Software Architecture

Unit 5 Notes

Documenting the Architecture

What to document

There are many different views, or aspects, of a software architecture that you could document. For any medium- to large-scale software development project, it is recommended you document, at a minimum, the following set of architectural artifacts:

System context

The system context documents how the entire system, represented as a black box, interacts with external entities (systems and end users). It also defines the information and control flows between the system and the external entities.

It is used to clarify, confirm, and document the environment in which the system operates. The nature of the external systems, their interfaces, and the information and control flows help in downstream specification of technical artifacts in the architecture.

Architecture overview

The architecture overview illustrates the main conceptual elements and relationships in an architecture through simple schematic representations. You can produce architecture overview diagrams that include an enterprise view and an IT system view. The overview helps to represent the business and IT capabilities required by the organization.

An architecture overview also provides high-level schematics that are further elaborated and documented in the functional and operational architectures. And it depicts the strategic direction the enterprise is taking as it pertains to IT systems.

Functional architecture

Also known as the component architecture, or model, the functional architecture artifact is used to document how the architecture is decomposed into IT subsystems that provide a logical grouping of the software components.

It describes the structure of an IT system in terms of its software components with their responsibilities, interfaces, static relationships, and the way they collaborate to deliver the required functions from the component. This artifact is developed iteratively through various stages of elaboration.

Operational architecture

The operational architecture artifact represents a network of computer systems that support some of the performance, scalability, and fault tolerance (among others) requirements of the solution. It also runs the middleware, systems software, and application software components.

This artifact is developed iteratively through various stages of elaboration.

Architecture decisions

The architecture decisions artifact provides a single place where all the architecturally relevant decisions are documented. Decisions are typically about but not limited to:

- The structure of systems.
- Identification of middleware components to support integration requirements.
- Allocation of functions to each architectural component (architectural building block).
- Allocation of architectural building blocks to the various layers in the architecture.
- Adherence to standards.
- Choice of technology to implement a particular architectural building block or functional component.

Any decision considered architecturally relevant to satisfy the business and engineering goals is documented. Documentation usually involves:

- Identification of the problem.
- Evaluation, including pros and cons, of various solutions.
- Selected solution, including adequate justification and other relevant details that would help downstream design and implementation.

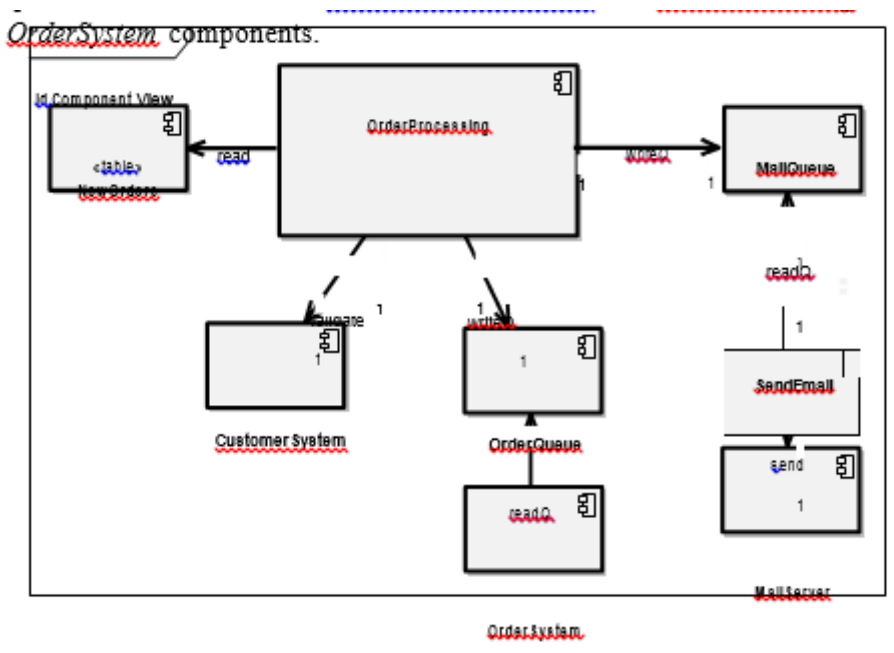
The UML 2.0 modeling notations cover both structural and behavioral aspects of software systems. The structure diagrams define the static archi- tecture of a model, and specifically are:

- **Class diagrams:** Show the classes in the system and their relationships.
- **Component diagrams:** Describe the relationship between components with well-defined interfaces. Components typically comprise multiple classes.
- **Package diagrams:** Divide the model into groups of elements and de- scribe the dependencies between them at a high level.
- **Deployment diagrams:** Show how components and other software arti- facts like processes are distributed to physical hardware.
- **Object diagrams:** Depict how objects are related and used at run-time. These are often called instance diagrams.
- **Composite Structure diagrams:** Show the internal structure of classes or components in terms of their composed objects and their relation- ships.

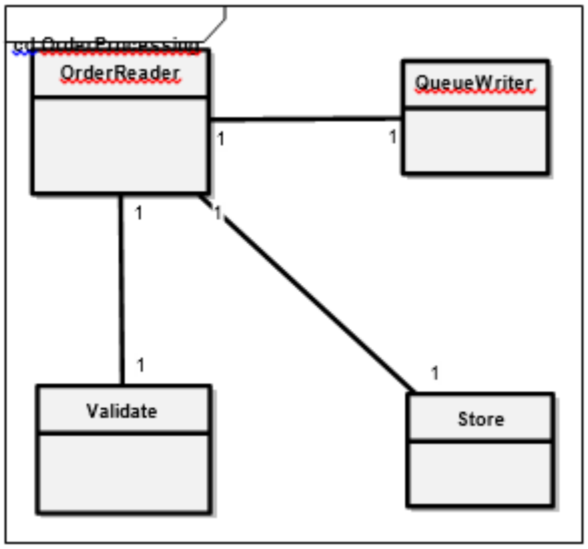
Behavior diagrams show the interactions and state changes that occur as elements in the model execute:

- **Activity diagrams:** Similar to flow charts, and used for defining pro- gram logic and business processes.
- **Communication diagrams:** Called collaboration diagrams in UML 1.x, they depict the sequence of calls between objects at run-time.
- **Sequence diagrams:** Often called swim-lane diagrams after their verti- cal timelines, they show the sequence of messages exchanged between objects.
- **State Machine diagrams:** Describe the internals of an object, showing its states and events, and conditions that cause state transitions.
- **Interaction Overview diagrams:** These are similar to activity dia- grams, but can include other UML interaction diagrams as well as ac- tivities. They are intended to show control flow across a number of sim- pler scenarios.
- **Timing diagrams:** These essentially combine sequence and state dia- grams to describe an object's various states over time and the messages that alter the object's state.
- **Use Case diagrams:** These capture interactions between the system and its environment, including users and other systems.

UML component diagram is used to represent an equivalent structural view of the order processing system architecture.

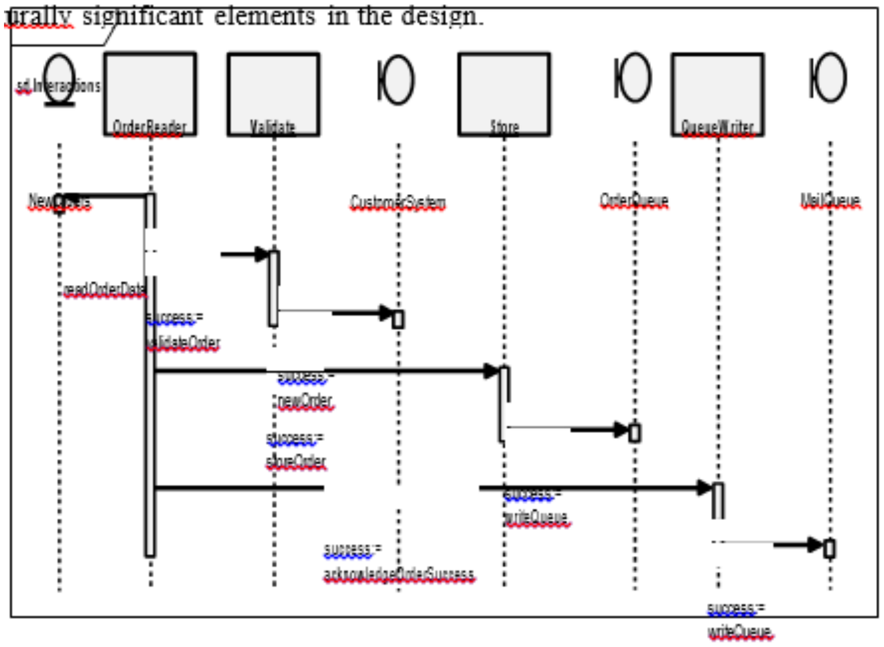


Only two of the components in the architecture require substantial new code to be created. The internal structure of the most complex of these, *OrderProcessing*, is shown in the class diagram. It includes a class essentially to encapsulate each interaction with an existing system. No doubt other classes will be introduced into the design as it is imple- mented, for example one to represent a new order, but these are not shown in the class diagram so that they do not clutter it with unnecessary detail. These are design details not necessary in an architecture description.

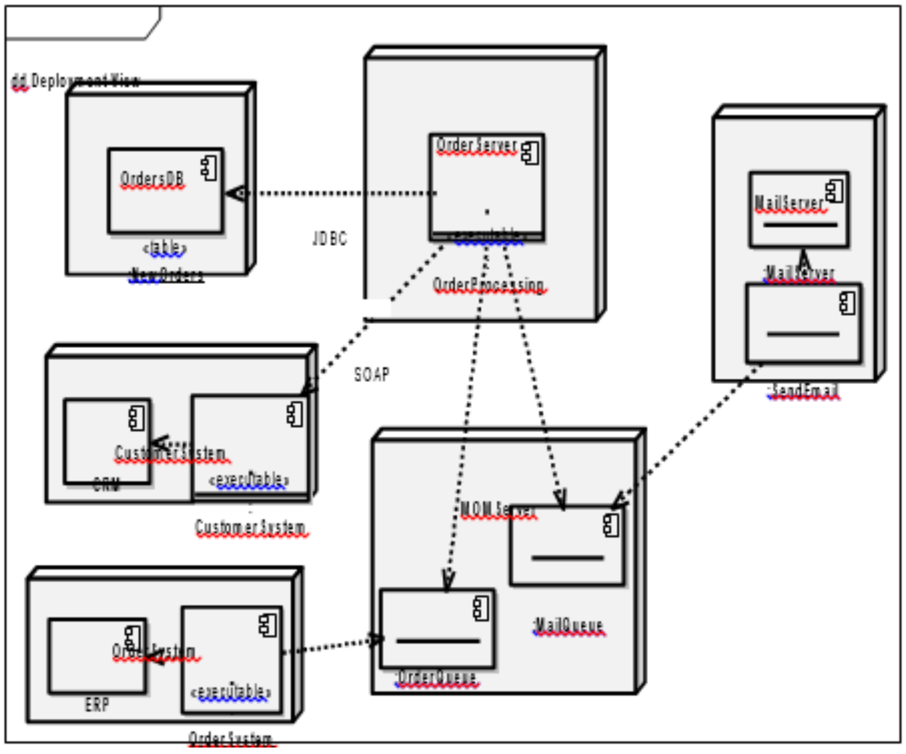


sequence diagram omits the behavior when a new order is invalid, and what happens once the messages have been placed on the *OrderQueue* and *MailQueue*. Sequence diagrams are probably the most useful technique in the UML for modeling the behavior of the components in an architecture. One of their strengths actually lies in their inherent weakness in describing com- plex processing and logic.

urally significant elements in the design.



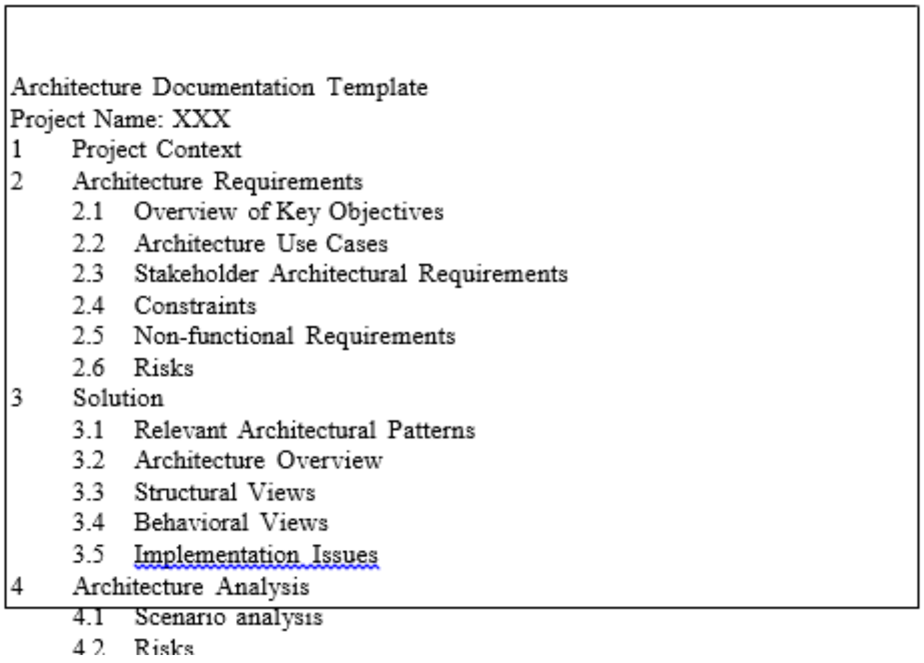
UML deployment diagram allocates components to servers and shows the dependencies between the components. It's often useful to label the dependencies with a name that indicates the protocol that is used to communicate between the components. For example, the *OrderProcessing* executable component requires JDBC[28] to access the *NewOrders* table in the *OrdersDB* database.



Architecture Documentation Template

Templates also help with the training of new staff as they tell developers what issues the organization requires them to consider and think about in the production of their system.

Documentation template can be used for capturing an architecture design. To deploy this template in an organization, it should be accompanied by explanatory text and illustra- tions of what information is expected in each section.


Architecture Documentation Template
Project Name: XXX
1 Project Context
2 Architecture Requirements
  2.1 Overview of Key Objectives
  2.2 Architecture Use Cases
  2.3 Stakeholder Architectural Requirements
  2.4 Constraints
  2.5 Non-functional Requirements
  2.6 Risks
3 Solution
  3.1 Relevant Architectural Patterns
  3.2 Architecture Overview
  3.3 Structural Views
  3.4 Behavioral Views
  3.5 Implementation Issues
4 Architecture Analysis
  4.1 Scenario analysis
  4.2 Risks

Architecture Description Languages

An architecture description language (ADL) is any form of expression for use in architecture descriptions. It provides one or more model kinds to frame concerns of its stakeholders

Common concepts of architecture

The ADL community generally agrees that Software Architecture is a set of components and the connections among them. But there are different kind of architectures like:

Object Connection Architecture

- Configuration consists of the interfaces and connections of an object-oriented system
- Interfaces specify the features that must be provided by modules conforming to an interface
- Connections represented by interfaces together with call graph
- Conformance usually enforced by the programming language
    - Decomposition — associating interfaces with unique modules
    - Interface conformance — static checking of syntactic rules
    - Communication integrity — visibility between modules

Interface Connection Architecture

- Expands the role of interfaces and connections
    - Interfaces specify both "required" and "provided" features
    - Connections are defined between "required" features and "provided" features

- Consists of interfaces, connections and constraints
    - Constraints restrict behavior of interfaces and connections in an architecture
    - Constraints in an architecture map to requirements for a system

Most ADLs implement an interface connection architecture.

Characteristics

The language must:

- Be suitable for communicating an architecture to all interested parties
- Support the tasks of architecture creation, refinement and validation
- Provide a basis for further implementation, so it must be able to add information to the ADL specification to enable the final system specification to be derived from the ADL
- Provide the ability to represent most of the common architectural styles
- Support analytical capabilities or provide quick generating prototype implementations

ADLs have in common:

- Graphical syntax with often a textual form and a formally defined syntax and semantics
- Features for modeling distributed systems
- Little support for capturing design information, except through general purpose annotation mechanisms
- Ability to represent hierarchical levels of detail including the creation of substructures by instantiating templates

ADLs differ in their ability to:

- Handle real-time constructs, such as deadlines and task priorities, at the architectural level
- Support the specification of different architectural styles. Few handle object oriented class inheritance or dynamic architectures
- Support the analysis of the architecture
- Handle different instantiations of the same architecture, in relation to product line architectures

Merits

ADLs represent a formal way of representing architecture

ADLs are intended to be both human and machine readable

ADLs support describing a system at a higher level than previously possible

ADLs permit analysis of architectures – completeness, consistency, ambiguity, and performance

ADLs can support automatic generation of software systems

Demerits

There is not universal agreement on what ADLs should represent, particularly as regards the behavior of the architecture

Representations currently in use are relatively difficult to parse and are not supported by commercial tools

Most ADL work today has been undertaken with academic rather than commercial goals in mind

Most ADLs tend to be very vertically optimized toward a particular kind of analysis

ACME

ACME was developed jointly by Monroe, Garlan (CMU) and Wile (USC)

ACME is a general purpose ADL originally designed to be a lowest common denominator interchange language

ACME as a language is extremely simple (befitting its origin as an interchange language)

ACME has no native behavioral specification facility so only syntactic linguistic analysis is possible, there are currently efforts under consideration to define a behavioral semantics for ACME, possibly along the Wright/CSP line

ACME has no native generation capability

ACME has seen some native tool development, and there are indications of more, as well as use of other language tools via interchange

The Acme language and toolkit provide three fundamental capabilities:

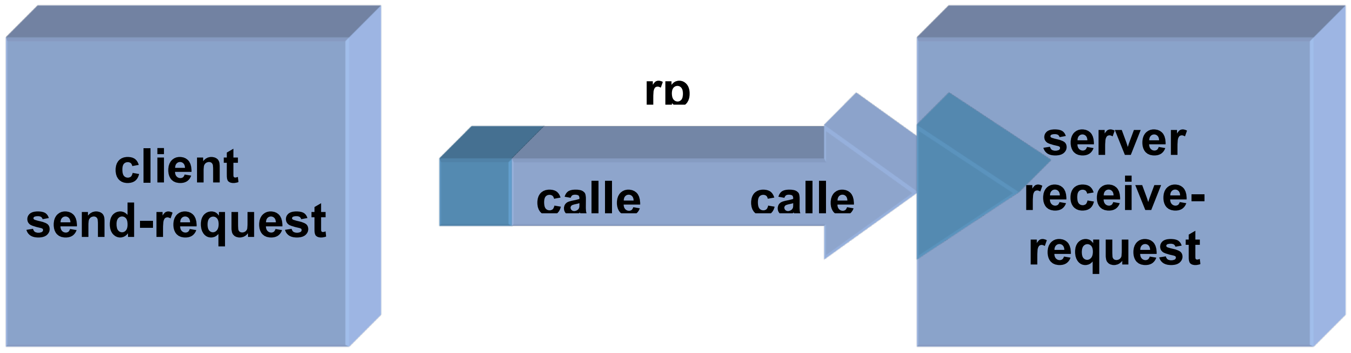
- **Architectural interchange.** By providing a generic interchange format for architectural designs, Acme allows architectural tool developers to readily integrate their tools with other complementary tools. Likewise, architects using Acme-compliant tools have a broader array of analysis and design tools available at their disposal than architects locked into a single ADL.
- **Extensible foundation for new architecture design and analysis tools.** Many, if not most, architectural design and analysis tools require a representation for describing, storing, and manipulating architectural designs. Unfortunately, developing good architectural representations is difficult, time consuming, and costly. Acme can mitigate the cost and difficulty of building architectural tools by providing a language and toolkit to use as a foundation for building tools. Acme provides a solid, extensible foundation and infrastructure that allows tool builders to avoid needlessly rebuilding standard tooling infrastructure. Further, Acme's origin as a generic interchange language allows tools developed using Acme as their native architectural representation to be compatible with a

broad variety of existing architecture description languages and toolsets with little or no additional developer effort.

- **Architecture Description.** Acme has emerged as a useful architecture description language in its own right. It provides a straightforward set of language constructs for describing architectural structure, architectural types and styles, and annotated properties of the architectural elements. Although not appropriate for all applications, the Acme architecture description language provides a good introduction to architectural modelling, and an easy way to describe relatively simple software architectures.

An ADL Example (in ACME)

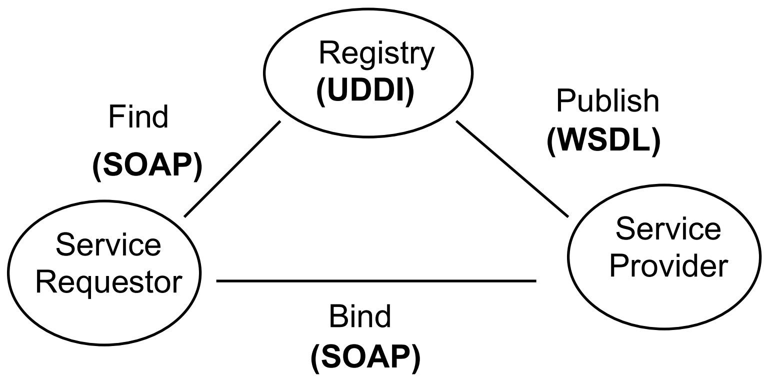
```
System simple_cs = {

Component client = {Port send-request}

Component server = {Port receive-request}

Connector rpc = {Roles {caller, callee}}

Attachments : {client.send-request to rpc.caller;

   server.receive-request to rpc.callee}

}
```



SOA and Web Services

SOA

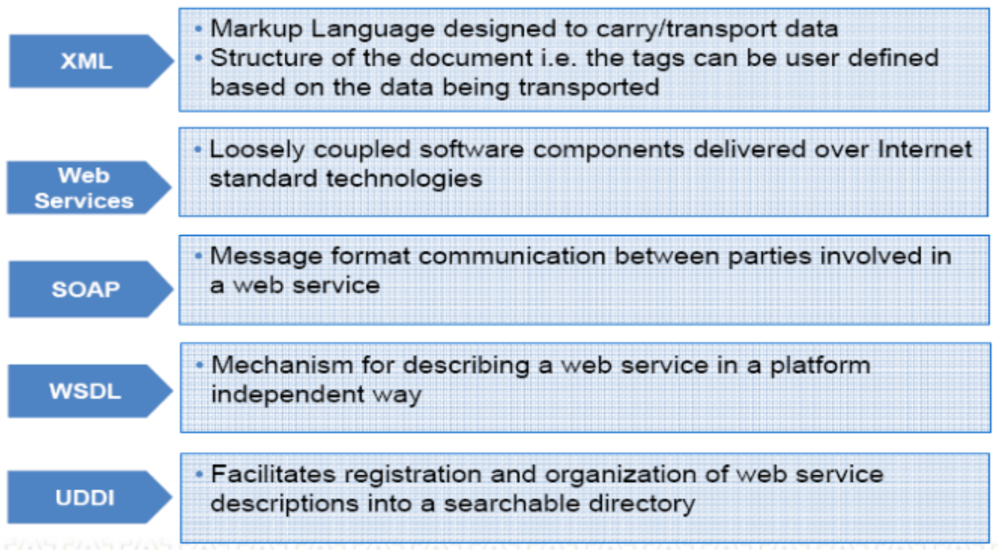
- **Contemporary Service-Oriented Architectures (SOA) represents an architecture that promotes service-orientation through the use of Web services.**
- **All functions, or services, are defined using a description language and have invokable interface that are called to perform business processes.**



Key components of SOA

• Services (common denominator)
• Service Description
• Advertising and Discovery
• Specification of associated data model
•Service contracts

Key Standards of and Technology of SOA



XML	• Markup Language designed to carry/transport data • Structure of the document i.e. the tags can be user defined based on the data being transported
Web Services	• Loosely coupled software components delivered over Internet standard technologies
SOAP	• Message format communication between parties involved in a web service
WSDL	• Mechanism for describing a web service in a platform independent way
UDDI	• Facilitates registration and organization of web service descriptions into a searchable directory

Challenges of SOA

Security challenges
- loosely coupled environment
Performance
- XML brings robustness not speed
Optimization
Organizing the services
– registry & repository
Finding the right services and right interfaces
Transaction management is complex in interactions between logically separate systems.

Web Services

- A Web service is a software system designed to support interoperable machine-to-machine interaction over a network.
- It has an interface described in a machine-processable format (specifically WSDL).
- Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."

Key features of Web Services

- A modular, well-defined, encapsulated function
- Used for loosely coupled integration between applications or systems
- Based on XML, transported in two forms:
  - Synchronous (RPC)
  - Asynchronous (messaging)
  - Both over Simple Object Access Protocol (SOAP)
- Specified in *Web Services Description Language (WSDL)*
- Sometimes advertised and discovered in a service registry – *Universal Description, Discovery and Integration (UDDI)*
- Over Intranet and Internet

Web Services Specification

There are three basic specifications, forming the basis for state-of-the-art Web-Service technology
Simple Object Access Protocol (SOAP) for a common message processing & coding Web Service Description Language (WSDL) as a metadata standard for Web-Services Universal Description, Discovery and Integration (UDDI) specifying a catalog / discovery service

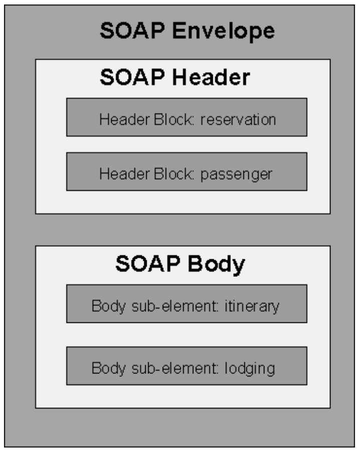
Use of SOA and Web Services

- Facilitates:
  - Marketing efforts
  - E-Commerce
  - Personalization
  - Direct services to end users
- Strategies:
  - Focus now on partnerships
  - Integration
  - Direct communication
  - Automating processes across organizational boundaries

SOAP

i. SOAP stands for Simple Object Access Protocol
ii. SOAP is a communication protocol
iii. SOAP is for communication between applications
iv. SOAP is a format for sending messages
v. SOAP communicates via Internet
vi. SOAP is platform independent
vii. SOAP is language independent
viii. SOAP is based on XML
ix. SOAP is simple and extensible
x. SOAP allows you to get around firewalls
xi. SOAP is a W3C recommendation

Message structure:


SOAP Envelope
SOAP Header
Header Block: reservation
Header Block: passenger
SOAP Body
Body sub-element: itinerary
Body sub-element: lodging

A SOAP message is an ordinary XML document containing the following elements.

Envelope:(Mandatory): Defines the start and the end of the message.

Header:(Optional): Contains any optional attributes of the message used in processing the message, either at an intermediary point or at the ultimate end point.

Body:(Mandatory): Contains the XML data comprising the message being sent.

Fault:(Optional): An optional Fault element that provides information about errors that occurred while processing the message

All these elements are declared in the default namespace for the SOAP envelope
SOAP Header element can be explained as:

- Header elements are optional part of SOAP messages.
- Header elements can occur multiple times.
- Headers are intended to add new features and functionality
- The SOAP header contains header entries defined in a namespace.
- The header is encoded as the first immediate child element of the SOAP envelope.

SOAP Nodes

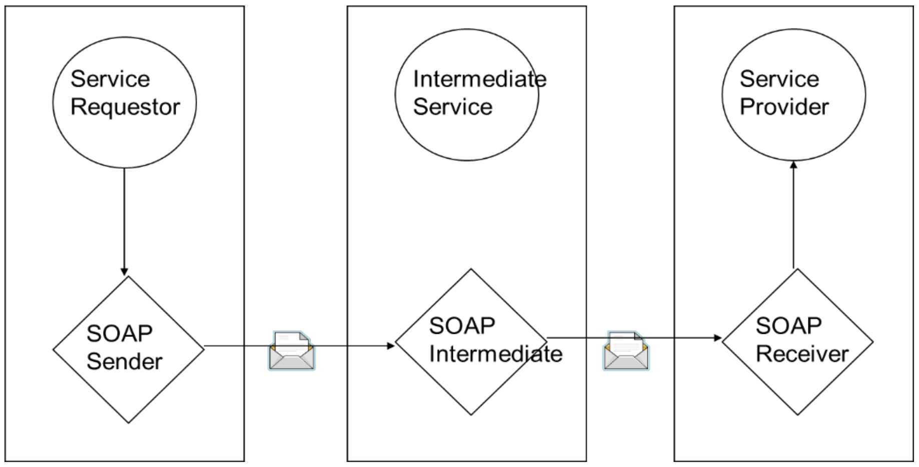
Nodes:

- Programs that services use to transmit and send SOAP messages are **SOAP nodes**
- SOAP nodes must conform to SOAP specification, as a result SOAP messages are vendor-neutral.

Node Types:

- **SOAP sender** – a SOAP node that transmits a message
- **SOAP receiver** – a SOAP node that receives a message
- **SOAP intermediary** – a SOAP node that receives and transmits a message with optional processing
- **Initial SOAP sender** – the first SOAP node to transmit the message
- **Ultimate SOAP receiver** – the last SOAP node to receive a message

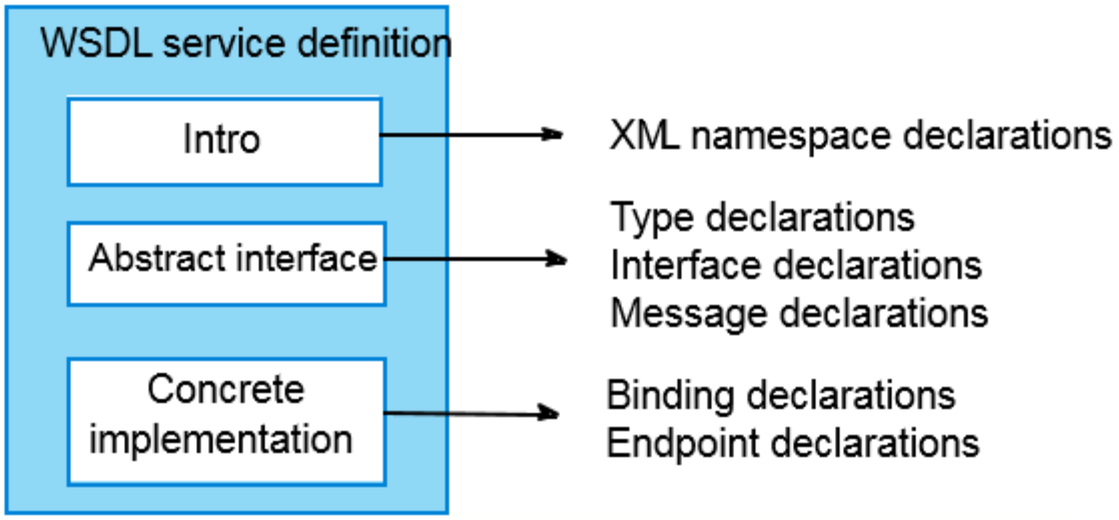
SOAP Intermediaries:



Web service description language

- The service interface is defined in a service description expressed in WSDL. The WSDL specification defines:
  - ❑ What operations the service supports and the format of the messages that are sent and received by the service.
  - ❑ How the service is accessed - that is, the binding maps the abstract interface onto a concrete set of protocols.
  - ❑ Where the service is located. This is usually expressed as a URI (Universal Resource Identifier).

Structure of a WSDL specification



UDDI

- Universal Description, Discovery, and Integration
- API for a Web based registry
- Implemented by an *Operator* Site
    - Replicate each others' information

UDDI Data Structures

- businessEntity:
    - Basic business information
    - Used by UDDI for "yellow" pages
- businessService:
    - Services provided by that business
    - Grouping of related businesses
- bindingTemplate:
    - What the service looks like (tModel element)
    - Where to access the service
- tModel
    - Technology model
    - Could contain just about anything
    - Has service details
        - Abstract industry specs
        - Service specs
- Designed to be reusable
- Can contain pointer to WSDL document

Cloud is a pool of virtualized computer resources networked, which can:

- ➢ Host a variety of workloads.
- ➢ Batch-style back-end jobs.

- Interactive user-facing applications.
- Workloads can be deployed and scaled out quickly through the rapid provisioning of virtual machines or physical machines.
- Support redundant, self recovering, highly scalable programming models that allow workloads to recover from many unavoidable hardware / software failures.
- Monitor resource use in real time to enable rebalancing of allocations when needed.

Five Key Cloud Attributes:

1. Shared / pooled resources
2. Broad network access
3. On-demand self-service
4. Scalable and elastic
5. Metered by use

Shared / Pooled Resources:

- Resources are drawn from a common pool
- Common resources build economies of scale
- Common infrastructure runs at high efficiency

Broad Network Access:

- Open standards and APIs
- Almost always IP, HTTP, and REST
- Available from anywhere with an internet connection

On-Demand Self-Service:

- Completely automated
- Users abstracted from the implementation
- Near real-time delivery (seconds or minutes)
- Services accessed through a self-serve

web interface

Scalable and Elastic:

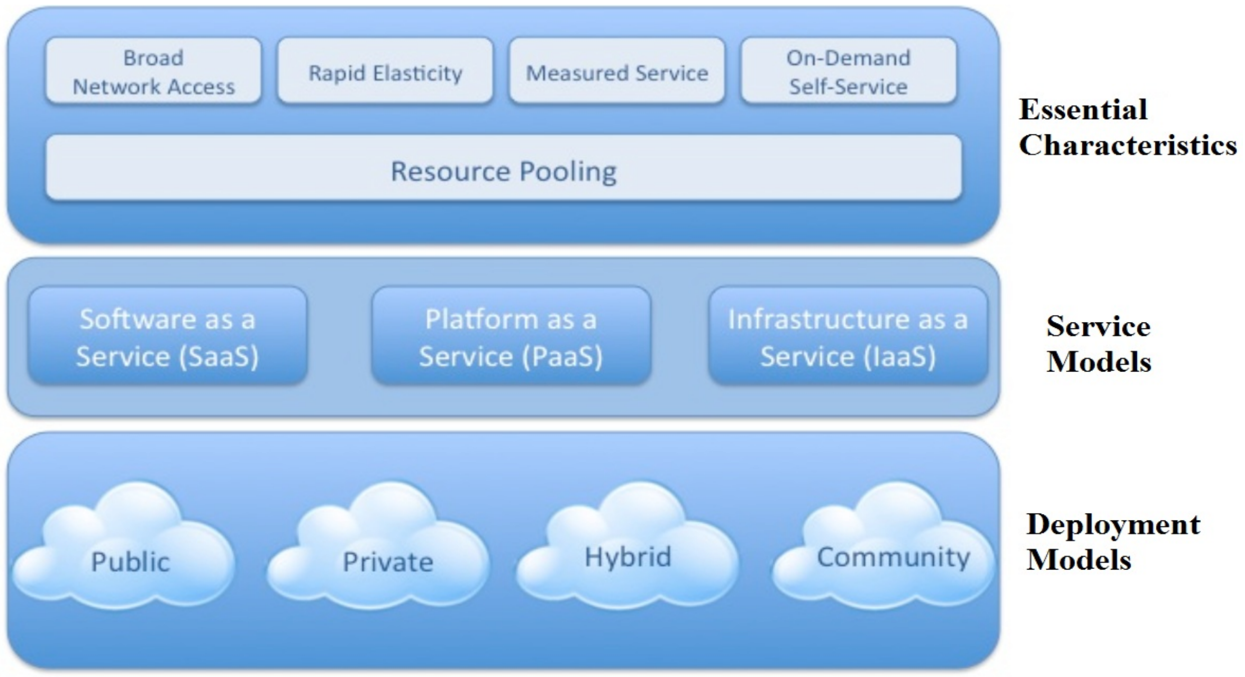
- Resources dynamically-allocated between users
- Additional resources dynamically-released when needed

- Fully automated

Metered by Use:

- Services are metered, like a utility
- Users pay only for services used
- Services can be cancelled at any time

Architecture Overview



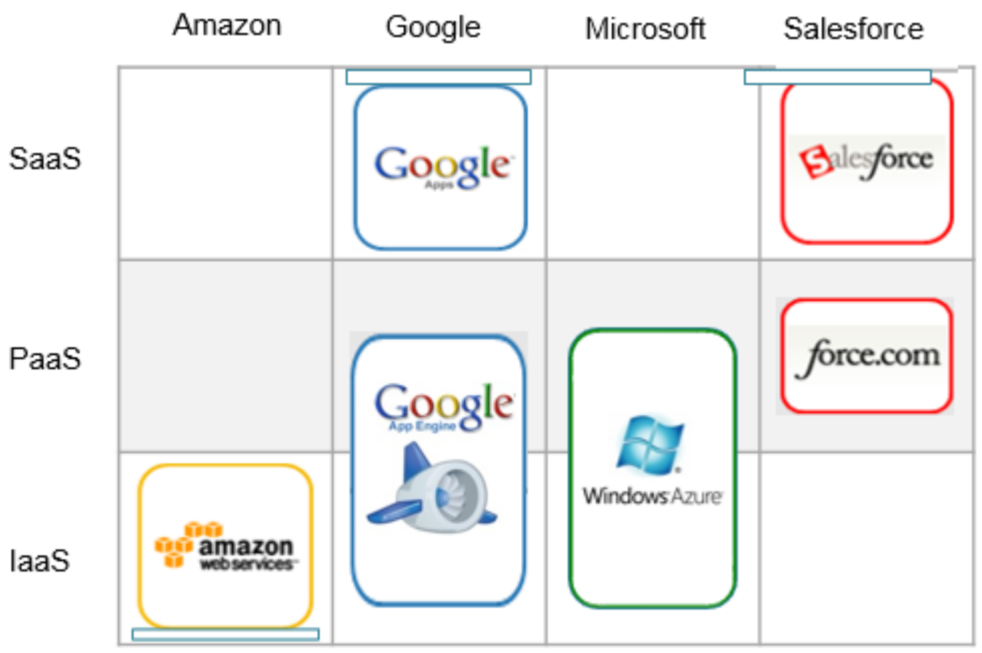
Architectural Layers of Cloud Computing

In the cloud computing stack, there are three basic layers that together create cloud environment. They are:

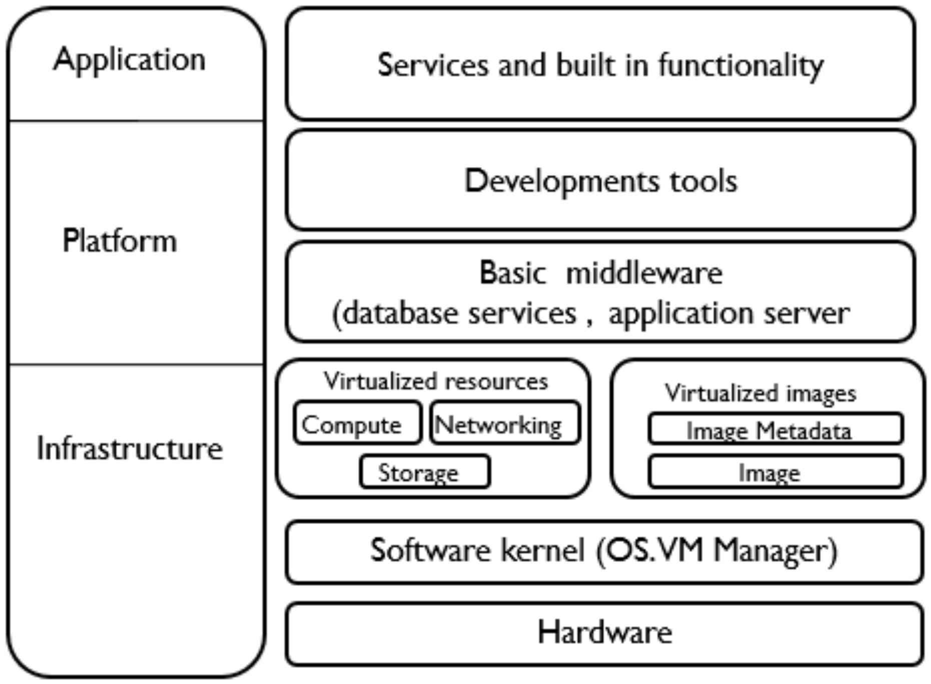
1. Infrastructure as a Service(IaaS)
2. Platform as a Service (PaaS)

Software as a Service (SaaS)

Service Delivery Model Examples



Framework of cloud computing

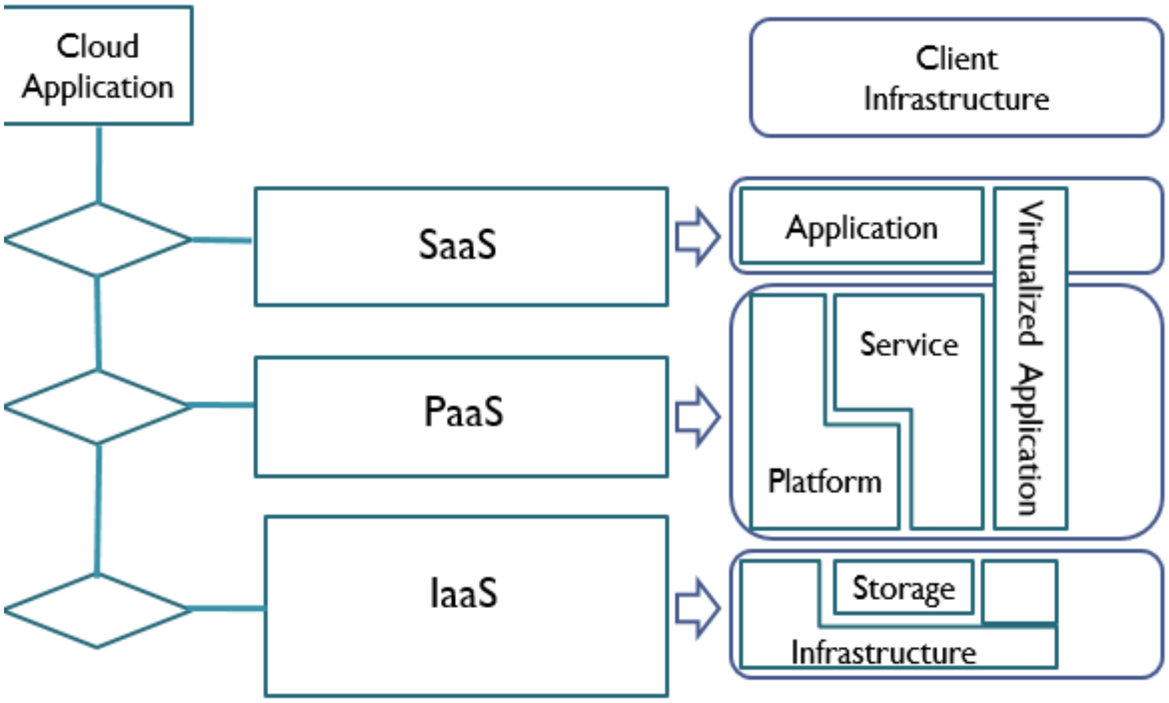


Virtual infrastructure management and Cloud Computing

- For building the cloud environment a variety of requirements must be met to provide a uniform and homogeneous view of the virtualized resources.

- ➢ Virtual Infrastructure Management is the key component to build the cloud environment which does the dynamic orchestration of virtual machines on a pool of physical resources.
- ➢ Virtual infrastructure management provide primitives to schedule and manage VMs across multiple physical hosts.
- ➢ Cloud management provide remote and secure interface for creating controlling and monitoring virtualized resources on IaaS.

View of Cloud Deployment



Software as a Service

- ❖ It is a Deployment/Delivery model
- • Hosted and managed by vendor
- • Delivered across the internet
- ❖ It is a Business Model : usage-based pricing(vs. perpetual license model of on – premise software).Examples:
- • Per user per month
- • Per transaction
- • Per GB of storage per month

Architectural

- Multi-tenancy
- Scalability
- Security
- Performance

Functional

- Provisioning
- Billing
- Metering
- Monitoring

MULTI-TENANCY

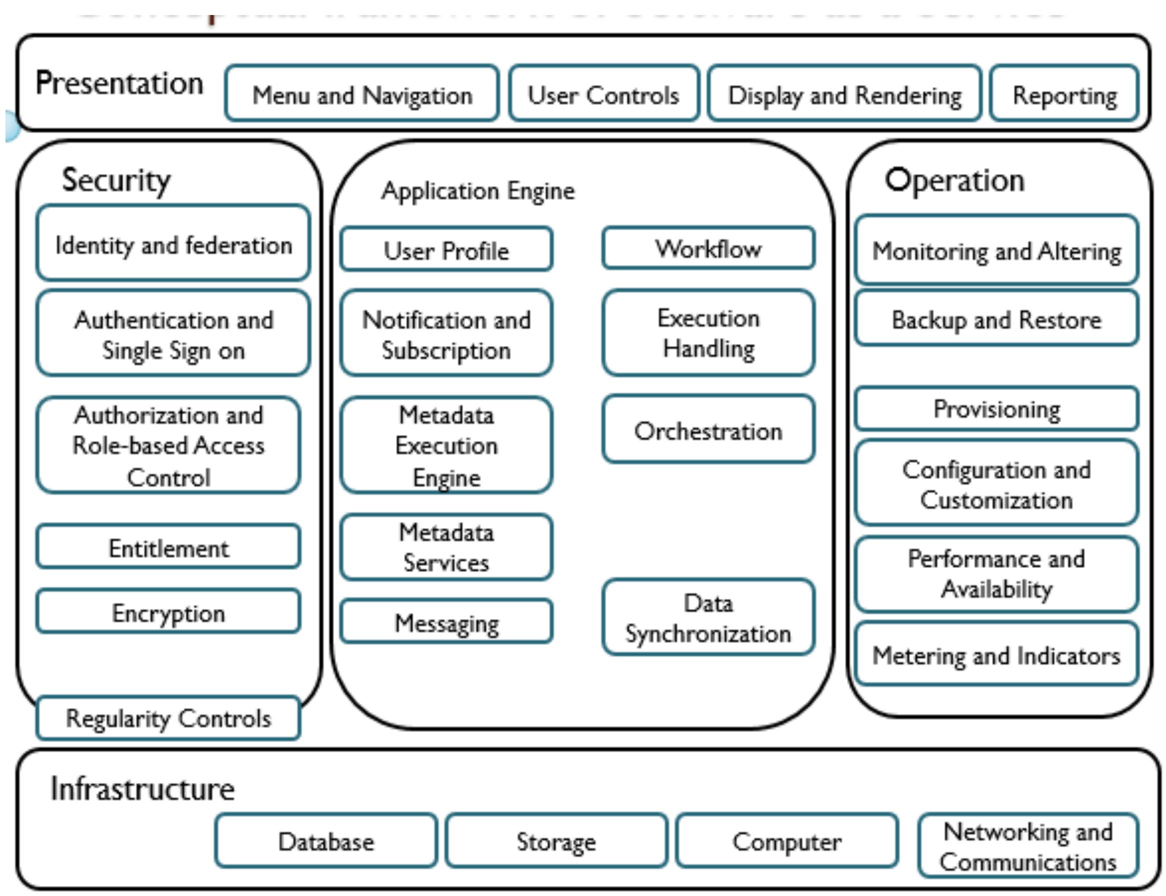
- Multi-tenancy is an architectural pattern
- A single instance of the software is run on the service provider's infrastructure
- Multiple tenants access the same instance.
- In contrast to the multi-user model, multi-tenancy requires customizing the single instance according to the multi-faceted requirements of many tenants.

A Multi-tenants application lets customers (tenants) share the same hardware resources, by offering them one shared application and database instance ,while allowing them to configure the application to fit there needs as if it runs on dedicated environment.

These definition focus on what we believe to be the key aspects of multi tenancy:

1. The ability of the application to share hardware resources.
2. The offering of a high degree of configurability of the software.
3. The architectural approach in which the tenants make use of a single application and database instance.

Conceptual framework of Software as a Service



SOA and Cloud Computing

In cloud environment we adopt the bundling of resources into layers of

Saas

Paas

Iaas

And furthur add a layer for business process management with the concept of service oriented architecture (SOA).

SOA is a base for furthur building of cloud environment for composite application with work flow concepts.

Adaptive Structures

Adaptive architecture is a system which changes its structure, behaviour or resources according to demand. The adaptation made is usually to non-functional characteristics rather than functional ones.

Something of a misnomer, because the thing that adapts is the working system, rather than the (more abstract) architecture which defines the adaptability that is required of that system.

Adaptive software architecture: Used by programmers in relation to a program. An adaptive algorithm "is an algorithm which changes its behavior based on the resources available. For example… in the C++ Standard Library, the stable partition [program] acquires as much memory as it can get (up to what it would need at most) and applies the algorithm using that available memory."

Adaptive infrastructure architecture: Used by infrastructure engineers in relation to the configuration of processors. The computing resources used by applications (the partition size, or the number of servers in a cluster, or the share of a processor, or the number of processes) are configured so that they shrink or grow with demand.

Adaptive business architecture: Could also be used (for example) in connection with a workflow system that assigns human resources to a task or service to match the demand for that task or service. Or an organisation structure that flexes in response to business changes.

Adaptation

- Change is endemic to software

  perceived and actual malleability of software induces stakeholders to initiate changes, e.g.:

  - Users want new features
  - Designer wants to improve performance
  - Application environment is changing

- Adaptation: modification of a software system to satisfy new requirements and changing circumstances

Sources and Motivations for Change

- Corrective Changes

  Bug fixes

- Modification to the functional requirements

  New features are needed

  Existing ones modified

  Perhaps some must be removed

- New or changed non-functional system properties

  Anticipation of future change requests

- Changed operating environment
- Observation and analysis

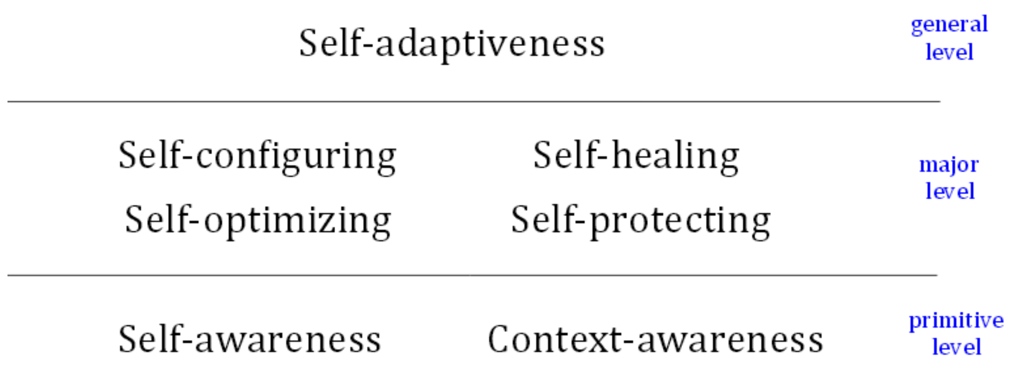
Self--adaptive software Software that evaluates its own performance and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do.

Aims to adjust various artifacts and attributes in response to changes in The self , that is, the whole body of the software, usually implemented in several layers

The context, that is, everything in the operating environment that affects the system's properties and its behavior

Its lifecycle cannot be stopped after its development and initial setup

Self--* properties



Open vs. closed systems

Closed systems
–Fixed set of elements
–Adaptation can only act on them to keep the system on track
•Open systems
–Elements can appear and disappear
–Adaptation must both "discover" existing elements and act on them to keep the system on track

Anticipated vs. un--anticipated

•Anticipated adaption (closed)
–Situations to be accommodated at run--time are known at design--time
•Un--anticipated adaption (open)
–Possibilities are recognized and computed at run--time
–Decisions are computed by using self--awareness and environmental context information

Externalized adaptation

•One or more models of the system are maintained at runtime

–They are used to identify and resolve problems
•Changes are described as operations on the model
•Changes to the model affect changes onto the underlying system

Different needs
•Topology
–Different interactions among the same elements
–New elements enter the system
•Behavior
–Same elements start behaving differently
–New elements are injected in the system
•Control
–MAPE elements must be added
–Reliability and robustness must be enforced