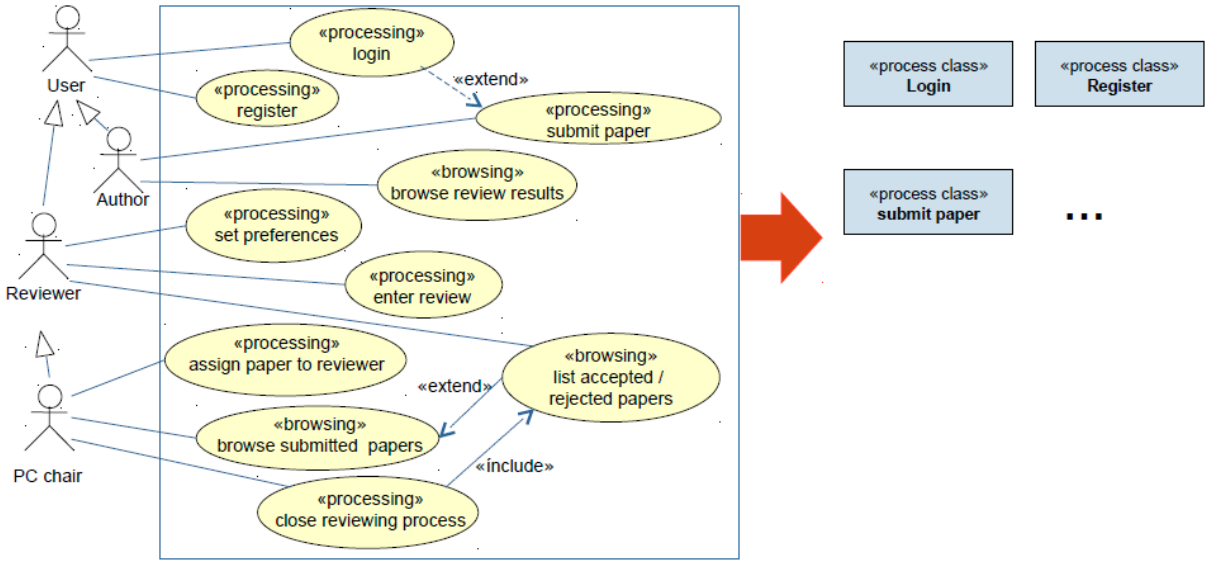


Process Modeling

Process modeling

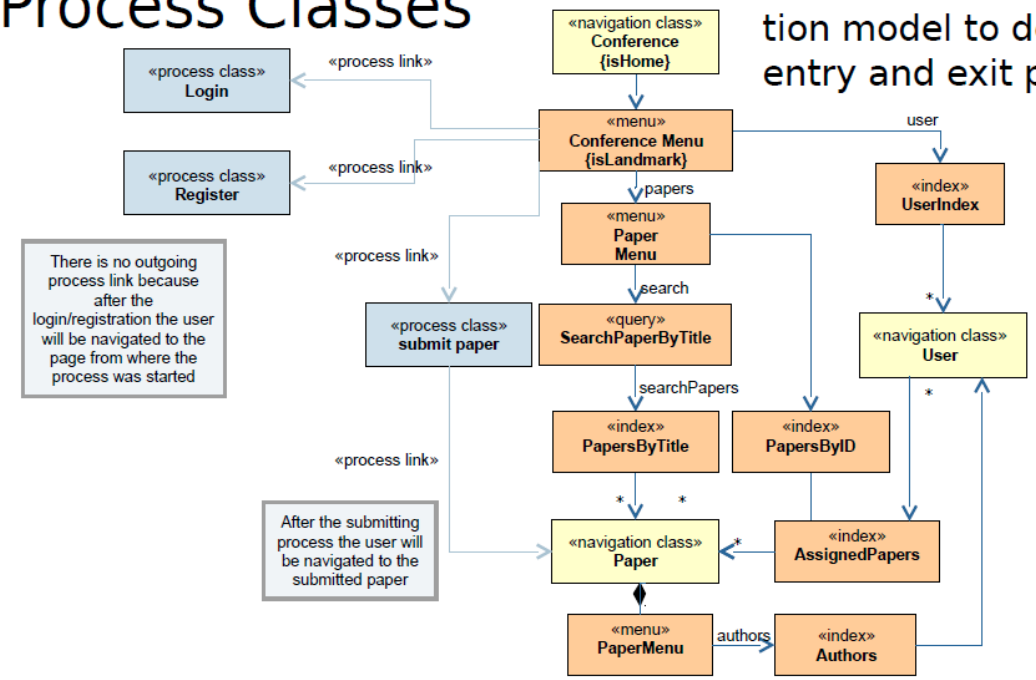
- Up to now only the access to the content is covered
- Process modeling in UWE defines the access to the workflows of the web application as well as their behavior
- Steps:
  1) Definition of process classes for processing use cases
  2) Integration of process classes into the navigation model to define entry and exit points
  3) Specifying the behavior of the process class using UML Activity Diagrams

1) Derive Process Classes



2) Integration of Process Classes

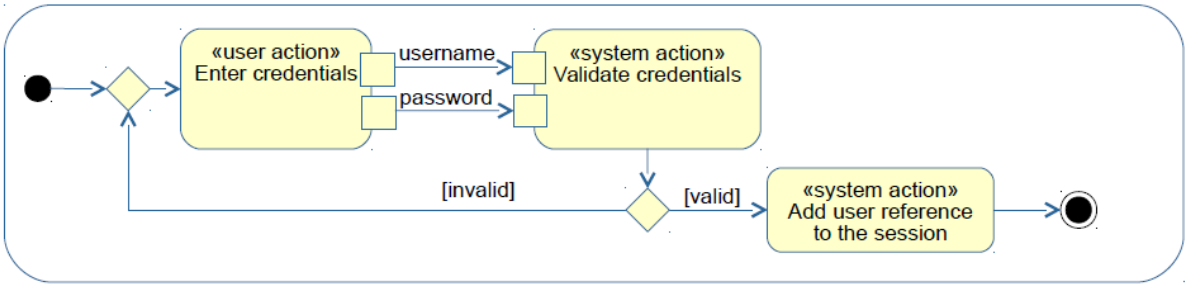
2) Integration of process classes into the navigation model to define entry and exit points



3) Specifying the Behavior

3) Specifying the behavior of the process class using UML Activity Diagrams

- Distinction between two kinds of actions
  - «userAction»: Interaction of the user with the webpage
  - «systemAction»: Action executed by the system
- Example for the process class *Login:*



Modeling the Presentation

Presentation Modeling

Presentation modeling deals with the **user interface** and the **look-and-feel** of the Web application. The page is the main visualization unit in the Web.

Goals:

- Interaction shall be simple and self-explanatory
- The user should be oriented where he/she is

Objectives

- Model of structure and behavior of user interface

Characteristics of presentation modeling

- Hierarchical composition of pages consisting of presentation elements

Results

- Static presentation model
- Dynamic interaction model

Parts of presentation models:

– a **uniform presentation of recurring elements**
- e.g., **headers**, **footers**, the composition of the page, the design of fields, texts, and forms, ...
– the **behavior-oriented aspects** of the user interface
- e.g., which button to click to activate a function

The graphical layout style of the user interface is often produced by a graphic designer based on some basic drawings. (This activity is not part of modeling.)

Presentation Structure

Constituents of presentations are, e.g.,

– Presentation page, denotes the page as a visualization unit
– Presentation unit, serves to group related user interface elements and may be nested

Widgets used in presentations are, e.g.,

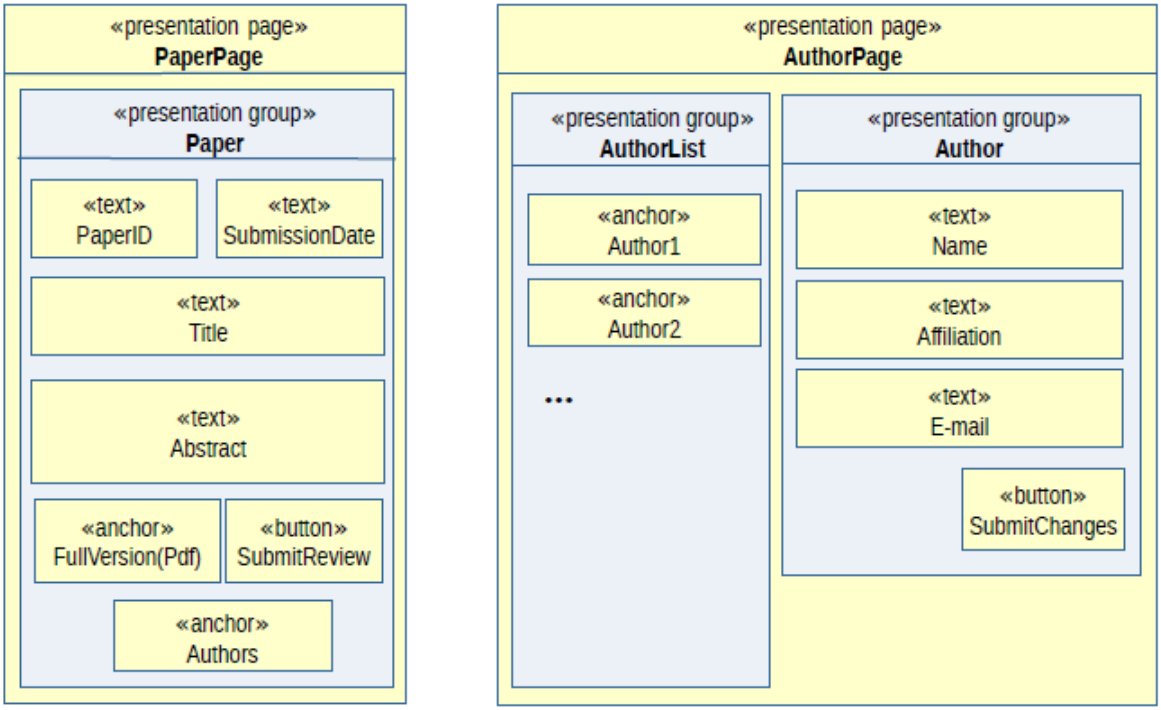
– **Anchor**: allows navigation to another unit
– **Button**: allows an explicit start of a function
– **Text, image, audio**: represent kinds of information

Presentation Modeling Concepts in UWE

- «**presentation page**» describes a page presented to the user as a **visualization unit**. It can be composed of different presentation groups.
- «**presentation group**» serves to group **related presentation elements**, representing a logical fragment of the page, and it presents a hypertext model node

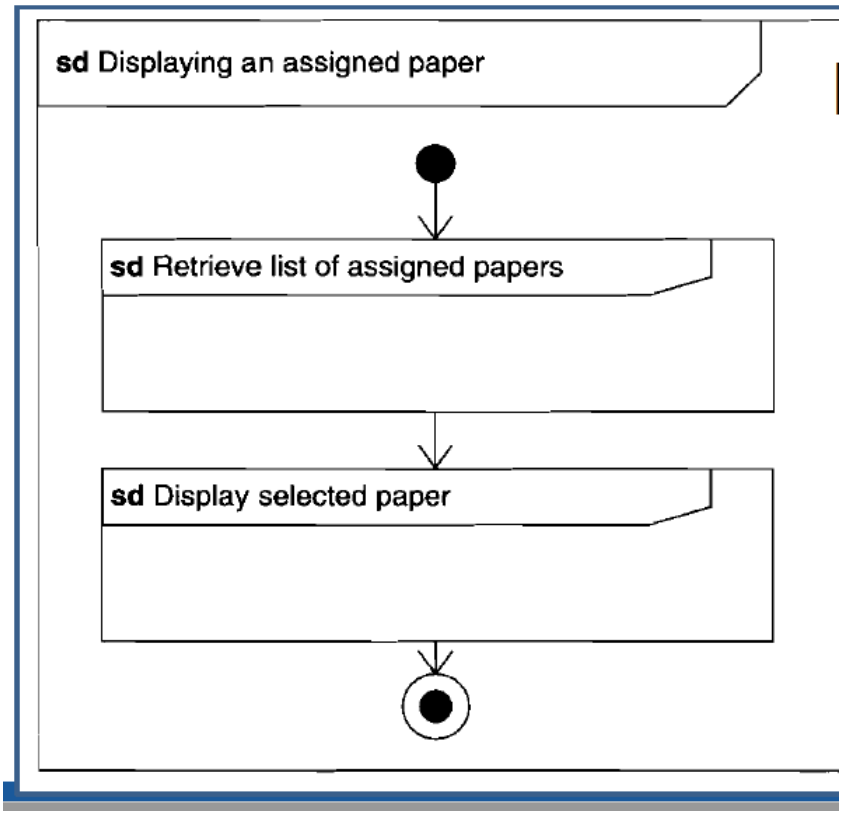
Presentation elements include:

- «**anchor**»
- «**text**»
- «**image**»

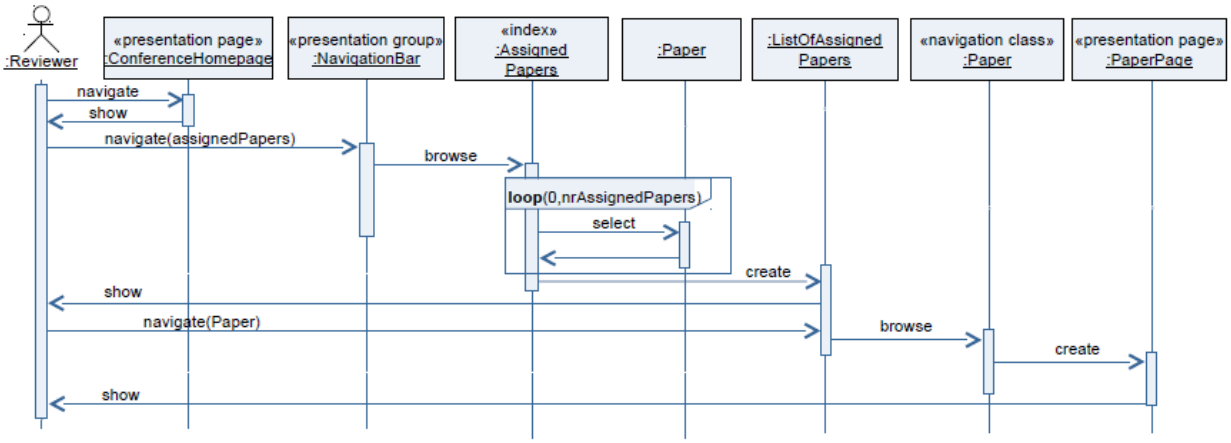


Presentation Behavior

Behavioral Aspects can be modeled by **behavior diagrams** (e.g. activity diagram).



Scenario from the Presentation Model for the Reviewing System



Model Customization

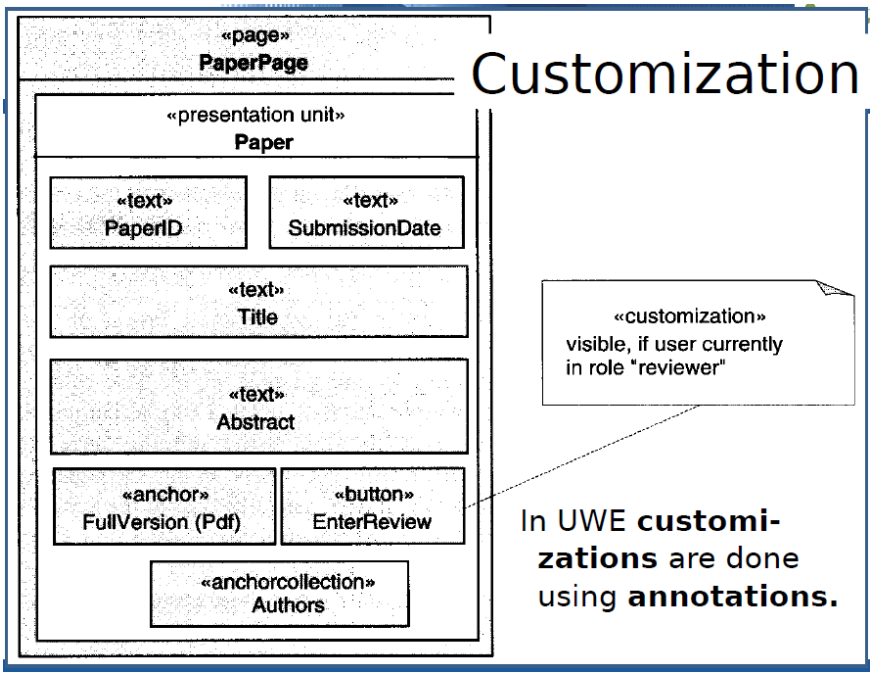
Customization pertains to

- **personalization** depending on a user profile
- **adaption** to technical requirements, e.g., for mobile computing: device profile, location information, transmission characteristics

Adaptation can be done, e.g.,

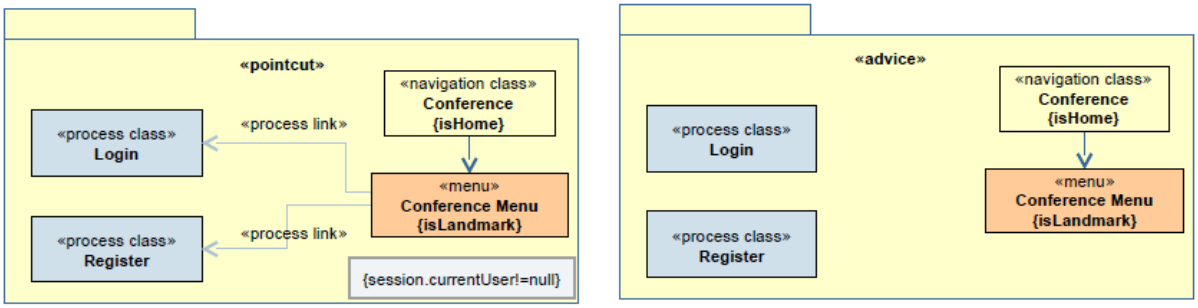
- **statically**, by including all different cases in the model, e.g., modeling the possible roles of an actor (may leads to combinatorial explosion)
- **dynamically**, created at generation time or runtime using
    - **aspects**
    - event-condition-action rules (**ECA rules**)

In **UWE customizations** are done using **annotations**.



Adaption

- UWE allows adaptation of the navigation, content, and presentation by specifying **pointcuts** and **advices**
- **A pointcut** defines a situation in which the **advice** should **be applied**
- Example: **Hide Login/Register if user is logged in**



Modeling Support in UWE

- **Requirements** Specification
– **Use case diagram**
- **Content** Modeling
– **Class diagram**
– **State chart**
- **Navigation** modeling
– **Navigation** model
– **Access** structures
- **Process** modeling
– **Activity diagram**
- **Presentation** modeling
– **Static presentation** model
– **Dynamic interaction** model
- **Adaptation**

Specific **extensions** concerning additional aspects are **supplied by the UWE** modeling profile.

This profile comprises **new elements** for requirements, content, hypertext, presentation, and customization.

Web architecture

Architecture

- Makes a **system understandable**
- Describes structure from different viewpoints (**perspectives**)
- Supports **design and implementation**
- Constitutes the frame for a **flexible system**
- Lays the grounds for achieving **non-functional properties**

Bad Architectures lead to

- Poor performance
- Insufficient maintainability and expandability
- Low availability

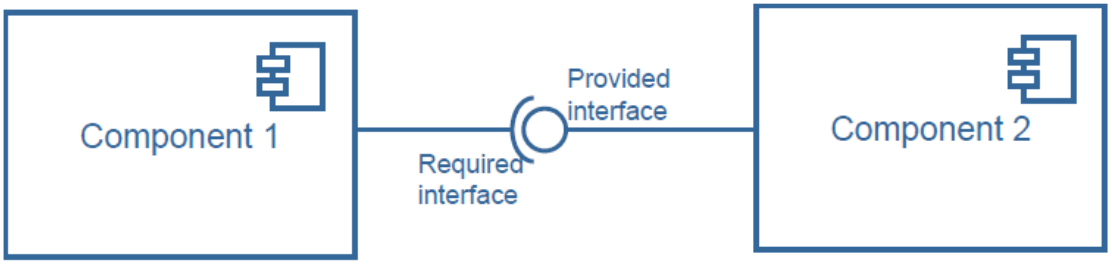
What is Architecture?

- It is a vehicle for communication among stakeholders
- It is the manifesto of the earliest design decisions
- It is a reusable transferable abstraction of a system

Building Blocks of a Software Architecture

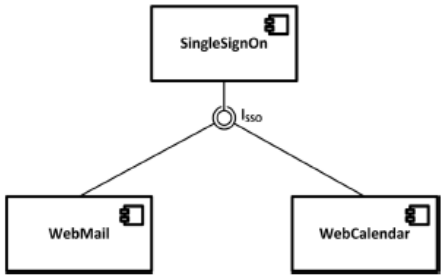
- **Component**
  - **Body**: hides implementation ("information hiding")
  - **Export interface**: provided information (types, data, operations)
  - **Import interface**: required information
- **Connector**
  - Between import and export interfaces
- **Protocol**
  - Description of allowed / forbidden interactions via connectors between components

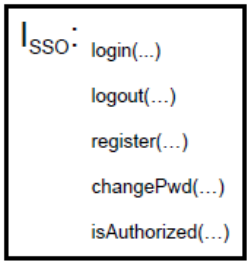
UML 2.0 Component Diagram

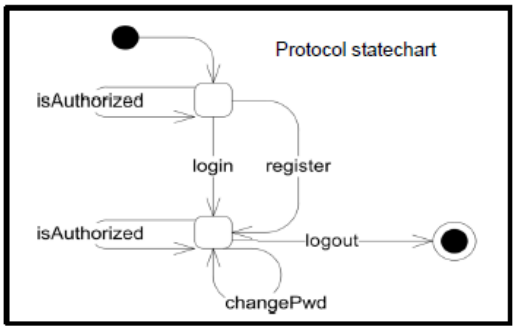


UML 2.0 statecharts for protocol specifications

UML 2.0 Component Diagram







The protocol statechart describes the interaction between a user of the system and the SingleSignOn component

Objectives of Component Structure

- **High cohesion**: for elements within a component
- **Low coupling**: between components, i.e., a component should depend on only a few other components
- **Reusability**: components should be reusable in different settings
- **Changeability**: modifications within a component should have no effect on other components

Communication Types

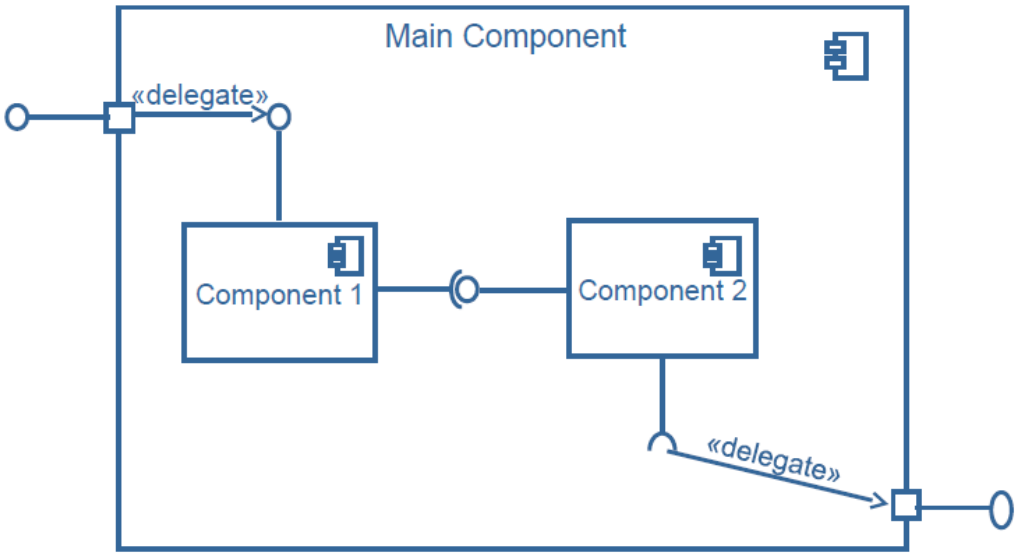
- Peer-to-peer communication
– free connections
- Client-server communication
– thin or thick clients
- Synchronous vs. asynchronous communication

Hierarchical Composition

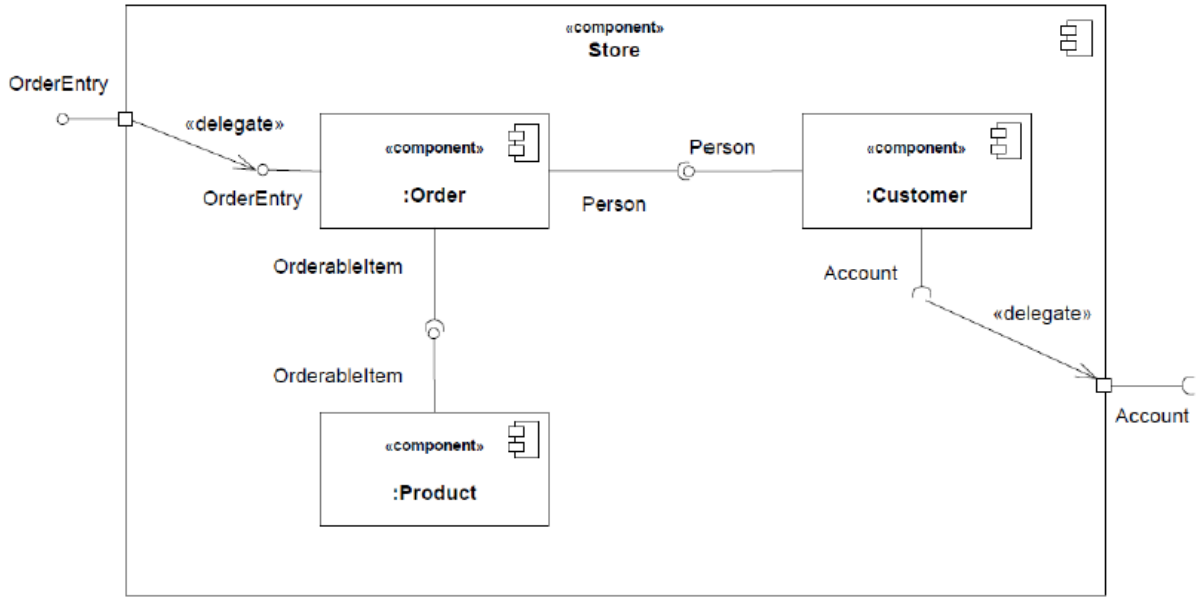
Encapsulation:

- Container as abstraction means
- Container consists of other components
- Inner components might be exported, i.e. are visible
- Behavior of container composed of behavior of constituents
- Containers are components, too

Hierarchical Composition of Components



(UML 2.0 Superstructure Spec., p. 147)



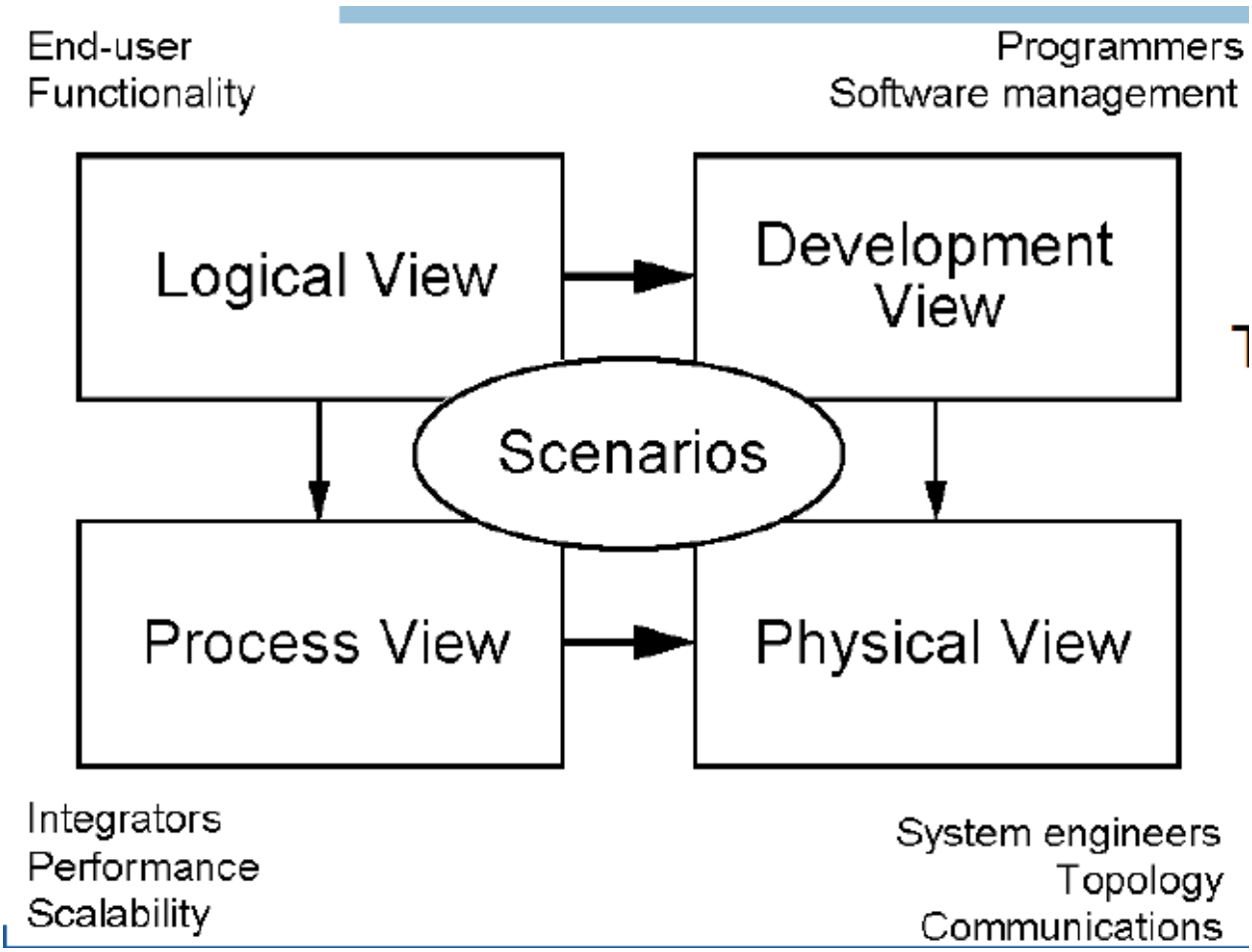
Modelling Architectures: Viewpoints and Views

A software system has **one overall architecture**. This architecture is usually described by a set of architecture views from different viewpoints.

Views are **instances** of **viewpoints**.

Different stakeholders may have **different viewpoints**.

Kruchten-Viewpoints



Logical view – use case diagram

Development view – class or package diagram

Physical (Deployment) View - deployment diagram (that client and server image jan likes so much)

Process view - component diagram (that web server image jan likes so much)

Scenarios

Kruchten proposes an iterative development procedure using the scenarios collected during requirements engineering: The designer team runs through the scenarios (one by one) and iteratively completes the different architecture views.

Specifics of Web Architectures

When discussing architecture in the context of Web Engineering, we have to differentiate:

– **The web infrastructure architecture** (depending on the technology)

– **The web application architecture** (depending on the domain)

Architectural Knowledge

The infrastructure architecture may be given in advance. For the development of applications, a good and appropriate application architecture has to be chosen.

Since architectures are a very abstract concept, knowledge about architectures (best practice) is encoded only semi formally by high-level constructs, e.g.,

- **architectural styles** (architecture patterns)
- **frameworks**
- **component libraries**

ARCHITECTURAL STYLES

Reoccurring architectural designs are listed in **catalogs of architectural styles**.

Architectural Styles are **patterns of architectures**, which bind their essential properties to a name.

Architectural styles support **reuse of proven and consolidated knowledge**.

There are some basic architectural styles that particularly fit for web applications, e.g.,

- **layered** architectures
- **data flow** architectures
- **data centered** architectures
- **model-view-controller** architectures.

Layered architectures

Layers

- Are arranged on top of another
- Elements of one layer are of a similar abstraction level
- A layer provides services to the upper level (i.e., it acts as a server)
- A layer uses only services from the immediate lower level (i.e., it acts as a client). It is not allowed to directly use layers below the immediate lower one

– Advantage:

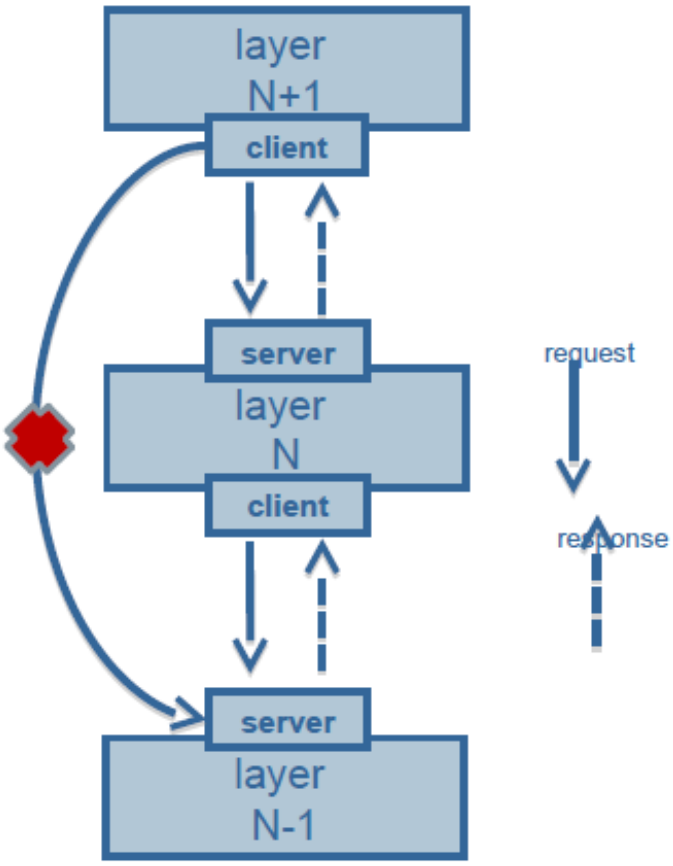
- Changes in a layer effect only one other layer
- Reusability of layers

– Example:

- ISO-OSI stack

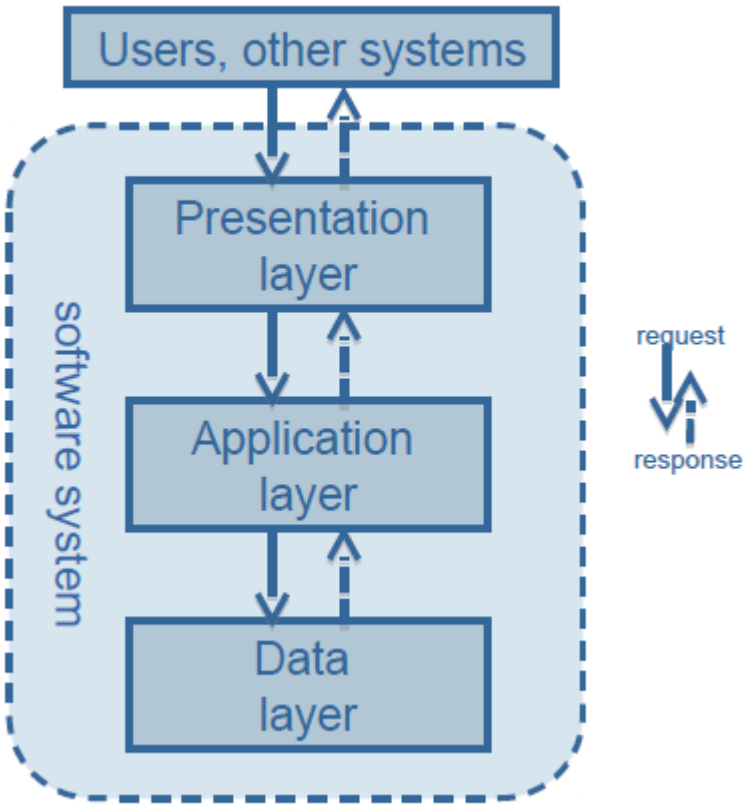
Often applications are structured on several layers with **different responsibilities**.

- The layers consist of **cooperating components**.
- **Lower** layers **supply services** to the next **higher** layer.



Most software systems are designed **around three layers:**

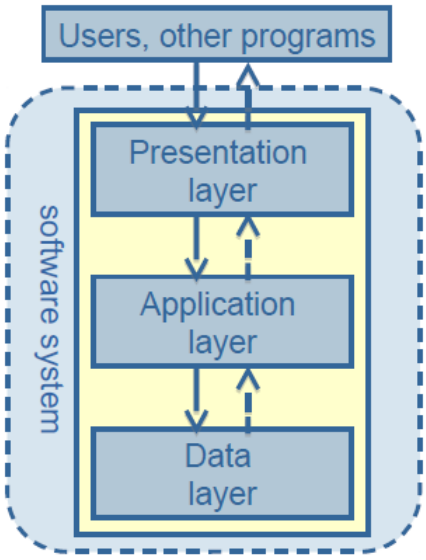
- **Presentation layer**: Encapsulates interactions with users or other systems
- **Application layer**: Determines what the system actually does. It takes care of enforcing the business rules and establish the business processes.
- **Data layer**: Deals with the organization (storage, indexing, and retrieval) of the data necessary to support the application layer.



Layer vs. tier:

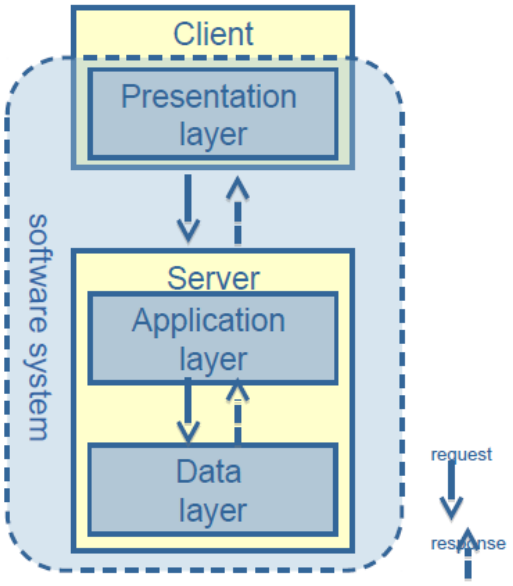
- Layers are conceptual constructs
- Tiers are physical constructs (“processors”)
- Layers are assigned to tiers (“deployment”)
- Mostly a tier for each layer
- A tier might consist of several layers

1-Tier Architecture: Fully Centralized



- The presentation, application, and data layers are built as a monolithic entity
- Users / programs access the system through display terminals
- What is displayed and how it appears is controlled by the server (terminals are “dumb”).
- Typical architecture of mainframes, offering several advantages:
  - No forced context switches in the control flow (everything happens within the system),
  - All is centralized, managing and controlling resources is easier,
  - The design can be highly optimized by blurring the separation between layers.

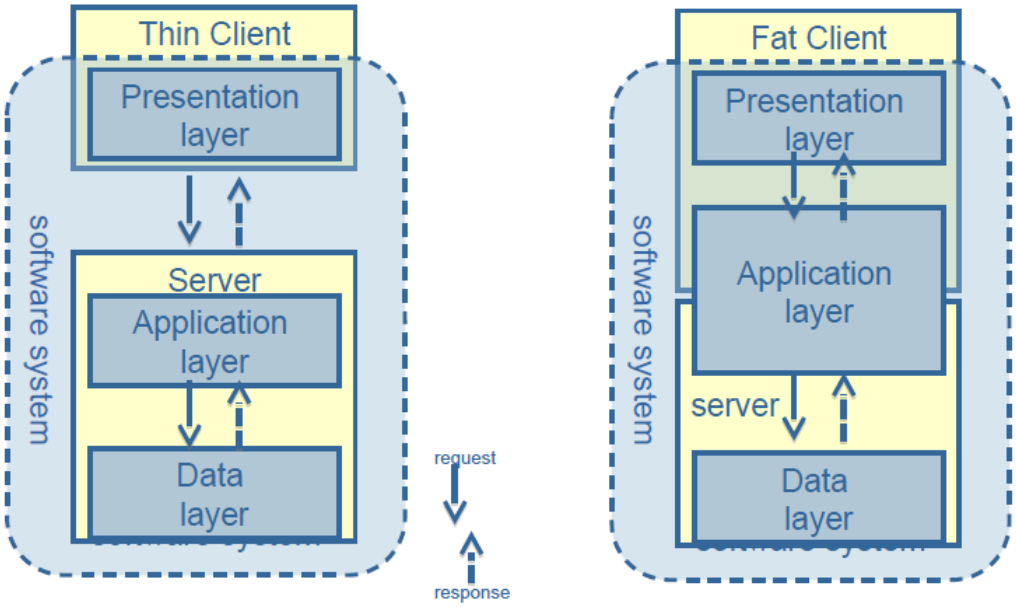
2-Tier Architecture (Client / Server)



As computers became more powerful, it was possible to move the presentation layer to the client. This has several advantages:

- Clients are independent of each other: one could have several presentation layers depending on what each client wants to do.
- One can take advantage of the computing power at the client machine to have more sophisticated presentation layers. This also saves computer resources at the server machine.
- It introduces the concept of *API (Application Program Interface)*. An interface to invoke the system from the outside.
- The data layer only sees one client: the application layer. This greatly helps with performance since there are no client connections / sessions to maintain

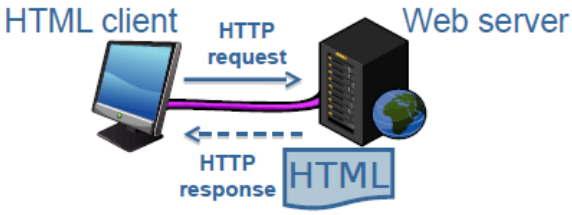
Fat Client (Rich, Smart) vs. Thin Client



WA Architecture

A priori all Web Applications have an overall **client / server architecture**.

Web Application Architecture (1): 2-Tier (Client-Server) Architecture

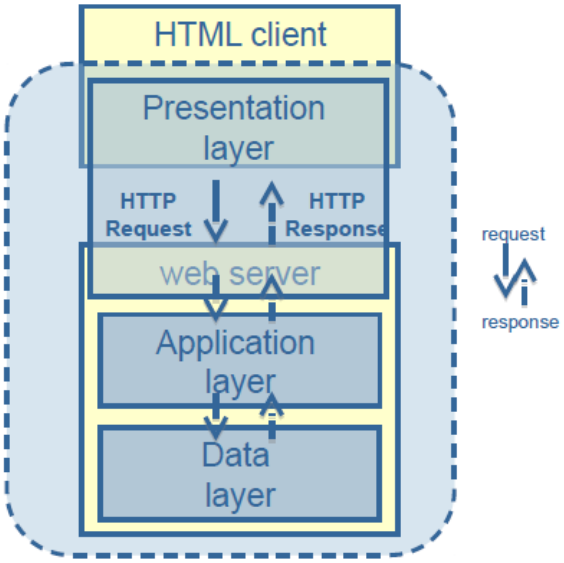


Category 1: document-centered

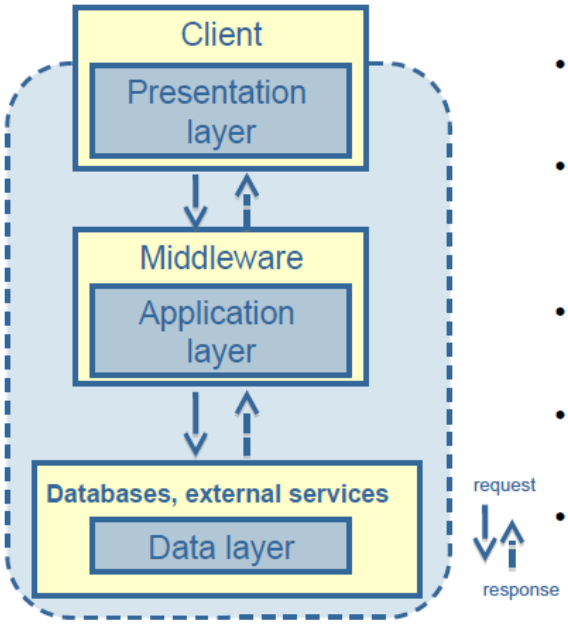
- „static“ HTML pages stored at the server

Category 2: interactive

- „dynamically“ created HTML pages within the web server

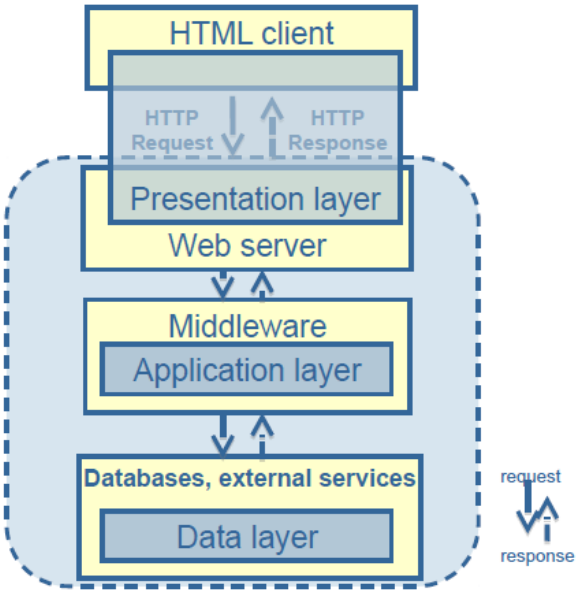


3-Tier Architecture (Middleware)



- In a 3-tier system, the three layers are fully separated.
- The layers are also typically distributed taking advantage of the complete modularity of the design
- A middleware-based system has a 3-tier architecture
- The middleware offers additional integration logic features
- The database tier may itself be once more a 3-tier system

Web Application Architecture (2): Web-based n-Tier (Client-Server) Architecture



- A 3-tier system extended by adding a web server to the presentation layer

Benefits

- Separate development of GUI, logic, and data aspect
- Business logic in the application layer may evolve independently from technology

Advantage of Layering

Layering **reduces complexity**:

- it has only **few constituents**
- it is **refineable** (allows zooming)
- it supports **separation of concerns**

Data Flow Architectures

The **dataflow-viewpoint** consists of activities (processes) and the **data they consume and/or produce.**

- -UML activity diagram
- -BPMN diagram

Batch-Sequential

Dataflow descriptions suggest a **batch-sequential** discipline, i.e.,

– activities are independent transformational processes
– every process is has to terminate, before the next process may start.

‚DATA-CENTERED' ARCHITECTURES

‚Shared Repository'

Several active and independent components use the services of a common repository (e.g., a database) according to their needs.

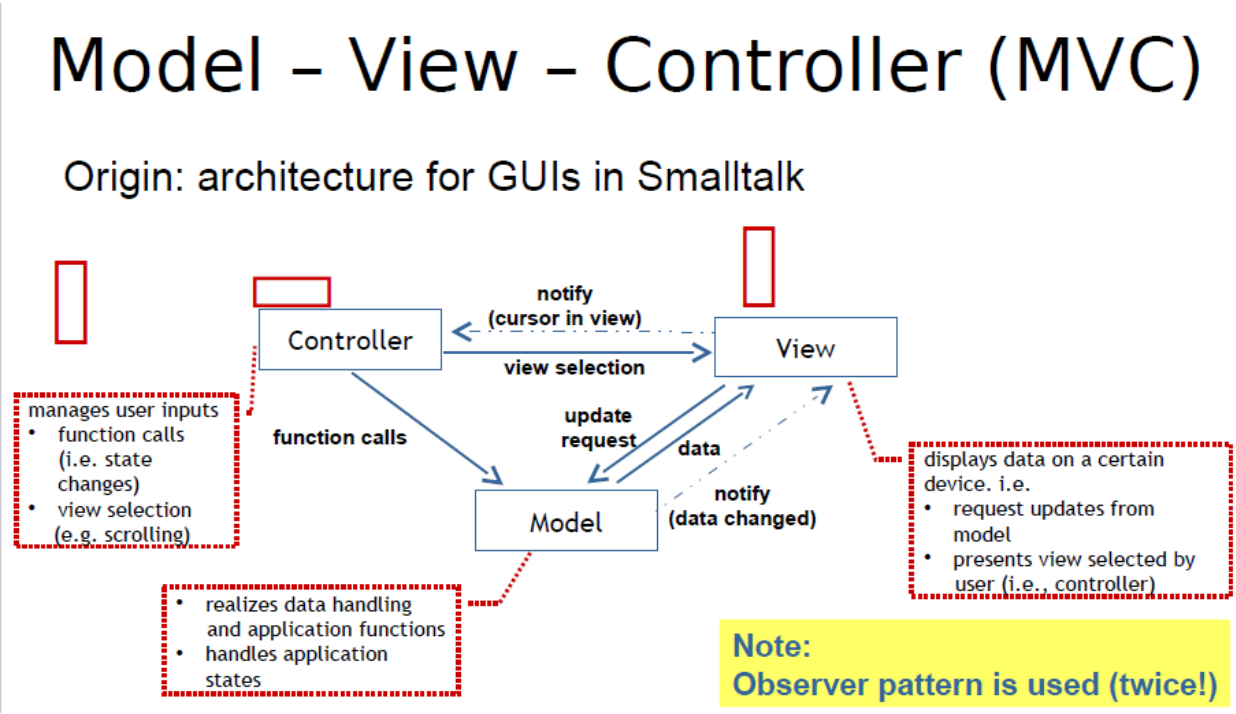
‚Active Repository'

The repository may also control the application by actively informing them about changes.

The components react to the changes independently.

Activation of the components may be done by (remote) invocation, by messages, or publish-subscribe mechanisms.

MODEL-VIEW-CONTROLLER ARCHITECTURES



MVC for Web Applications

Interactive applications (like dynamic web pages) are often built on **three different kinds of classes:**

– The **model** objects contain the **core functionality and the data**.

– The **view** objects **present the model** to the user.

– The **controller** objects **manage the overall workflow** (e.g., reactions to the user's input).

MVC: Variants

Note, that there are several variants of MVC (wrt. the tasks and interactions between the different parts), e.g.,

– **push-based**: data are pushed to the view from the model to render the results
– **pull-based**: the view pulls the results from the model when needed

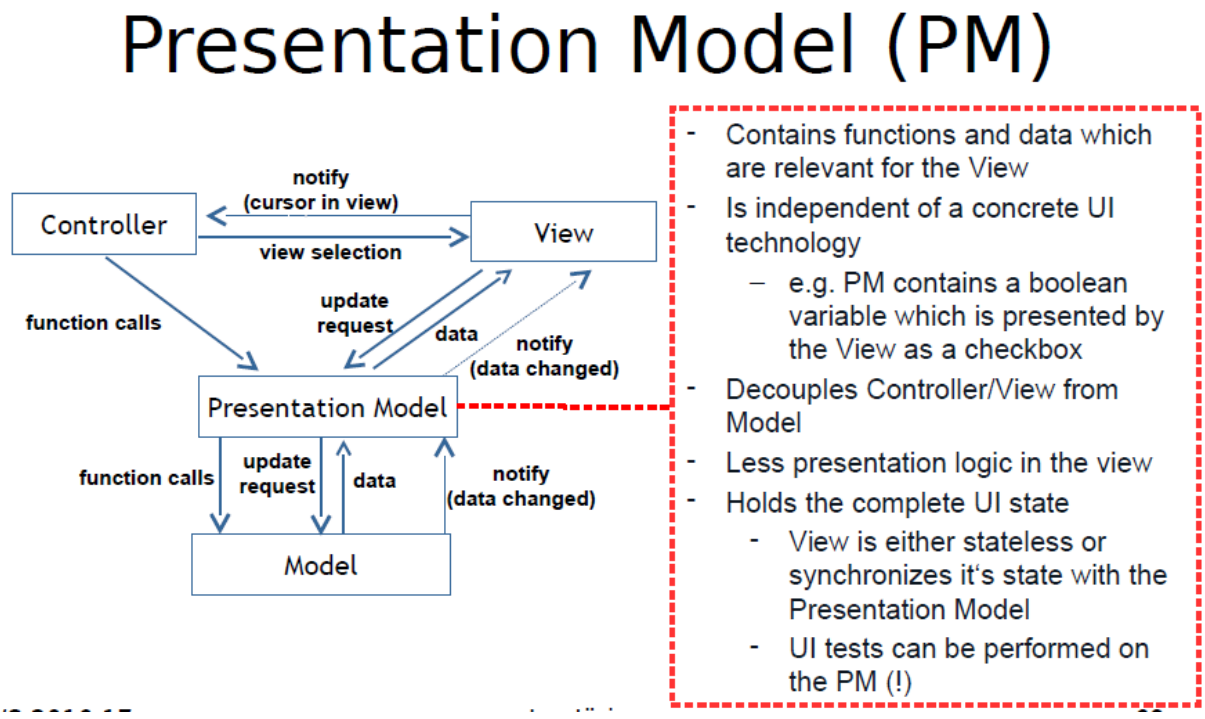
MVC: Advantages

- Different representations (views) for the same data
  - ⇨ good separation of concerns
- Support of several GUIs for the same data
  - ⇨ different input controller
- Easily extendable (for new user stories)
- Easily modifiable wrt. GUI and look-and-feel
- Supports different forms of interaction (keybord, icons, buttons, …)
  - ⇨ MVC pattern is (re-)used in most software systems!

Drawbacks of MVC

View requires presentation logic to process the model's data

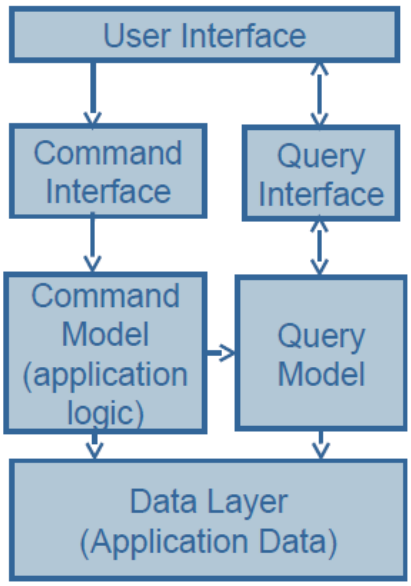
Improvement: **Introduce a Presentation Model**



Model – View – ViewModel (MVVM)

- Similar to MVC with a Presentation Model but **with a declarative binding between View and ViewModel**
– ViewModel is a synonym for Presentation Model
- Initially MVVM was developed for WPF and Silverlight
- Nowadays used in combination with other UI technologies (e.g. HTML and JavaScript)

Command Query Responsibility Segregation (CQRS)



- Applicable if #readOperations is far greater than #writeOperations
- Distinction between:
  - *Commands* which alter data (application logic)
  - *Queries* which just retrieve information without altering data
- Separate interface and model for Commands and Queries
  - Query model is just for data retrieval
  - Command model allows data manipulation and updates the query model
- Query performance can be increased by:
  - Optimizing the query model (e.g. storing preaggregated data)
  - Deploying the query model multiple times

Advantages:

- **High scalability** because queries and commands can be handled, scaled and optimized independently from each other
- Command model and Query model can be realized with **different technologies** (e.g. different programming languages or database systems)

Disadvantages:

- **Different models** might become **inconsistent**

WEB APPLICATION FRAMEWORKS

A **framework** is a blueprint of a software system that supplies basic functionality in the form of an (almost) complete system, which can be adapted to a new solution by specialization of provided classes and by addition of application-specific code.

A framework already contains the basic **architecture**.

Advantages and Drawbacks of Frameworks

Advantages:

– A framework serves as a blueprint: functionality and architectural knowledge can be reused.
– A framework saves routine implementation work: much work is supplied by others.

Drawbacks:

– high training effort,
– lack of standards,
– dependence on manufacturers.

Architectural Style vs. Framework vs. Component Library

While an **architectural style** is (only) an **abstract pattern** to transfer best practice knowledge, a **framework** is a **concrete implementation** that is refined and adapted to build a concrete application.

The boundary between the concepts “**framework**” and “**component library**” is fuzzy. Some frameworks are just **collections of reusable components**.

The **Zend Framework (ZF)** is an object-oriented web application framework for developing PHP-applications

Zend Components:

- Model-View-Controller (14): support MVC, request, response, forms, layout, ...
- Tooling and Rapid Application development (4): code generator, reflection, support for application development by command-line tools to generate project structure, etc.
- Database (5): data adapters for all major databases
- Internationalization (18) and Localization
- Authentication, Authorization, Session Management (3): user management
- Web and Web Services (14): exposing / consuming services
- Mail, Formats, Search (5): Ajax (JSON), PDF-generation, e-mail, search
- Core Infrastructure (12): logging, debugging, caching, filtering, ...

MVC in Zend

- **Controller**
- The developer writes several controllers, including Action-Controllers and Error-Controllers
- **View**
- The developer creates a Zend Layout-file containing HTML-text and PHP scripts
- **Model**
- The model is not prepared (there is no model-class in Zend), but has to be programmed “as usual” (using a database or web services).

WEB APPLICATION SERVERS

An **application server** represents an **environment** for the **development** and **operation** (!) of component-based, distributed **applications**.

It offers basic services, e.g., transactions, resource pooling, load balancing, naming, or directory services, which can be used by all applications.

Example: **JavaEE** is a Java variant and runtime environment to develop open, platform-independent, distributed client/server applications in Java.

In JavaEE, enterprise Java beans (EJBs) are used to

– implement application logic (session bean) or

– represent data (entity bean).

Web SOA

Web Applications quite often functionally (i.e., not by hyperlinks) use other web applications on other nodes (to find rselevant information, eg. Weather)

A **Web Service** is a server side interface that **offers server resources** to a client.

Web Services are a controlled way of communication between applications on different Web servers.

Web Services support component-based distributed solutions in a „**service-oriented architecture (SOA)“.**

Service vs. Component

Services are specifications of components. One service may be implemented by different components (supplied by different suppliers). A component may implement (provide) several services.

Services (and components) are encapsulated and possess clearly defined **interfaces**. The **interfaces** are **small APIs** consisting of types, attributes, methods with defined responsibilities.

Services are interfaces provided by components.

The term component in the stronger sense (as used in component-based systems) denotes constituents of software which are:

- **encapsulated** (information hiding)
- **Transparent** wrt. to location and technical platform
- **coarse-grained**
- **usable by other components** via interfaces
- **loosely coupled** (by synchronous or asynchronous calls)

SERVICE-ORIENTED ARCHITECTURES (SOA)

”**Service-Oriented Architecture (SOA)**“ is a general approach to structure systems along business functions and business processes. (SOA is another **architectural style**.)

Business functions are defined as „**services**“.

Business processes are implemented by „**orchestrating**“those services.

SOA vs. Business-IT-Alignment

The goal of SOA is to achieve flexibility in application construction by aligning the **business view** with the **technical** implementation („**Business-IT-Alignment**“) in a robust, scalable, efficient, and secure manner.

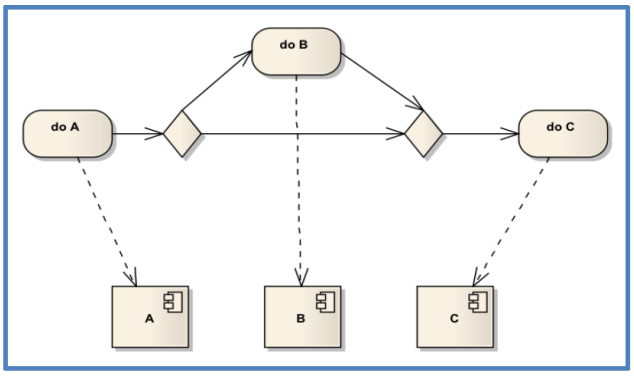
The business view is defined in **business services** which are implemented by one or more **IT-services**.

The main **characteristic of SOA** is the differentiation between

- **a process layer** (called orchestration) and
- **an implementation layer** (called service provision), that supplies the functionalities as reusable services.

Orchestration

Service Provision



SOA Reference Architecture

The introduction of Web Services requires

- a good understanding of the business processes
- a suitable infrastructure which is platform-independent and uniform.

This is supported by the **SOA reference architecture**, which adheres to the layered architectural style.

W3C-WEB-SERVICES

To enable service-orientation, **an infrastructure is needed** to, for example:

- manage all services
- deliver information about existing services
- assure security of service usage
- handle exceptions
- log execution

An important technology, to orchestrate and execute services in the Web (Web services) is supplied by **W3C (W3C-Webservices).**

W3C-Webservices are closed, self-describing, and modular services, which are described by **XML-based standards.**

Web Services are an approach for **realizing service-oriented architectures over the Web**:

– All information is **serialized in XML**.

– The components may be **distributed over different nodes**.

– Internet protocols are used for communication (**HTTP**).

Web Services are **atomic** and **stateless**. **Sessions** (long-living transactions) have to be handled **explicitly**.

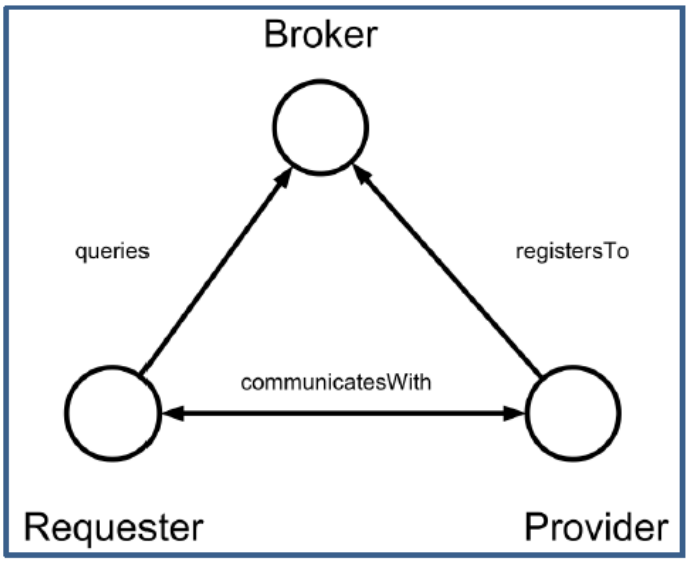
Since Web Services are transferred via HTTP, they can easily **pass firewalls**.

The **vision of Web Services** is „**loose coupling**“:

Applications may look for providers of needed functionality, choose between alternatives, and immediately use them at runtime. Thus, **binding between requester and provider happens at runtime**.

Requester-Provider-Broker

Web-Services in general follow a „**Requester-Provider-Broker**“-discipline, which is also an architectural style.

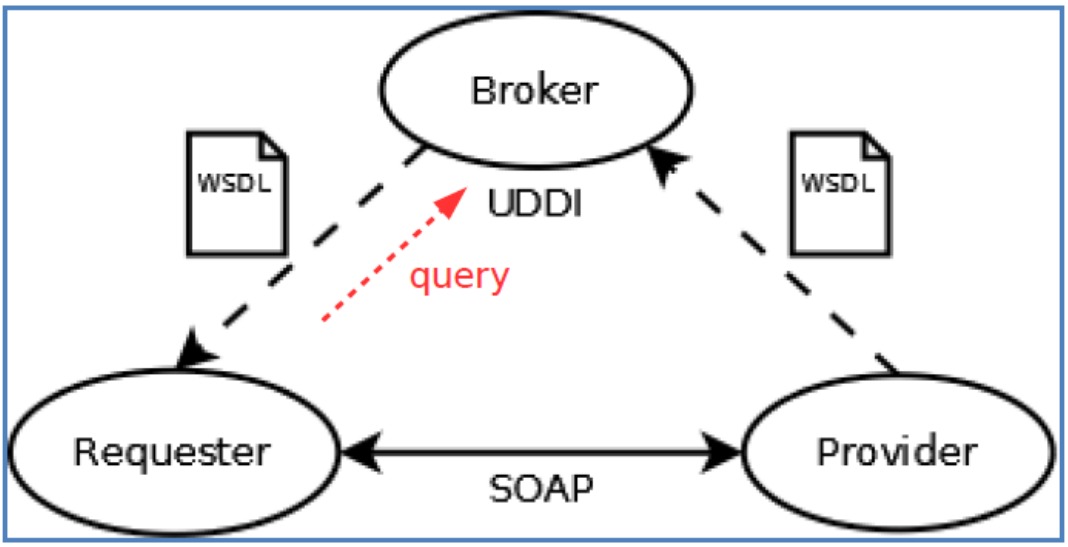


Standardization is done by **W3C** (for the technical standards) and **OASIS** (for special domains (e.g., e-Business, Web-Services)).

Parts of **W3C's Web service technology** are:

- **SOAP** (originally for Simple Object Access Protocol): **Defines messages** for web services (message format: XML).
- **WSDL** (Web Services Description Language): **XML-based interface definition** language for web services.
- **UDDI** (Universal Description, Discovery and Integration): **Protocol for service retrieval**, including XML-based registry.

Requester-Broker-Provider (W3C)



W3C/OASIS technology for Web Services is **based on XML**. In practice, all XML-code is generated by tools from higher-level descriptions.

SOAP

SOAP is an XML language for supporting arbitrary information via XML.

SOAP is a generic message format.

SOAP is

- **extendable** (e.g., to support more security) and
- **independent** of the transport protocol (usually HTTP)

SOAP supports extension of serialized data by **addresses** and further **transport information**.

SOAP is well-suited

- for **simple data transport** as well as
- for supporting the call of service methods by **Remote Procedure Calls (RPCs).**

Next coming:

- (A1) the SOAP message format in general
- (A2) SOAP conventions how to encode RPCs.

(A1) SOAP MESSAGE FORMAT

A **SOAP message consists of**

- **several headers** containing meta-information (addressee, version number, …)
- **one body** containing the application dependent data

A message **may have several headers**, since it may **pass several intermediary nodes**.

Each intermediary removes one header on the way to the final addressee.

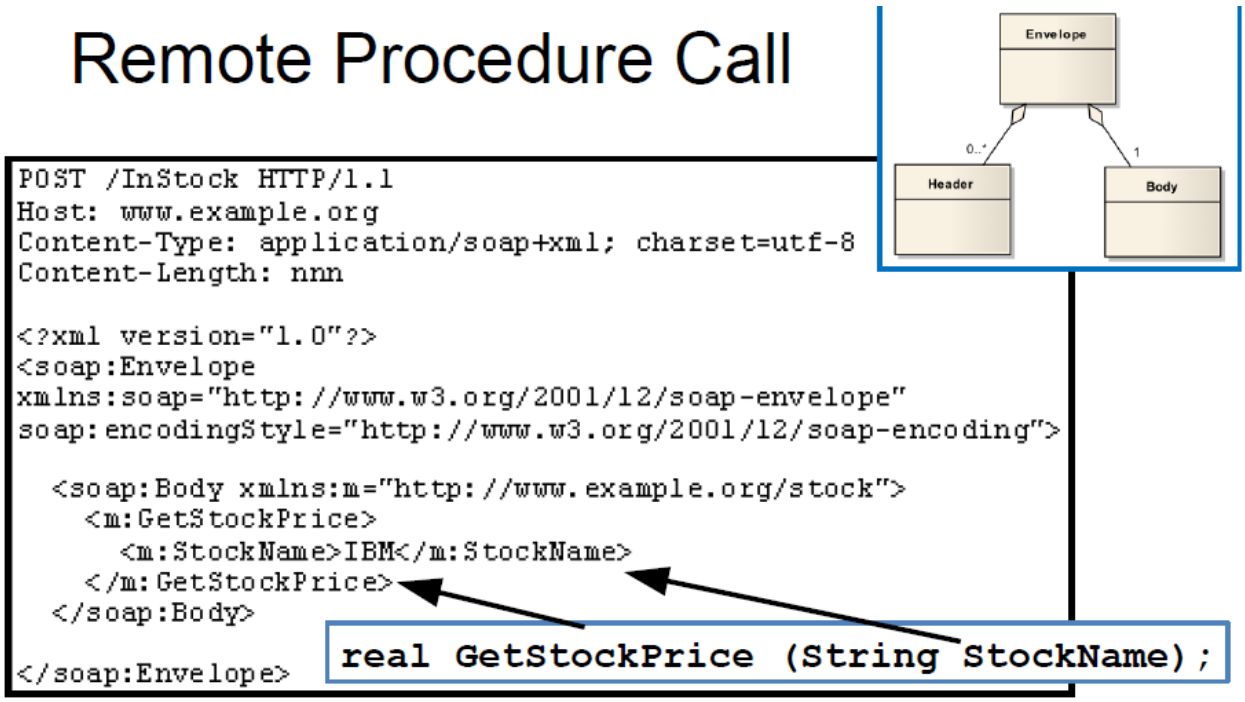
(A2) SOAP CONVENTIONS

RPC Conventions

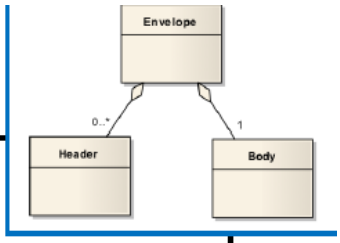
The body of a SOAP message may be a remote procedure call of a method.

Method calls and method responses are encoded as XML elements whose content are the input/output parameters.

- **Call**: The parent element is named after the method, and the child elements are the arguments.
- **response**: The parent element is named after the method and suffixed by response, and the result is the child element



RPC Response




```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

  <soap:Body xmlns:m="http://www.example.org/stock">
    <m:GetStockPriceResponse>
      <m:Price>34.5</m:Price>
    </m:GetStockPriceResponse>
  </soap:Body>

</soap:Envelope>
```

```
real GetStockPrice (String StockName);
```

Flow of Control:

– The **provider** gets a **SOAP request** and passes it as a concrete method call to the server object.

– The **result** is sent back as a **SOAP response**.

WSDL

WSDL (Web Service Description Language) is an **XML language for the service interface**.

WSDL: Technical Aspects

A **WSDL description** defines the **interfaces**, the **transport protocols**, and the **technical aspects** of a service.

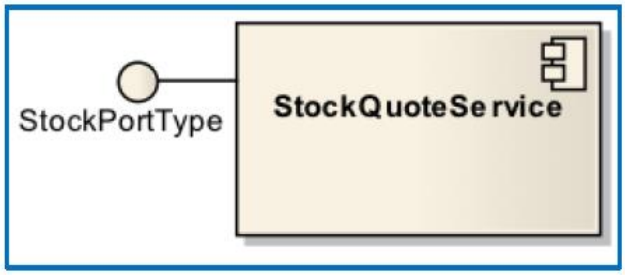
Requester and provider must have a compatible **understanding of the semantics**. (Semantics is not described.) WSDL assumes the **realization of method calls by messages**.

Parts of a WSDL Definition

- **types**: defines the data types used
- **message**: defines all messages used
- **port type** (version 2.0ff: interface): defines abstractly the operations performed (i.e. the interface)
- **binding**: assigns a data format (e.g. RPC) and a communication protocol (e.g. SOAP via HTTP) to the operations
- **service**: identifies the implementation of the Web service (e.g. a URI)

Websoa slide 50-58 are examples of SOA

StockQuoteService: Interface, Messages



Interface

```
1  interface stockPortType {
2    float GetLastTradePrice
3          (StringTickerSymbol);
4  }
```

Messages

```
1  GetLastTradePriceInput  // call
2
3  GetLastTradePriceOutput // return
```

UDDI

UDDI (**Universal Description, Discovery and Integration**) supports **registration and search** of web services at a broker node.

UDDI

– is a **directory service** (not a language (!))

– **contains** Web service interfaces in **WSDL**

– **communicates using SOAP**

UDDI is an OASIS Standard

UDDI differentiates between three kinds of „services pages“

- white: information about companies
- yellow: information about contents
- green: information about technology

RESTFUL SERVICES

The mechanisms used by **W3C-Services** (RPC via SOAP and WSDL) are **very time-consuming**.

REST is a light-weight alternative, which is based on the mechanisms of the Web.

REST is primarily used inside **closed applications**

REST (Representational State Transfer) is **not a standard**. REST builds on Web standards, like

- URIs
- MIME-Types
- HTTP
- HTML, XML, etc.

REST is an **architectural style** for communication over the Web.

Communication is done by **linked resources** (instead of RPCs) ”**Hypermedia as an ‚engine’**“

Resources are manipulated by their representations. The **hyper-link structure** defines the **control flow**.

The interface between client and server consists only of:

– the HTTP-Operations GET, PUT, POST, and DELETE and
– URLs.

REST supports the **Client-Server style**. The client sends a **request**, the server returns a **response** (which is a **representation** of a resource). Both operations are done using **HTTP**. The resource is described by a URI.

The resource representation is a document that describes the resource in some MIME type (e.g., HTML, XML). **URI links in this representation lead to further requests.**

Design Decisions of REST:

– The fundamental communication unit is the resource.
– A document is a representation of a resource.
– By loading a document the client changes its state „Representational State Transfer“
– REST is used to implement Web services. „ RESTful Services“

REST: Example

```
ValueAndUnit GetWaterGaugeData (Time t);
```

```
1  GET /WaterGaugeData?time=201001261753000100 HTTP/1.1
2  Host: www.pegel-kaub.de
```

```
1  HTTP/1.1 200 OK
2  Content-Type: text/xml; charset="utf-8"
3  Content-Length: 266
4
5  <?xml version="1.0" encoding="UTF-8"?>
6  <m:GetWaterGaugeDataResponse
7    xmlns:m="http://www.pegel-kaub.de/watergaugedata.wsdl">
8    <m:value>169.5</m:value>
9    <m:unit>cm</m:unit>
10 </m:GetWaterGaugeDataResponse>
```

By giving xmlns:m, the namespace MathML will be connected to the html Element

[Bruckhoff2011PEW]

Page content is presented in **machine readable form**. **XML** is often used as data exchange format.

RESTful Web Services - JSON

JavaScript Object Notation

- Any valid JSON document is at the same time a valid JavaScript fragment that can be interpreted by a function **JSON.parse(…) or eval()**
- In contrast to **XML reduced overhead**, less memory consumption, less transfer amount, less bandwidth usage
- **Simple syntax**, easy to generate and to process

RESTful Web Services are used via JavaScript **XMLHttpRequest**

- Advantage: Data can be loaded dynamically without need to reload the complete page.
- XMLHttpRequests in general use asynchronous communication - the script doesn't have to wait for the response
- All HTTP methods can be used
- It offers timeouts, transfer of binary data, and some security means

- Recent browsers support XMLHttpRequest in their embedded JavaScript interpreters (XHR objects).

The so-called **same-origin-policy** is a security feature of recent Web browsers.

JSONP (JSON with Padding) circumvents the Same Origin- Policy.

- The client adds **the name** of a JavaScript callback **function**, e.g. as GET parameter
- In the response, the **server** encapsulates the **JSON result into a function call**
- In the **client**, the **callback function has to be defined**.
- To call the function, the **client modifies the DOM** tree of the current page by script-element-injection. **A <script> element is added and gets executed.**
- Recent JavaScript libraries, e.g. **JQuery**, provide **support for JSONP**. A callback function with random name is created „on-the-fly".

XMLHttpRequest - CORS

Cross Origin Resource Sharing (CORS) is a technology that enables JavaScript **access to resources** on **foreign domains**. The **client adds** an **Origin field** to the HTTP request header. The **value** of the field is the **requesting domain**. The **server** checks the **Origin field** to decide **if access is granted**.

In the HTTP **response** header, the server adds the **field Access-Control-Allow-Origin** which contains all **domains for which access is granted**. Then client can check whether its own domain is listed.

Further **security measure** can be put into effect by the **header field Access-Control-Allow-Methods**. The field lists **all admissible HTTP methods** for a resource.

REST Server with JavaEE

JavaEE specifies **JAX-RS** (Java API for RESTful Services). Grouping of multiple REST resources in a JAX-RS-Application

REST is simpler than SOAP

- SOAP packages data, which are sent via HTTP
- **REST uses HTTP directly**

Advantages:

- **simplified communication**
- **direct use of HTTP**
- **scalability**
- **performance**
- **reliability**

REpresentational **S**tate **T**ransfer

Main concept of REST is that a **URL represents exactly one resource**. Different resources have different URL paths. HTTP methods are used to execute **CRUD** operations on resources

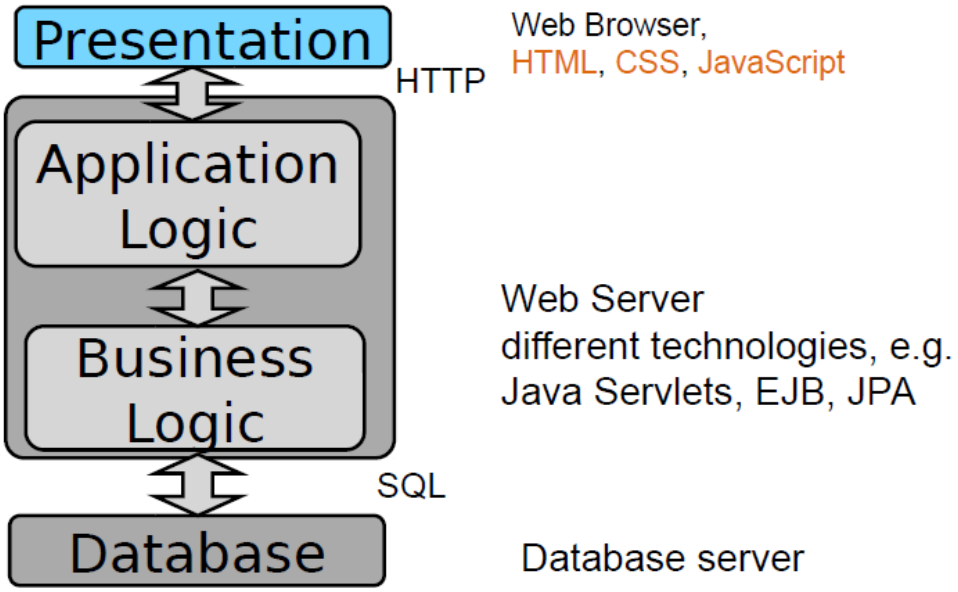
– **Create** by means of **PUT**

– **Read** by means of **GET**

– **Update** by means of **POST**

– **Delete** by means of **DELETE**

4 Tier Application Architecture



Web Browser,
HTML, CSS, JavaScript

Web Server
different technologies, e.g.
Java Servlets, EJB, JPA

Database server

BPMN

Orchestration

The **process layer** implements processes by **calling** the appropriate **services** („**Orchestration**“).

The **orchestration is itself an executable process**, which controls the involved services in a centralized manner.

BPMN (**Business Process Model and Notation**) is a visual language defined by (the BPM-initiative) of OMG to describe business processes.

BPMN is well documented, including

- notation
- exchange format
- formal semantics

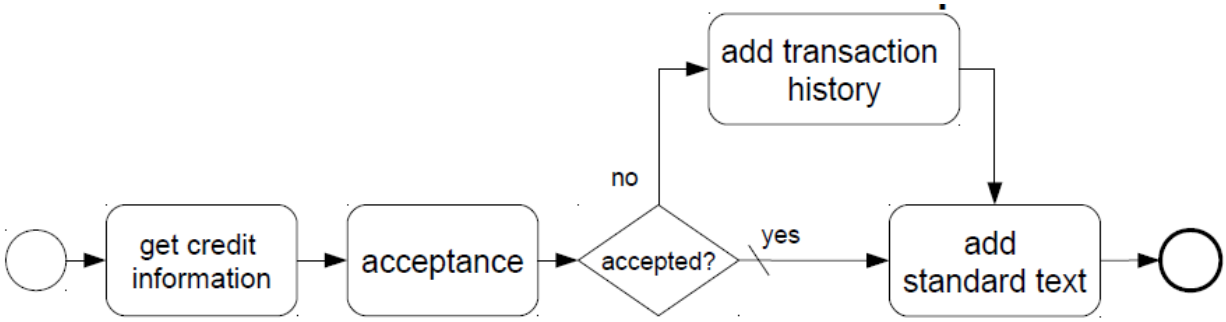
BPMN describes

- unstructured control flow and
- data flow.

BPMN is

- intuitive (business user),
- precise (technical user),
- extendible and
- supports use of icons

BPMN is a Flowchart-based notation to define business processes. Also, it is a convention by different providers of notations.



BPMN Goal: Business process modeling notation with the following characteristics:

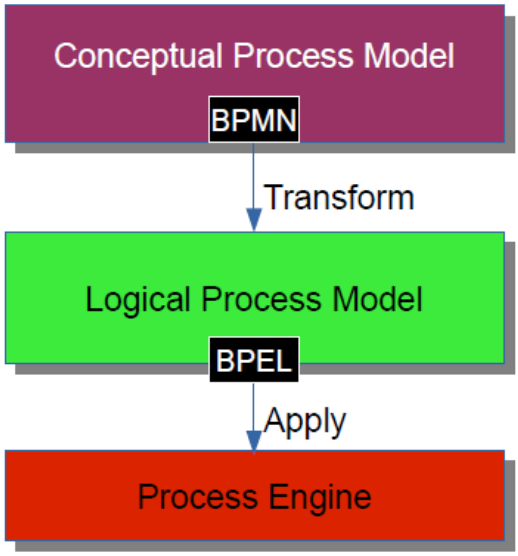
- Combination of **graphical elements** and **additional information** (attributes).
- Generate **executable process** (e.g. BPEL) from BPMN model.
- Usable for general **business purposes**.
- Acceptable and **suitable for companies**.
- **Agnostic methodology**:
  - Hints for **purpose** and **level of detail** of modeling.
  - Full BPMN notation is complex, in general only a subset is needed.

Execution of BPMN Processes using BPEL representation

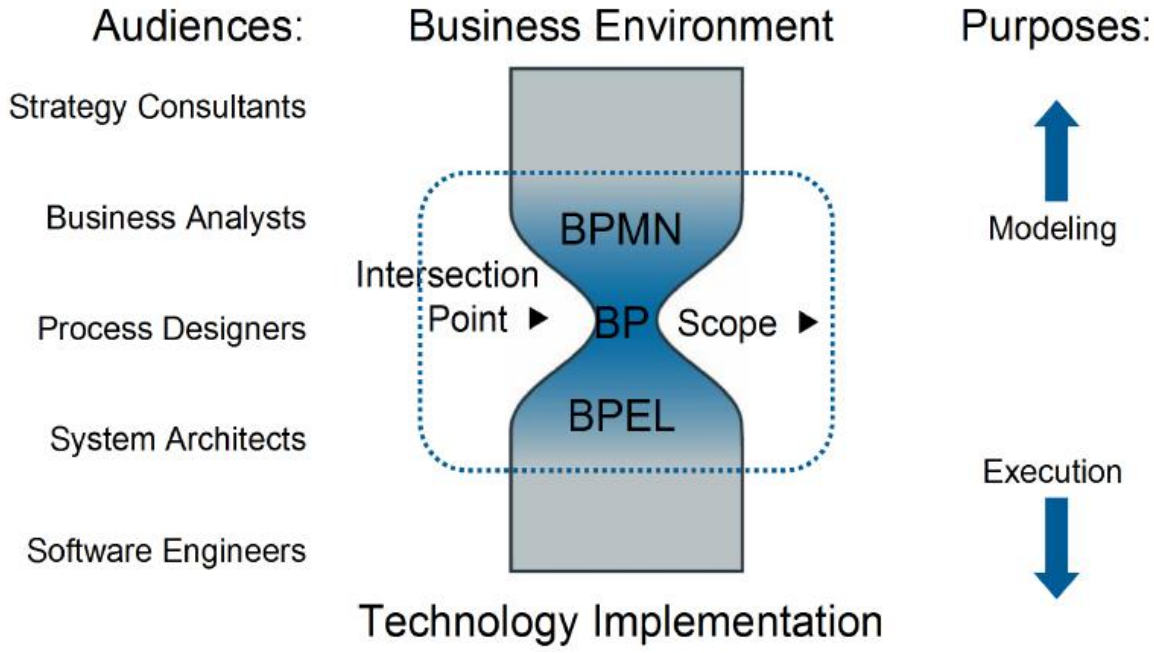
- Comprehensible
- Graphical

- XML-based
- Executable
- Execution environment

BP directly in **BPEL Engine** loadable, without any changes made by **human interpretation / transformation**.



Modeling vs. Execution



BPMN 1.x fills a Gap

- **BPEL concentrates on syntax and operational semantics**.
- Goal: Fast time to market → visual representation ignored.
  - → **Hard to use**.
- BPMN 1.x fills this gap.
- Non IT experts can communicate about business processes.
- Huge improvement in BPM technology.

BPEL – BPMN History

BPEL as starting point for BPMN:

- Goal: **complete bijective transformation**.

Automatic **generation of BPEL**:

- e.g. by using a graphical modeling tool from BPMN to BPEL.
- Problem: despite **different expressive power** of the languages, **high overlap**.
– e.g. data aspects are neglected in BPMN.

Important characteristics of BPMN 2.0

- Operational semantics.
- “native” XML interchange format.
- Meta model.
- Event- & Exception-Handler (BPEL counterpart).
- Choreography extension.
- Pattern-based mapping to BPEL (subset of BPMN):
- BPMN 2.0 includes significant subset which is isomorphic to BPEL.
- → Visual representation of BPEL.

Disadvantages of BPMN 2.0

- **High complexity**.
– Guarantee executability is always complex.
- Only a subset of BPMN 2.0 can be mapped easily to BPEL.
– BPMN processes can be modeled, that don't have a canonical representation in BPEL.
- Some functions don't have a runtime environment yet.
– e.g. corresponding domain outside the scope of BPEL (see BPEL4Chor).

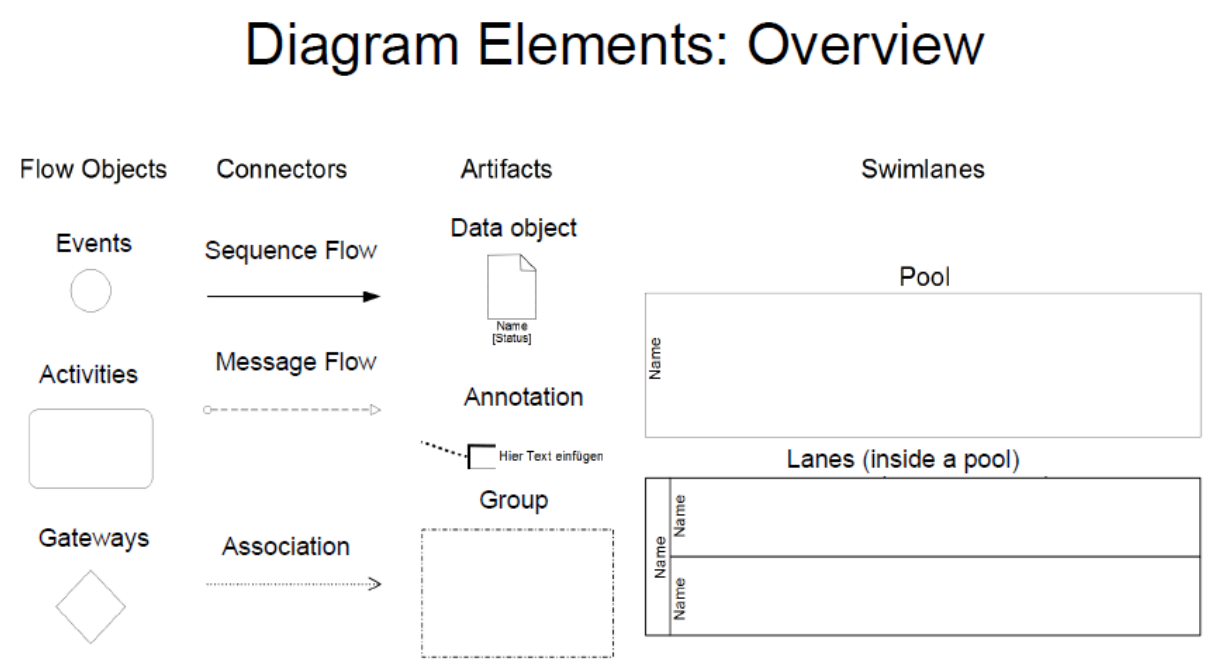
BPMN 2.0 vs. BPEL: Possible Views

Viewpoint 1 (**BPEL-centric**):

- Subset of BPMN 2.0 is isomorphic to BPEL.
– Canonically transformed to BPEL and executable by BPEL Engines.
- Suitable subset: visual layer on top of BPEL.

Viewpoint 2 (**BPMN-centric**):

- BPMN 2.0: Process language with well-defined operational semantics by using BPEL.
- BPMN 2.0 Engine can created, without using a separate BPEL Engine.



Activity in BPMN: In business processes conducted work.

- Atomic or non-atomic (composed).
- Kinds of activities: Sub-process and Task.
- Activities are visualized as rounded rectangle.
- Single or as intern defined multiple repetition executable.

Tasks Purpose: atomic activity.

Used, if process is not expressed in a higher level of detail.

Special kinds of tasks for sending, receiving, or user-oriented operations.

Marks and symbols can be assigned to tasks.

Important:

- Redundancy: The presented information should also be visible without marks. (Marks will not be transformed → loss of information)
- Confusion freedom: Marks should not lead to confusion.

Sub-Process: Joint activity set inside process.

- Collapsed sub-process: **Marked with "+"** at the lower end of the symbol. No details are visible.
- Expanded sub-process: **Details visible** inside sub-process borders.
- Makes **hierarchically process development** possible.

Events

Start Events

Start events define, where process starts. Different triggers: Indicate under which circumstances the process starts:

• "None": Start of a sub-process or undefined start circumstances.

• Message: Event triggered by a message.

• Timer: Event triggered by a timeout.

• Conditional: Triggered by breaking or activating a rule. (e.g. breaking speed limit). (document sign)

• Multiple (OR): Every included start event starts process. (pentagon shape – **double check this**)

• Multiple (AND): All included start events required to start process. (+ sign)

Intermediate Events

In the process flow placed events represent things that happen during execution.

- They can
  - Represent reactions to an event (e.g. receiving a message).
  - Indicate an occurrence of an event (e.g. sending a message).

- **Throwing intermediate events**: triggered by the process.
  - → Token activates an event and goes on, no abort!
- **Catching intermediate events**: noted at the border of the to be interrupted activity
  - → Abort of the activity!

Throwing Intermediate Events

• Part of the normal flow.

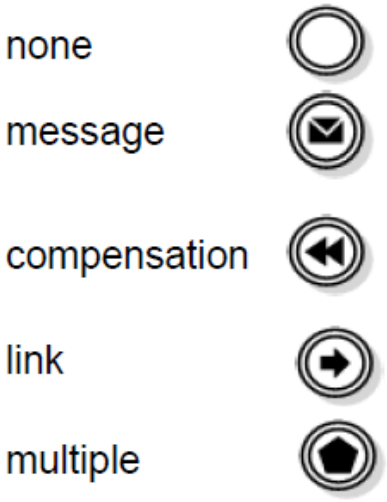
• Presentation: filled symbol.

• If a **throwing intermediate event** is reached in the **sequence flow**, the process triggers the event (e.g. sending a message) **and goes on** immediately.

• Different **triggers** (same as at end event). Also:

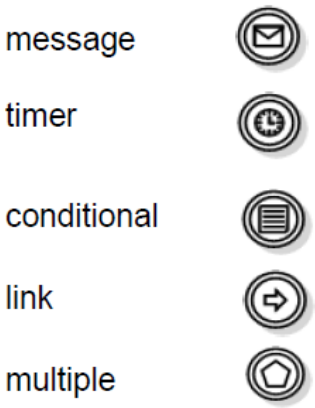
– **Compensation**: Defines reverse transaction of previous tasks (e.g. return remittance of money)

– **Link event**:



Catching Intermediate Events

- **Intermediate events** occur after the start and before the end of the process.
- Part of the normal flow / attachable to activity boundary
- Presentation: **not filled symbol.**
- If a **catching intermediate event** is reached in the sequence flow, the process goes on only if the corresponding event happens (e.g. receiving a message).
- Different **triggers** specify the circumstances of an event (same as at start event).



Intermediate Events attached to an Activity Boundary

Events that are attached to an activity boundary represent triggers.

- → can interrupt activities.
- Attachable to tasks and sub-processes.
- Are used for example to handle errors or exceptions.
- Work inside the activity will be terminated and flow continues at the event.
- Timer, errors or messages can be the triggers.

Attached Intermediate Events

- Attachable to activity boundary.
- Presentation: **Not filled symbol.**
- Reacts to corresponding event in case of attached intermediate event (e.g receiving a message).
- Cancels activity in case of drawn **through boundary line**.

- Does not cancel activity in case of **dashed boundary line**.
- Different triggers specify the circumstances of an event (same as at start event) and:

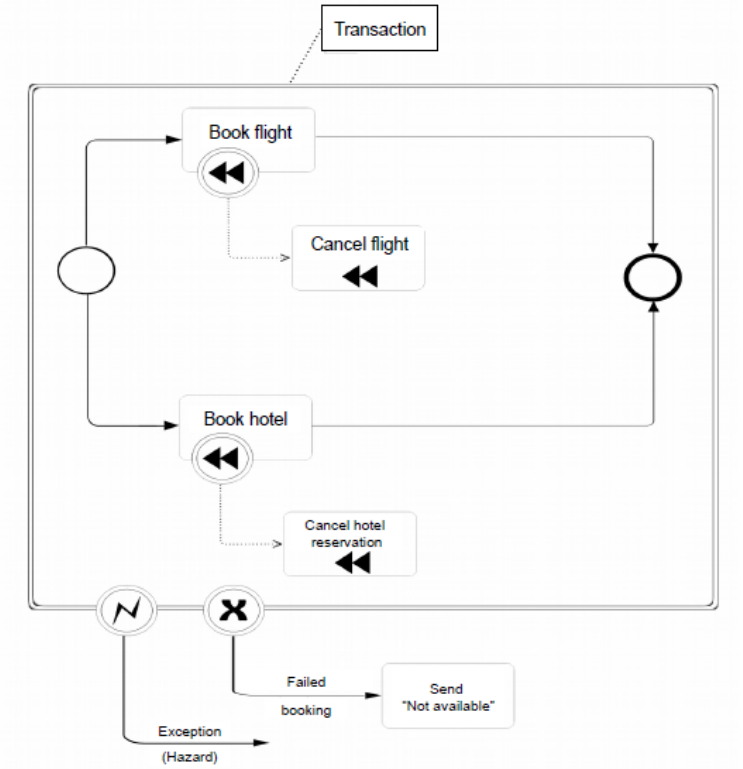
– Cancel, error and compensation

Compensation

For **cancel**: Flow that starts with a cancel event.

For **error** (exception): Flow, that starts with an error event (no compensation).

For **compensation**: Activity with corresponding mark outside of the normal flow.



Transactions: Activities with two boundaries. Supported by transaction protocol (e.g. WS-Transaction). On successful termination: Outgoing sequence flow goes on.

On cancel: Compensation for included activities (atomicity of transactions).

Link Events: Graphical simplification to connect sequence flow connectors at different diagram parts.

End Events

• End events indicate the end of the process.

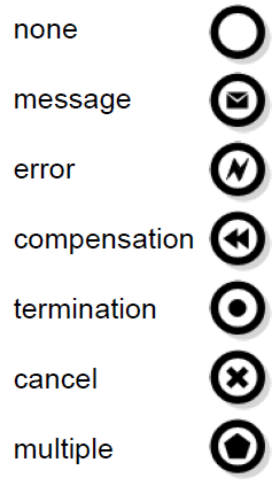
• Different kinds of events indicate how the process ends. Analog to intermediate events. Especially:

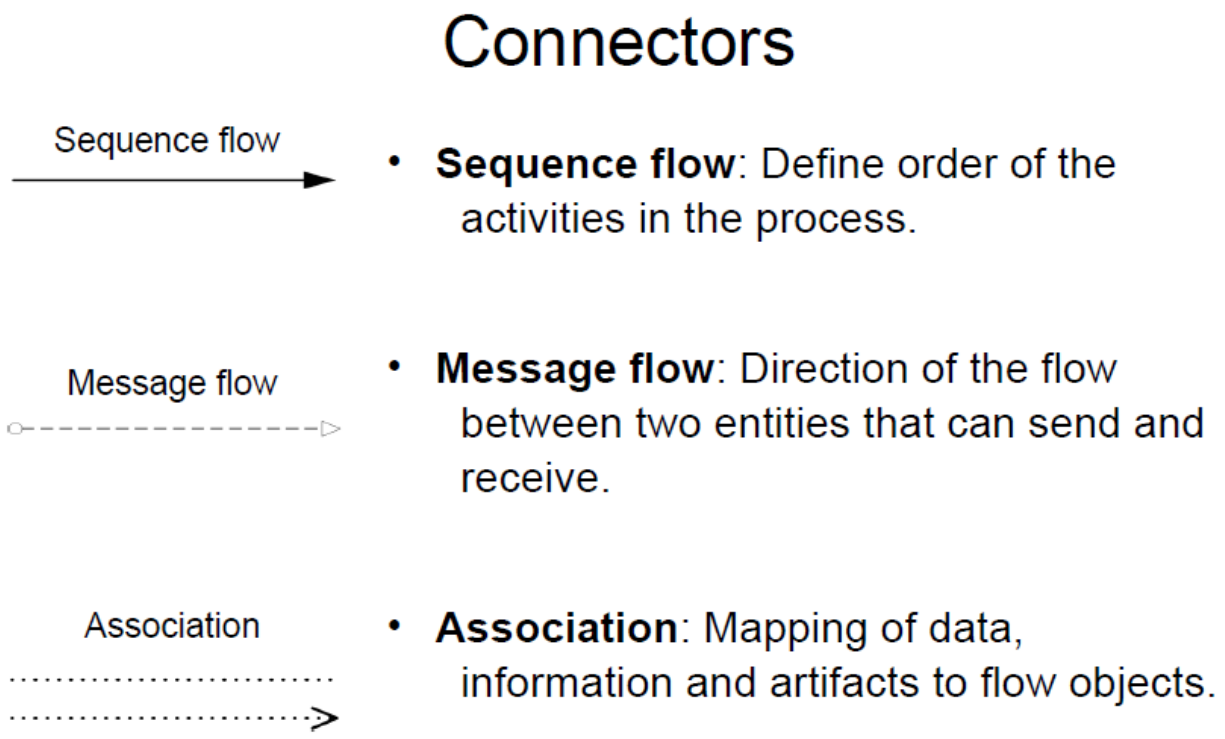
– **None event**: Defines the end of an intermediate process or an undefined end of a process.

– **Termination:** Terminates all active parts of the process

– **Cancel:** Only in context of transactions

– **Link event** from BPMN 1.0 does **not exist anymore** in BPMN 2.0.





Sequence Flow

- Sequence flow: Define order of activities in the process.
- Starting point and end point are: events, activities or gateways.
- Sequence flow cannot pass sub-process boundaries or pool boundaries.

Normal Flow

Normal sequence flow: Flow which …

- starts at start event, flows through activities (eventually alternative or parallel paths) and ends in an end event.
- not includes exception flow or compensation flow.

Conditional Sequence Flow

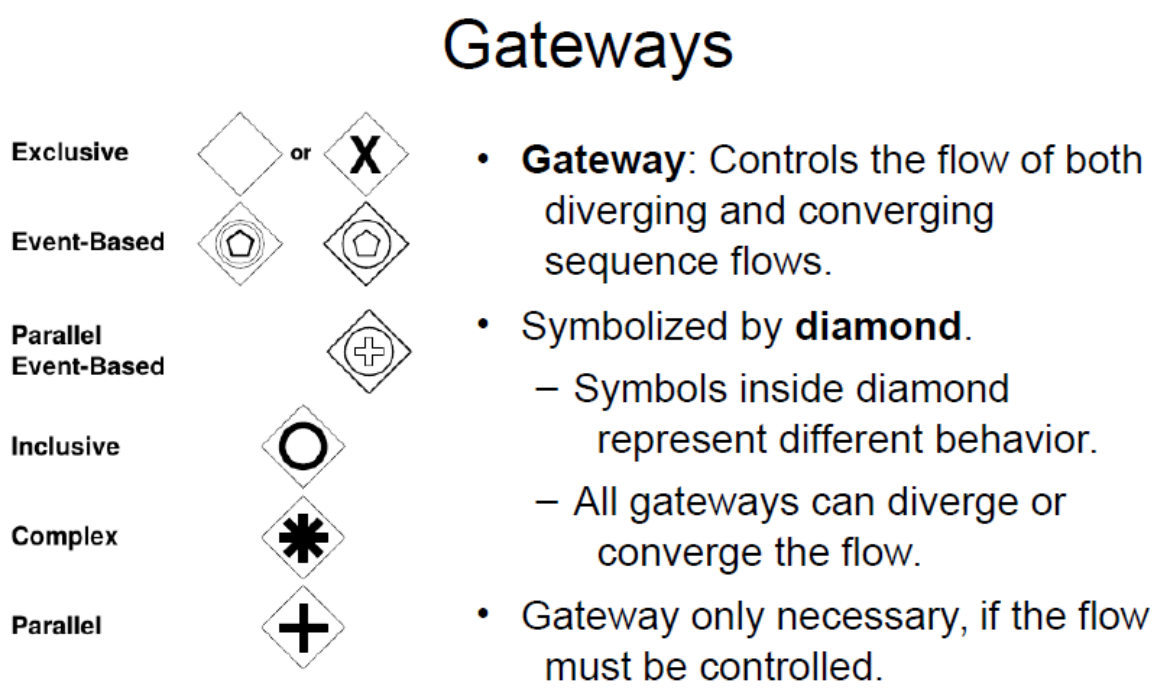
Sequence flow CAN have defined condition when it quits an activity.

• Activity must have **at least two** outgoing sequence flows. Condition has to be fulfilled so that the flow can continue.

• Diamond symbol shows that the sequence flow has a condition. **At least one** outgoing sequence flow during the execution of the process has to be able to choose.

Default Sequence Flow

In case of **exclusive or inclusive gateway**: declare outgoing sequence flow as **standard path** (crossbar). Standard path is chosen if **all other conditions are not fulfilled**.



Exclusive Gateway (decision): Alternative paths based on defined conditions. Optional symbolized by "X" inside the diamond.

Event-based Gateway: Branching point where the alternative paths are based on events that occur.

- Characteristic: Multiple parallel occurrence of intermediate events.
- Event that follow on the gateway diamond defines the selected path.
– First happening event "wins" the decision.

Inclusive Gateway: Decisions with more than one (simultaneous) possible event.

- Identified by symbol "O".
- Normally terminated by corresponding converging inclusive gateway.

Complex gateway: Decision at which detailed definitions of behavior can be added.

- Marked by an asterisk.
- Defined for diverging and converging gateways.

Parallel Gateway: Spot where parallel paths can defined.

- Usually not necessary to fork the path.
- Can be used because of methodical reasons.
- Marked by "+"-Symbol.
- Can also be used to synchronize parallel paths.

Message Flow

- Message flow: Defines direction of the messages between two participants of the process.
  - Participants are represented by separate pools.
- A message flow can be attached to the border of a pool or to the border of an object inside a pool.
- A message flow between two objects inside the same pool is not allowed.

Associations

- Association: Is used to link two objects to each other (e.g. artifacts and activities).
- Associations can define how data leave or enter an activity.
- → Textual annotation assignable to an object.

Summary: BPMN: Main Concepts

Elements of BPMN diagrams are:

- Activities, i.e., something to be done
- Data, to be read or processed
- Events, i.e., something that happens
- Flows, execution sequencing, message flow (catching or throwing), data flow
- Gateways, that change the control flow

BPEL

XML-based execution language.

- Specification of business processes and interaction protocols based on web services.
- Based on different XML standards ("layer")
- Integration and coordination of activities as web services (by using WSDL)
- Execution of a BPEL process by process engine.

Key aspects of BPEL

BPEL is supported by most middleware vendors:

- Combination of two model paradigms.
- Avoiding fragmentation of the workflow market.
- The runtime standard.
- Well defined syntax and operational semantics.
- Portability, vendor-independent.

Disadvantage: no "beautiful" language.

Web Services Business Process Execution Language (WS-BPEL)

Properties:

- BPEL is an **XML language**
- BPEL **describes processes** by hierarchical and structured control flow
- BPEL processes **orchestrate web services**, which are described by WSDL
- BPEL **processes are web services** themselves

BPEL descriptions are **interpreted by a "BPEL Engine".**

Example

– Apache ODE (Orchestration Director Engine)

BPEL Execution: Overview

- Basics model deployment
  - Native meta model of a workflow engine and model transformation
- BPEL and Transformation: BPMN 2 to BPEL 2

– Short introduction BPEL, activities
– Events
– Structured Activities

Deployment vs. Metamodel

Deployment: Get process model productive.

• e.g. get ready for execution.

Central role: Data format for the models.

Defined as a metamodel: Definition of a modeling language that is a model itself.

“Native metamodel”: Meta model for defining the internal data format model of a workflow engine.

• “Native” BPEL (respectively BPMN) engine: BPEL (respectively BPMN) is internal (“native”) metamodel.

Solution: Extension of the Target-metamodel

- Models, that are specified in metamodel M2 have to perfectly supported by the engine with another metamodel M1.
- Approach to a solution: Add constructs from M2, that are difficult to emulate in M1, to M1.
- BPEL is designed to be extensible: New constructs can be added.
  - → Optimal mapping in different metamodels.
- Extended variant of a M1 engine (“M1++ engine”) can for example support process models of metamodels M2, M3, …, Mn.

BPEL: Structure

Elements:

– Root element: process
– Child elements: extensions, import, partnerLinks, messageExchanges, variables, correlationSets, faultHandlers, event-Handlers
– Groups: activities

BPEL Activity elements:

– Exchange of messages: receive, reply, pick
– Call of a service: invoke
– Assignment to variables: assign
– Signaling of an error: throw
– Simple control flow: exit, wait, empty
– Structured control flow: sequence, if, while, repeatUntil, forEach
– Concurrency: flow

BPEL Example: partnerLinks

```
<process name="insuranceSelectionProcess"
targetNamespace="http://packtpub.com/bpel/example/ "
xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
 xmlns:ins="http://packtpub.com/bpel/insurance/"
 xmlns:com="http://packtpub.com/bpel/company/" >
   <partnerLinks>
      <partnerLink name="client"
                   partnerLinkType="com:selectionLT"
                   myRole="insuranceSelectionService"/>
      <partnerLink name="insuranceA"
                   partnerLinkType="ins:insuranceLT"
                   myRole="insuranceRequester"
                   partnerRole="insuranceService"/>
…
</partnerLinks>
```

partnerRole not defined (because incoming communication)

BPEL Example: Variables

```
<variables>
 <!-- input for BPEL process -->
 <variable name="InsuranceRequest"
          messageType="ins:InsuranceRequestMessage"/>
 <!-- output from insurance A -->
 <variable name="InsuranceAResponse"
          messageType="ins:InsuranceResponseMessage"/>
…
 <!-- output from BPEL process -->
 <variable name="InsuranceSelectionResponse"
          messageType="ins:InsuranceResponseMessage"/>
</variables>
```

Message name of the partner process definition

BPEL Example: Process (1)

```
<sequence>
      <!-- Receive the initial request from client -->
      <receive partnerLink="client"
               portType="com:InsuranceSelectionPT"
               operation="SelectInsurance"
               variable="InsuranceRequest"
               createInstance="yes" />
     <!-- Make concurrent invocations to Insurance A and B -->
      <flow>
         <!-- Invoke Insurance A web service -->
         <invoke partnerLink="insuranceA"
                 portType="ins:ComputeInsurancePremiumPT"
                 operation="ComputeInsurancePremium"
                 inputVariable="InsuranceRequest"
                 outputVariable="InsuranceAResponse" />
         ...
       </flow>
```

BPEL Example: Process (2)

```
<!-- Select the best offer and construct the response -->
      <switch>
        <case condition="bpws:getVariableData('InsuranceAResponse',
                        'confirmationData','/confirmationData/Amount')
                     <= bpws:getVariableData('InsuranceBResponse',
                        'confirmationData','/confirmationData/Amount')">
          <!-- Select Insurance A -->
          <assign>
            <copy>
              <from variable="InsuranceAResponse" />
              <to variable="InsuranceSelectionResponse" />
            </copy>
          </assign>
        </case>
        <otherwise> … </otherwise>
      </switch>
      <!-- Send a response to the client -->
<reply partnerLink="client" portType="com:InsuranceSelectionPT"
          operation="SelectInsurance" variable="InsuranceSelectionResponse"/>
   </sequence>
</process>
```

Quality Assurance for Web Applications

INTRODUCTION

Quality Assurance (QA) comprises all planned and systematically performed activities to assure an acceptable level of trust that something meets given requirements. QA is a **permanent activity**.

Quality Assurance applies to

- **processes** as well as
- **Products**.

Testing denotes the **experimental** assessment of concrete artifacts with respect to their correctness and/or their quality properties. Testing is a **permanent activity.**

Every test has a **goal**. The goal is described by the required **quality characteristic** and by the **test object**.

Developers should not test their own products (except for unit and integration tests).

Exploring the sources of **risks** may point to defects more directly than basing tests mainly on requirements. It is impossible to achieve complete test coverage. That's why testing should be done **risk-based**.

There are (often) **special risks** for WAs:

- **lack of skill of the team -** poor acceptance of methodologies
- **immaturity of methods -** for some new technologies there are no tools yet
- **dominance of change -** tests have to be adapted to changing requirements
- **immature third party components -** dependence on unvalidated sources

For Web Applications, there is a shift **from validation against the technical specifications** towards **validation against user expectations**.

A strongly **iterative evolutionary development process** reduces the risk, since smaller parts are frequently tested on all test levels. The permanent use of **user stories** supports test-driven development.

Test-Driven Development (TDD, Kent Beck 2003) uses tests as manifestations of requirements (and user stories):

- Define **automated test cases** (or test suites), **first**.
- **produce code** to pass the test
- **refactor the code** to conform to standards
- do **regression tests**

TDD fits well to **agile development** methods.

In the context of TDD, tests have to be executed multiple times (**regression tests**). Automated **regression tests** help to protect existing functionality.

Testing must be supported by **tools** (e.g., **junit**) to support repetitive execution of tests automatically. **Automation** allows to run more tests in less time.

Web Applications have more **testing dimensions:**

- **content** (to be done by experts): proofreading, spell checking, checking of domain specific constraints
- **hypertext structure** (to be done by tools): check accessibility of pages, broken links, check state change when going back in history
- **aesthetics:** review presentation, let experts evaluate adequacy (methods from the publishing industry are needed)
- **multi-platform delivery:** simulate devices

– **multilingualism and usability:** recognize cultural interdependencies, do excessive beta testing

There is a broad range of **testing techniques** to be applied:

– **functional** testing
– **usability** testing
– **link** testing
– **security** testing
– **load and stress** testing
– **target platform** testing

All tests must be **repeatable** (for regression testing)

The goal of **functional testing** is to **find errors**.

An **error** is the difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition.

Usability testing evaluates the ease-of-use of different Web designs, the overall layout, and the navigations of a Web application for different users.

Usability testing is usually done **in a laboratory setting**, using workrooms fitted with one-way glass, cameras, etc.

The **Web Accessibility Initiative (WAI)** has developed approaches for evaluating Web sites for accessibility.

Link testing follows systematically all links and produces a **link graph** (**sitemap**). There may be isolated pages which are not cited any more.

Security is a critical property for Web Applications:

– **encrypt** confidential information

– verify user **identities**

– regulate **access**

It is important to **simulate attacks** systematically.

Load generators may generate requests which are sent to the Web application concurrently by simulated users.

Objectives:

– **Load tests:** does the system meet the required response time?
– **Stress test:** does the system react in a controlled way when stressed?
– **Continuous testing:** does the system work as expected over a lengthy period?

Run the tests on **different target systems** with different browsers and different versions. Especially test against the **known weaknesses** of the browsers.

One may also use **metrics** to get deeper insight on the quality of a Web application, e.g.:

– depth and breadth of the navigation structure
– distance between two related pages
– load time of pages
– latency of request handling
– size of transferred documents

Process Quality

First of all,

– a productive **organizational structure** and
– an adequate software **development process** must be defined.

Capability Maturity Model Integration (**CMMI®**) supports the assessment of software development processes.

CMMI defines five maturity levels (of organizations) by the techniques (**key process areas, KPAs**) applied:

- 1: **initial;** no defined process, unpredictable costs, duration and quality
- 2: **managed** (repeatable); fundamental process exists, new projects build on experience *(requirements management, software project management, software quality assurance, software configuration management, …)*
- 3: **defined;** a standard process established, control of costs and duration *(organizational process definition, training program, integrated software management, software product engineering, intergroup coordination, peer reviews)*
- 4: **quantitatively managed;** quantitative goals for product and process are defined and controlled, predictable costs, duration and quality *(quantitative process management, software quality management)*
- 5: **optimized;** finding of risks and weaknesses is stressed, the process is continuously enhanced *(defect prevention, technology change management, process change management)*

Quality of a system can be assured

Constructively

- using good **methods** and **tools** and employing good well-educated people
- by **automatizing** the activities using model/program transformations and generators

Analytically using

- **measurement**
- **testing**
- **inspection**
- **verification**

The **granularity** of tests differs (TEST LEVELS):

- **unit test**: test each of the smallest testable units (classes, web pages) independently
- **integration test**: evaluate the interaction between separately tested units client side, server side, communication)
- **system test**: test the complete integrated system
- **beta test**: let friendly users work with early versions (informal tests without test plans and test cases)
- **acceptance test**: evaluate the system under the auspices of the client with real conditions and real data

A **test** consists of the execution of a **sequence of test cases** (called: **test suite**) for a specific **object under test**. Test cases und suites are **versioned.**

A **test case** consists of

- an **identifier,**
- a set of **inputs** and **execution conditions**,
- as well a description expected **results** (usually a predicate).

Testing should be a planned and organized **process**:

- **Planning**: define the quality goals, the general testing strategy, the test plans for all test levels, the metrics and measuring methods, and the test environment
- **Preparing**: select the testing techniques and tools, specify the test cases
- **Performing**: prepare the test infrastructure, run the test cases, document and evaluate the results

- **Reporting**: summarize the test results, produce the test reports

Test cases are usually derived by

- by **Black-Box** approaches i.e. the cases are generated from the requirements and specifications and/or
- by **White-Box** (also: Glass-Box) approaches i.e. the cases are derived from the implementation

In both cases a large **coverage factor** has to be achieved.

Web Security

Security is a multi-faceted matter. It subsumes, e.g.,:

- confidentiality, exchanged data cannot be read by a third party
- integrity, nobody is able to modify the exchanged information
- authentication, the identity of a requesting person or a program is verified
- authorization, requesters are granted the right privileges

The goal of **confidentiality** is to prevent reading of exchanged documents by a third party.

HTTPS combines **HTTP** with **SSL/TLS** to encrypt communication data.

Integrity is the protection of the exchanged artifacts against changes by a third party.

To prevent modification through third parties, **digital signatures** can be used.

Checksums are one approach to detect forgery

Authentication denotes the verification of the identity of a person or program.

A **digital certificate** binds the **public key** to the identity of the **private-key** holder.

Authorization aims at identifying the privileges that authenticated users are granted.

Non-repudiation refers to the property that client and server appeal against their own statements, e.g., customers must not be able to deny orders they placed.

Privacy demands the reliable handling of data.

P3P policies describe

- **which information the server stores,**

e.g., which kind of information (identifying or not) and which information in particular (IP address, email address, name,

- **use of the collected information,**

e.g., how it is used (for navigation, tracking, personalization, etc.) and who will receive it (current company, third party,

- **permanence and visibility,**

e.g., how long information is stored and whether and how it can be accessed (read-only, optin, optout).

Cryptography

Many aspects of Web security rely on **cryptographic** methods. There is a long history of cryptographic algorithms.

The main groups are

- **symmetric** algorithms and
- **asymmetric** algorithms.

Encryption denotes the use of mathematical transformations of plain text into cipher text.

Decryption denotes the inverse process.

Symmetric Cryptography Receiver and sender use the same key S. Keys are a shared secret.

DES (Data Encryption standard) is a method for symmetric encryption.

Asymmetric Cryptography

Sharing a key implies the need to communicate the key between sender and receiver, which is a major risk. In **asymmetric cryptographic** methods sender and receiver **use different keys** (**private/public keys).**

The private key D is not revealed, the public key E is potentially available to everyone.

The main technique is the **RSA method**

Cross-site scripting (XSS) exploits the fact that parameters are sometimes not checked when they are sent to dynamically generated pages. This can be misused to infect infected code into the page.

Cross-Site Request Forgery (CSRF)

- ● Brings the browser of a logged-in user to send a manipulated HTTP request to a vulnerable application
- ● Session cookies and other authentication information are automatically sent by the browser
- ● Allows the attacker to perform actions within the affected applications in the name and context of the attacked user

SQL Injection: exploit errors in the implementation of the web application to execute SQL code on server.