Software Architecture and Systems-of-Systems

Milena Guessi Margarido

Prof. Dr. Elisa Yumi Nakagawa

Prof. Dr. José Carlos Maldonado

• • •

SCC-5944 Software Engineering - 2016

Program

I. Software Architecture
   1. Overview
   2. Architectural Requirements
   3. Architectural Description
   4. Architectural Evaluation

Break

I. Systems-of-Systems
   1. Software Architecture
   2. Architectural Description

Software Architecture

Part I

Overview

- What is it?
- Who does it?
- Why is it important?
- What are the steps?
- What is the work product?
- How do I ensure that I’ve done it right?

What is it?

- Architectural design represents the structure of data and program components that are required to build a computer-based system
  - ○ Architectural style
  - ○ Struture and properties of components
  - ○ Interrelationships that occur among them

Blueprint from which software is constructed

Definition

Fundamental concepts or properties of a system embodied in its elements, relationships, and in the principles guiding its design and evolution over time

ISO/IEC/IEEE 42010

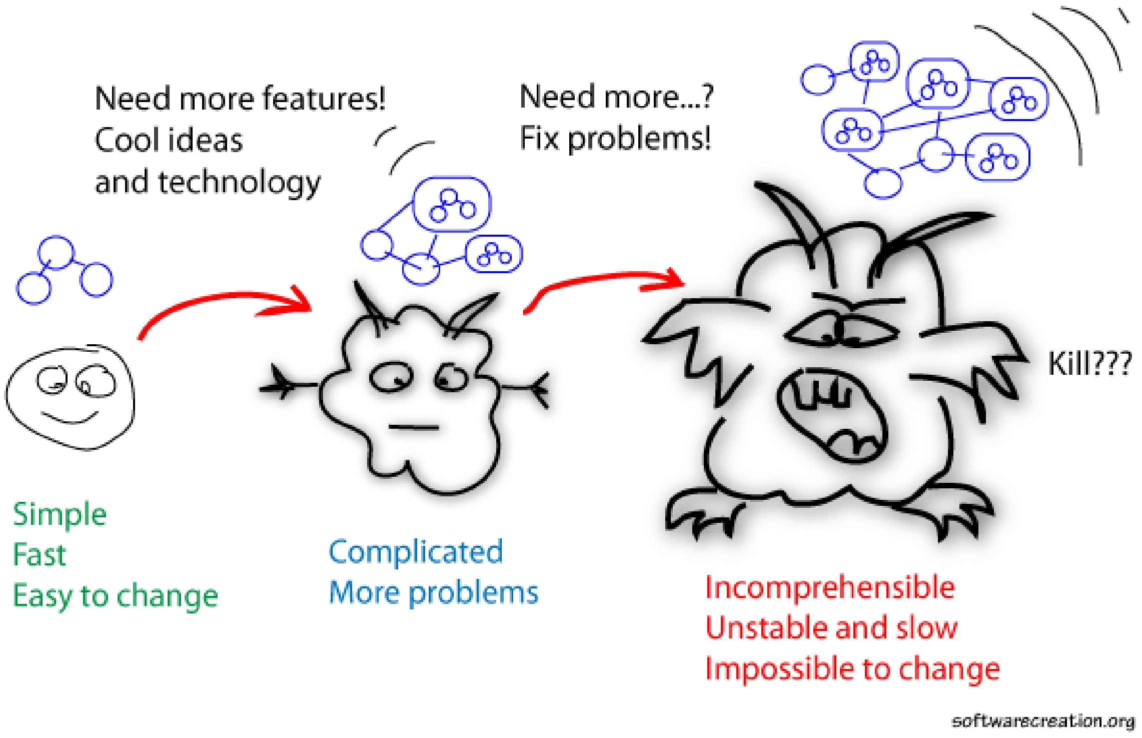
What is it?

- The architecture ***is not*** the operational software

  It is an abstraction that enables you to:

  i. Analyze the effectiveness of the design in meeting its stated requirements
  ii. Consider architectural alternatives at a stage when making design changes is still relatively easy
  iii. Reduce the risks associated with the construction of the software

Why is it important?



Need more features!
Cool ideas
and technology

Need more...?
Fix problems!

Kill???

Simple
Fast
Easy to change

Complicated
More problems

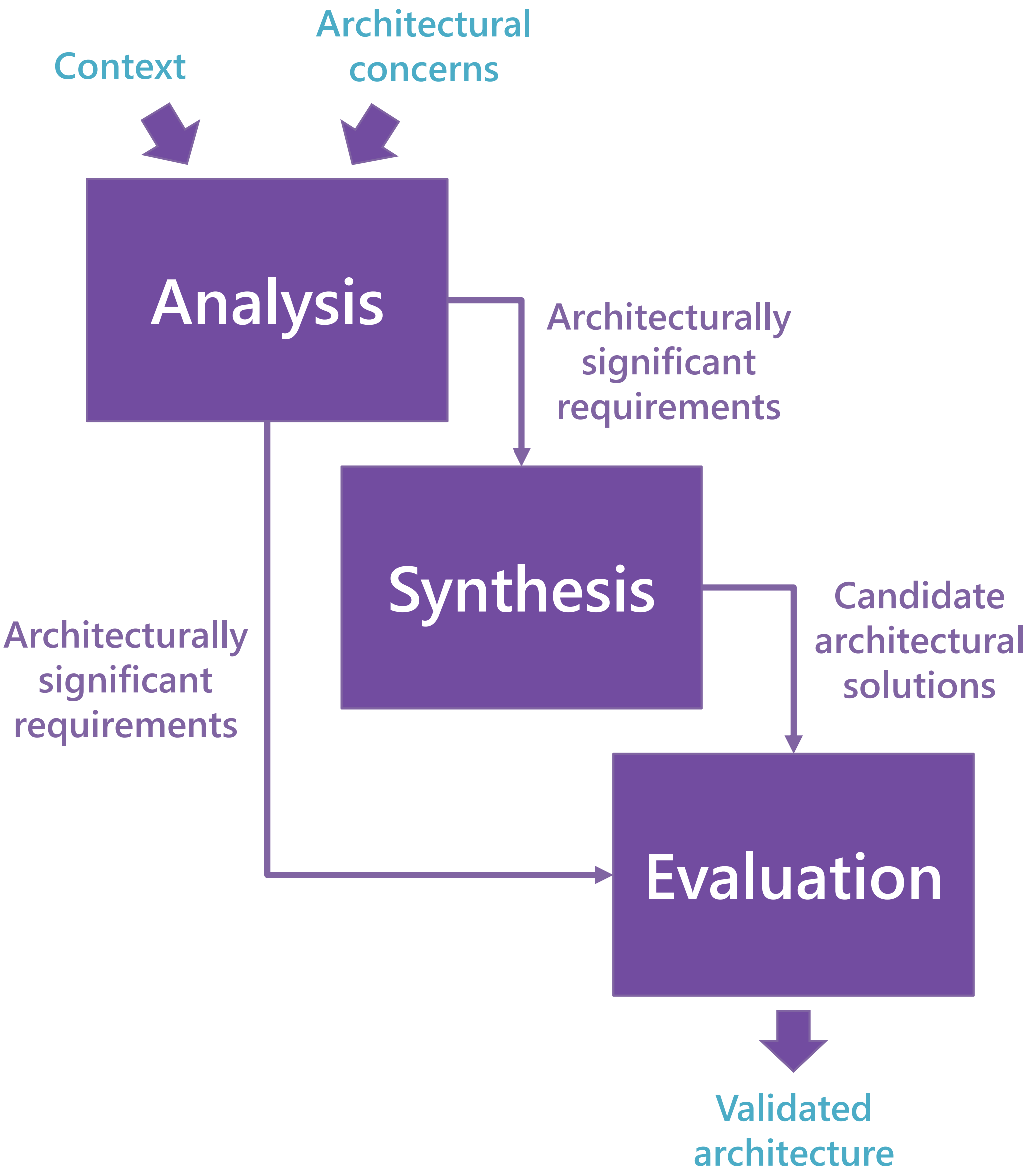
Incomprehensible
Unstable and slow
Impossible to change

softwarecreation.org

What are the steps?

1. Data design
2. One or more representations of the architectural structure
3. Selection of architectural styles or patterns that are best suited to customer requirements and quality attributes
4. Selection of an architectural alternative
5. Elaboration of the architecture using an architectural design method

What are the steps?


Context
Architectural concerns
Analysis
Architecturally significant requirements
Synthesis
Candidate architectural solutions
Architecturally significant requirements
Evaluation
Validated architecture

Source: Hofmeister, C. et al., 2007

What is the work product?

- **Architecture description** is created during the architectural synthesis
  - Encompasses the set of tangible artifacts expressing a software architecture (ISO/IEC/IEEE 42010)
- Communicates the architecture design to stakeholders
- *"Software architecture documentation speaks for the architect, today, tomorrow and 20 years from now."* (SEI)

How do I ensure that I’ve done it right?

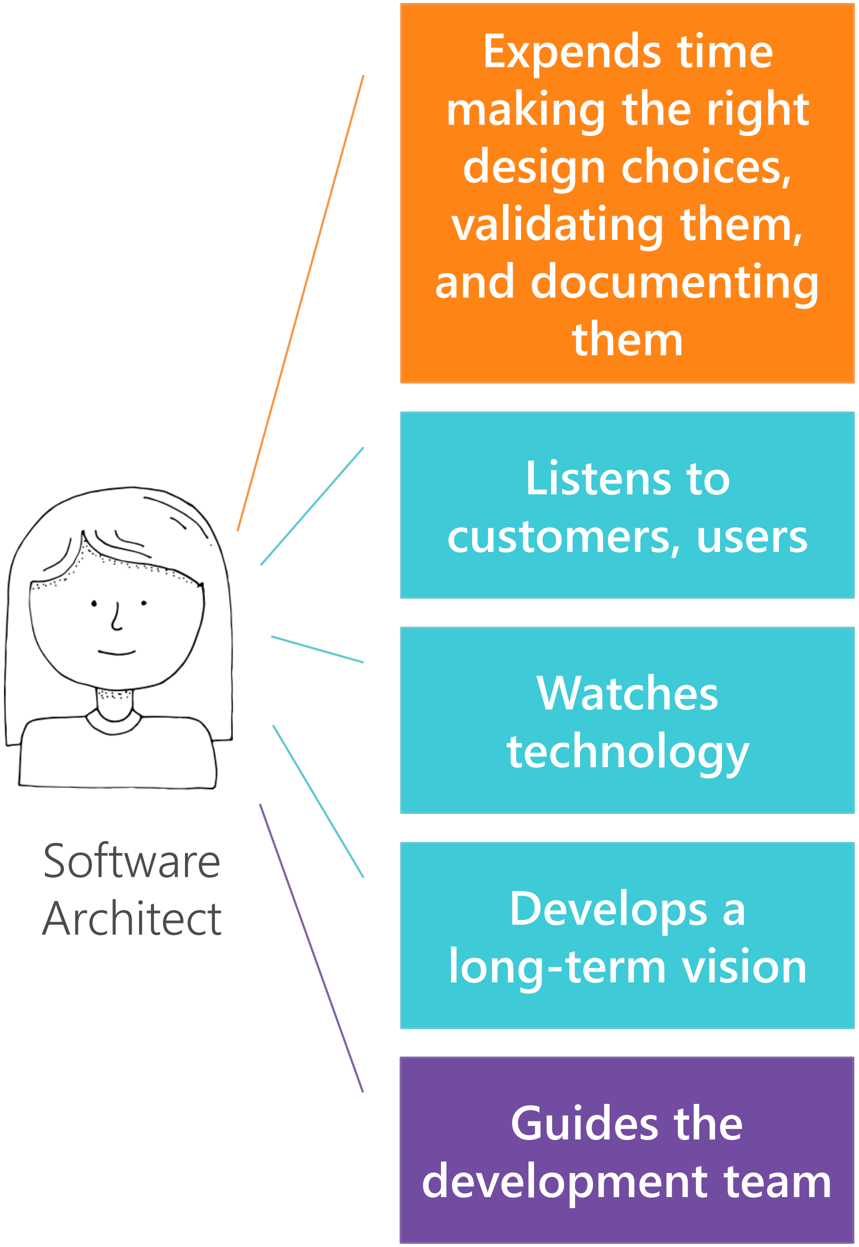
- We need to ensure that the architectural decisions taken are the right ones
  - Architecture reviews (or evaluations) are independent examinations of the software architecture to identify potential architectural problems
  - At each stage of the architecture design method, the architecture description is reviewed for
    - Clarity
    - Correctness
    - Completeness
    - Consistency

    with requirements and with one another

Who does it?

- Architects’ tasks and responsabilities could be manyfold (Garland and Anthony, 2003):
  - Technical Risk Analyst
    - Manage risk
    - Evaluate requirements change risk
  - Domain Analyst
    - Divide problems and create solutions that fit the organization needs
  - Deliverables Reviewer
  - Development Team Mentor
  - Developer
  - Team Lider

What do they do?



Software Architect

Expends time making the right design choices, validating them, and documenting them

Listens to customers, users

Watches technology

Develops a long-term vision

Guides the development team

Time

50%

Internal

25%

External: Inwards

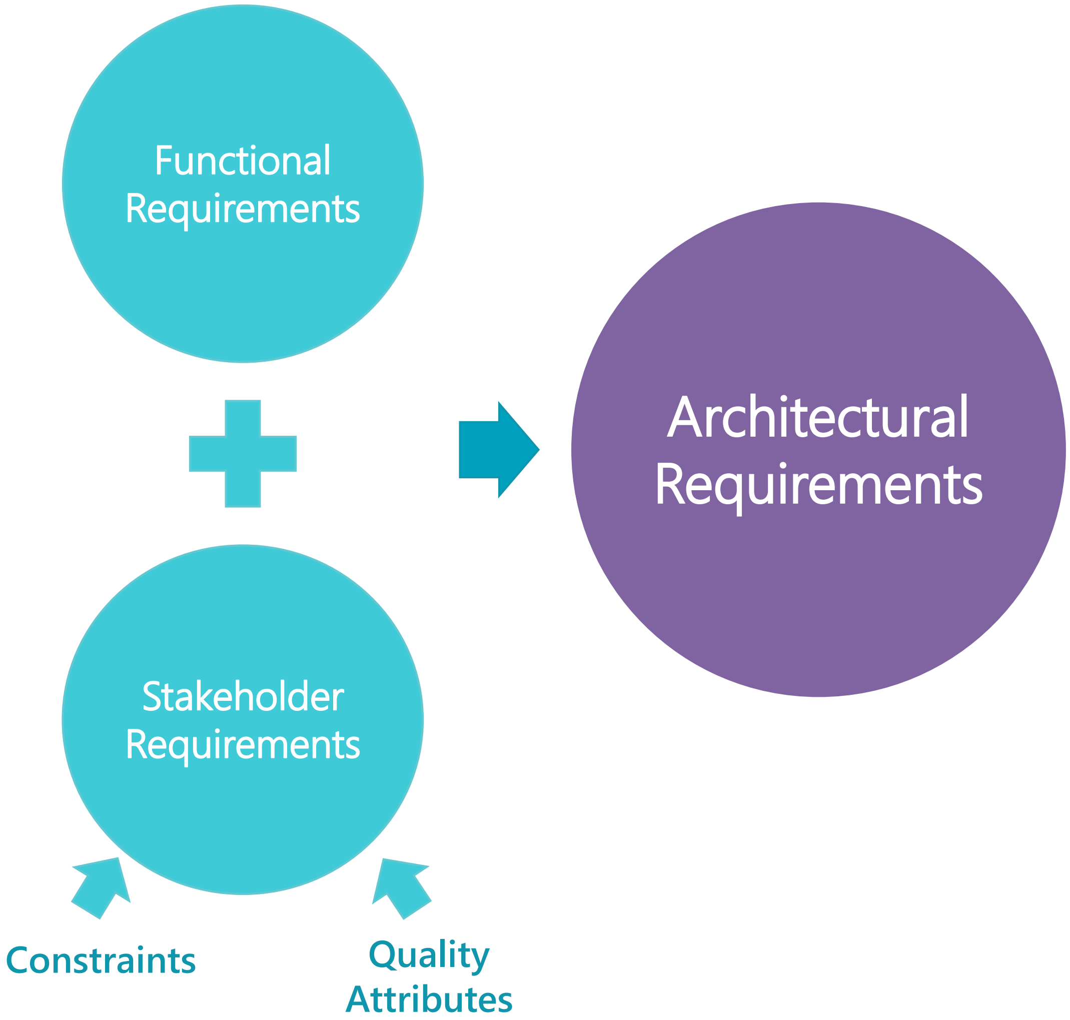
25%

External: Outwards

Source: Kruchten, P. 2008

Architectural Requirements

Architectural Analysis



Source: Gorton, I., 2006

Architectural Requirements Examples

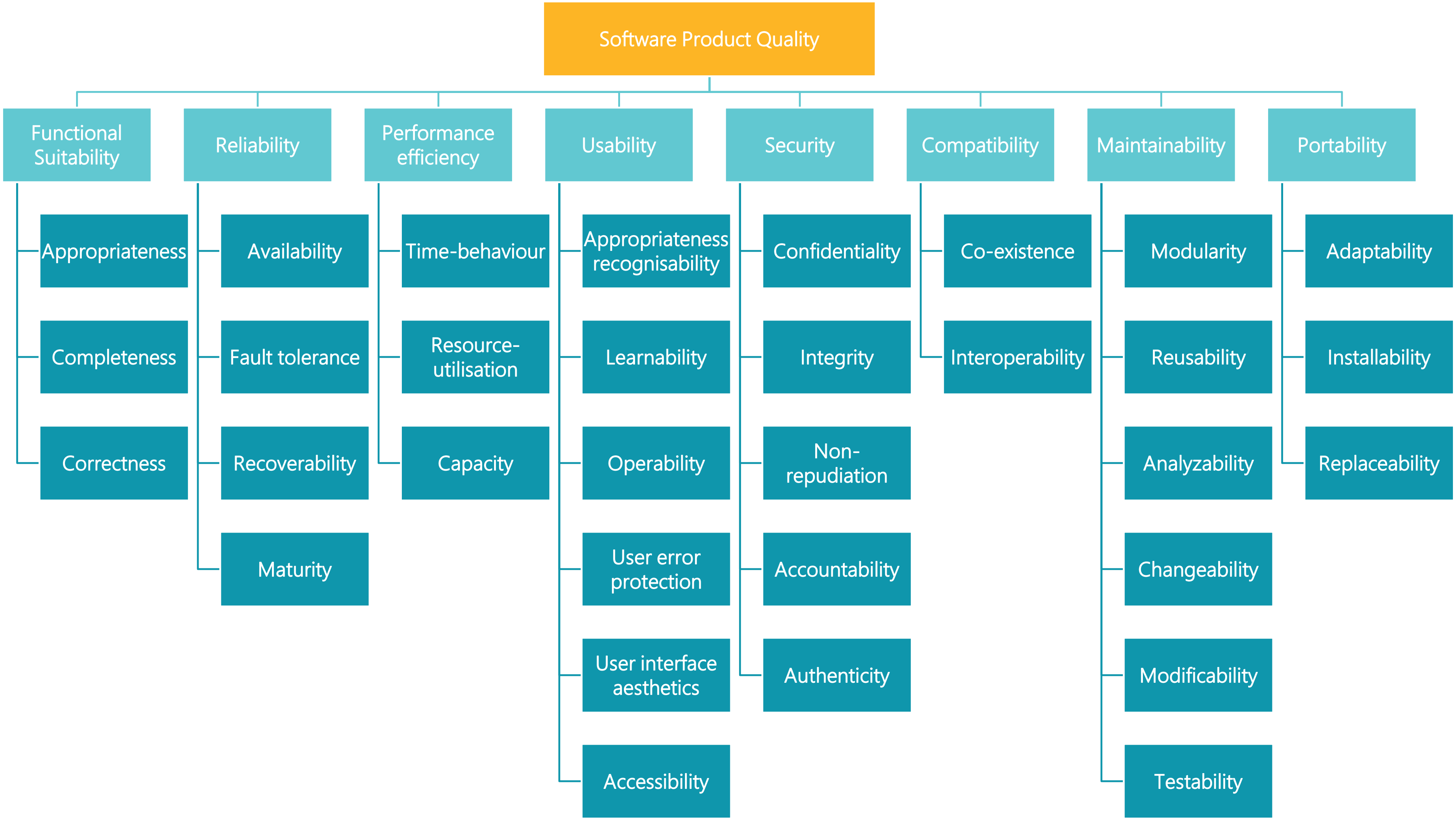
- Reliability of communications:
  - “Communications between components **must be guaranteed to succeed** with no message loss”
- Constraints:
  - “The system **must use** the existing IIS-based web server and use Active Server Page to process web requests”

Software Quality

ISO/IEC 25000

- **Software Product Quality**
  - Satisfaction level reached by a software product when it is used within specific conditions
- **Quality Attribute**
  - Software characteristic that specifies the level of a given attribute impacting software quality
  - Examples: usability, reliability, performance, etc.
- **Quality Model**
  - Set of characteristics, and their interrelationships, used as a benchmark for specifying quality requirements and measuring software quality

ISO/IEC 25010 Quality Model



- Software Product Quality
  - Functional Suitability
    - Appropriateness
    - Completeness
    - Correctness
  - Reliability
    - Availability
    - Fault tolerance
    - Recoverability
    - Maturity
  - Performance efficiency
    - Time-behaviour
    - Resource-utilisation
    - Capacity
  - Usability
    - Appropriateness recognisability
    - Learnability
    - Operability
    - User error protection
    - User interface aesthetics
    - Accessibility
  - Security
    - Confidentiality
    - Integrity
    - Non-repudiation
    - Accountability
    - Authenticity
  - Compatibility
    - Co-existence
    - Interoperability
  - Maintainability
    - Modularity
    - Reusability
    - Analyzability
    - Changeability
    - Modificability
    - Testability
  - Portability
    - Adaptability
    - Installability
    - Replaceability

ISO/IEC 25010 Quality Model

Quality Attribute	Definition	Architectural Requirement Example
Functional Suitability	degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions	The system must provide a safe payment method by credit card.
Reliability	degree to which a system, product or component performs specified functions under specified conditions for a specified period of time	The loss of data package must be smaller than 0,1%.
Performance efficiency	performance relative to the amount of resources used under stated conditions	The system must process any user request under 1ms
Usability	degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use	The system must provide an interface for visually impaired users.
Security	degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization	The system must use cryptographic passwords.
Compatibility	degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions, while sharing the same hardware or software environment	The system must share information with Facebook, Twitter, and Instagram.
Maintainability	degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainer	The system must take less than 2 hours to update.
Portability	degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another	The system must be compatible wirh several operational systems, including Windows, iOS, Linux, and Android.

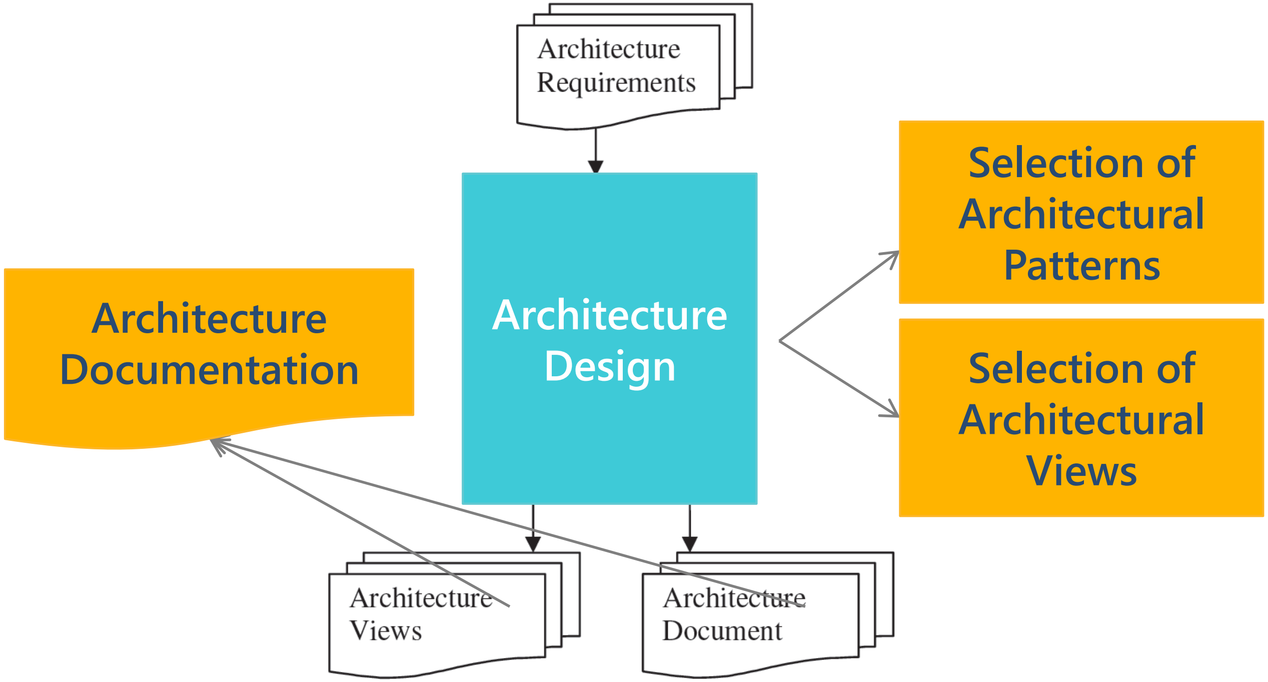
Architectural Requirements

- Quality attributes depend on each other
  - They have subtle relationships with each other
  - Example: high performance vs portability
- It is impossible to completely satisfy **all** quality attributes of a software system

© Bob Elsdale

Architectural Synthesis

- Task of finding the architectural design that meets the architectural requirements



Architecture Requirements
Architecture Design
Selection of Architectural Patterns
Selection of Architectural Views
Architecture Documentation
Architecture Views
Architecture Document

Fig. 38. Inputs and outputs of architecture design

Source: Gorton, I. 2006

Architectural Patterns

"Most of the IT applications I've worked on in the last ten years are based around a small number of well understood, proven architectures. There's a good reason for this – they work"

Ian Gorton (2006)

Architectural Patterns

- Architectural patterns dictate a particular high-level modular decomposition of the system that helps to **satisfy the essential requirements**
- One or more architectural patterns can be selected depending on the size of the system
  - Architect must specify how these patterns were incorporated in the whole solution
- **Why?** Take advantage of known, proven solutions for decreasing the risk of selecting an inappropriate architecture

⬇

Architects must understand how each pattern addresses quality atributes

Architectural Patterns

- Module Patterns
  - Describe an architecture in terms of modules
- Component and Connector Patterns
  - Describe an architecture in terms of components and connectors
  - Show software systems as a set of interacting elements at run-time
- Allocation Patterns
  - Describe an architecture as a combination of software elements and other types of elements (e.g., servers, networks, etc)

Component and Connector Patterns

Data flow Pattern

- Components act as transformers whereas connectors move data from one component’s output to another componente’s input
- It is possible when computing tasks can be devided as a sequence of transformations

Call-return Pattern

- Components interact with each other by means of syncronous calls to others provided capabilities
- Component that makes a call is paused until its request has been answered
- Connectors forward requests and return their outcome

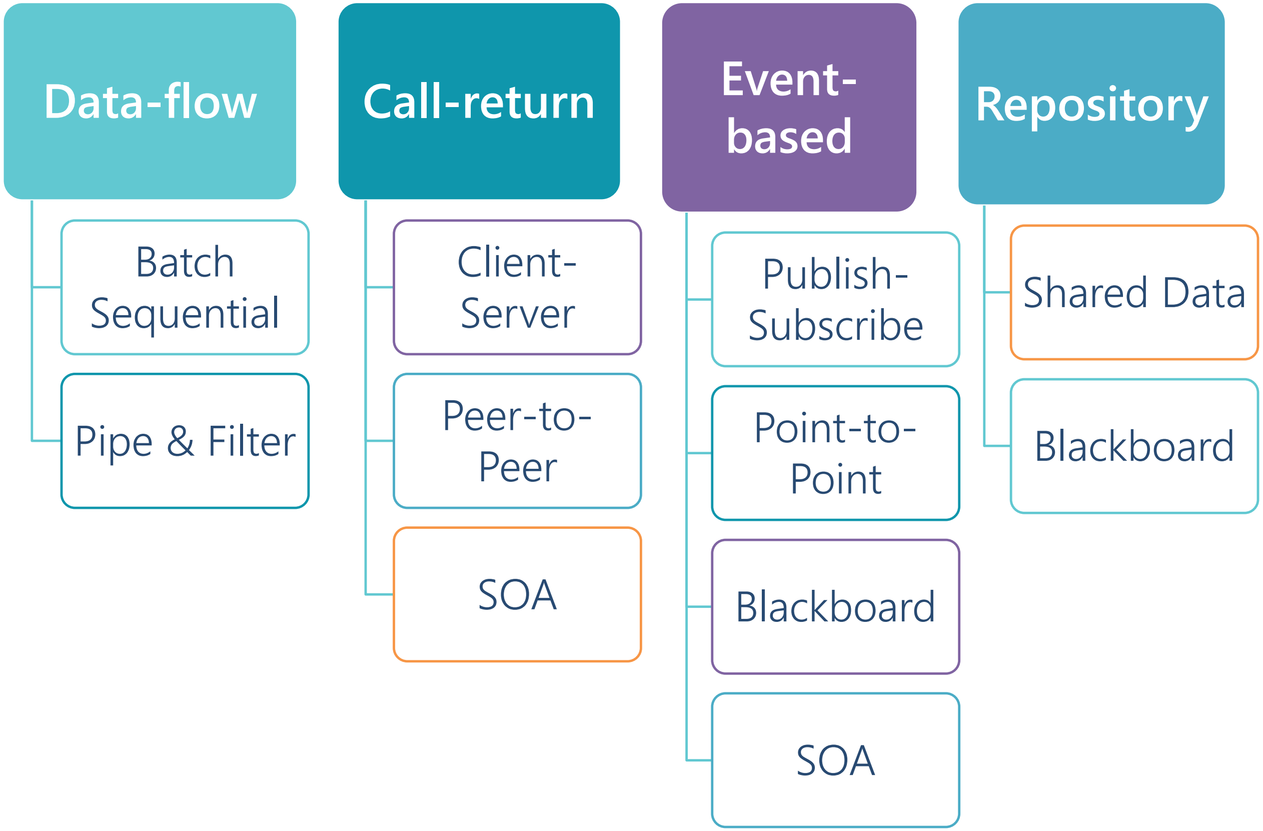
Event-based Pattern

- Components interact with each other by means of events ou assyncronous messages
- Systems are organized as loosely coupled coalitions of components

Repository Pattern

- Components interact with each other by means of sharing a data repository
- Access to this repository is mediated by DBMS, which provides a call-return interface enabling data recovery and management

Component and Connector Patterns



Component and Connector Patterns

Client-server

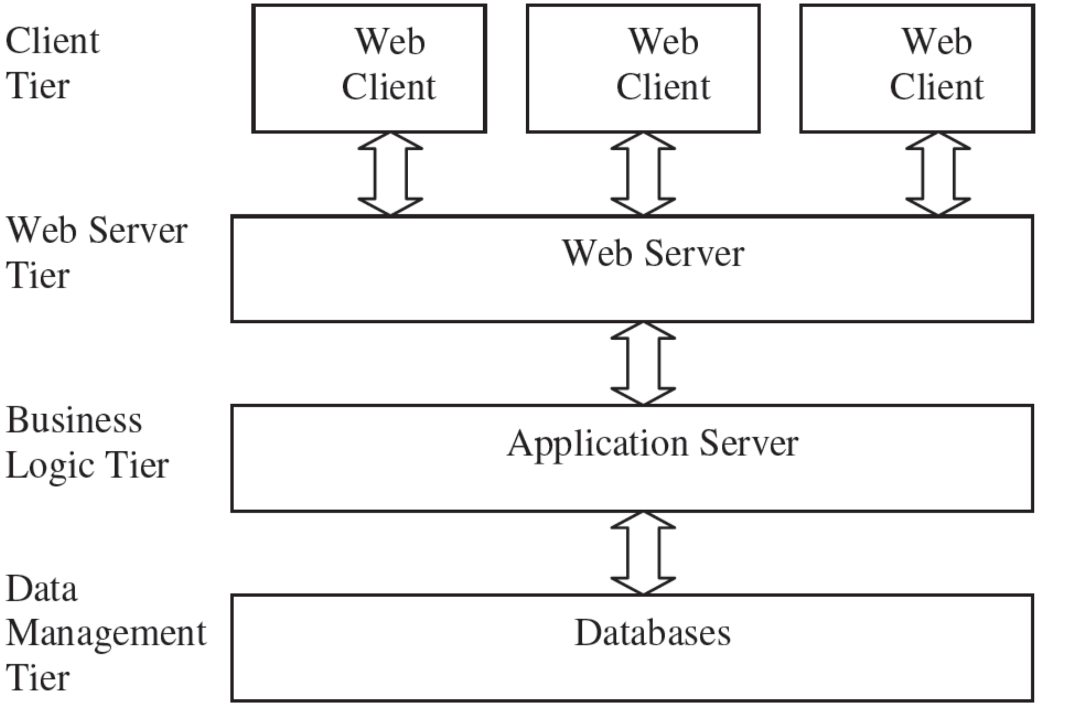
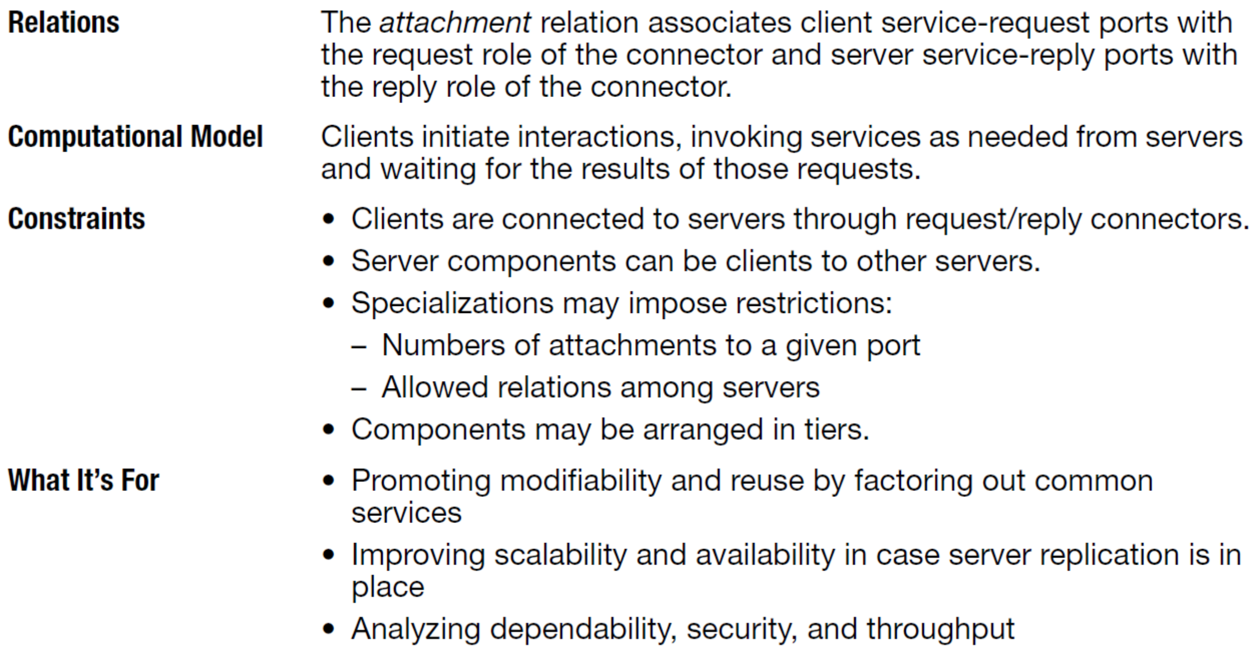


Fig. 39. N-tier client-server example

Component and Connector Patterns

Client-server



Relations	The *attachment* relation associates client service-request ports with the request role of the connector and server service-reply ports with the reply role of the connector.
Computational Model	Clients initiate interactions, invoking services as needed from servers and waiting for the results of those requests.
Constraints	• Clients are connected to servers through request/reply connectors. • Server components can be clients to other servers. • Specializations may impose restrictions: – Numbers of attachments to a given port – Allowed relations among servers • Components may be arranged in tiers.
What It's For	• Promoting modifiability and reuse by factoring out common services • Improving scalability and availability in case server replication is in place • Analyzing dependability, security, and throughput

Component and Connector Patterns

Client-server

- N-tier Client-Server properties:
  - **Separation of concerns:** Presentation, business, and data management logics are clearly separated in different layers
  - **Syncronous communication between layers:** i.e., requests come from one direction and each layer waits for their response before moving on.
  - **Flexible deployment:** all layers can be deployed to the same machine or they can be delegated to separate machines.

Component and Connector Patterns

Client-server

Availability

- Servers in different layers can be cloned so that they can be quickly replaced whenever one of them fails

Fault Tolerance

- Transparent implementation of failure control
- Client requests can be forwarded to clones

Modifiability

- Separation of concerns enables to make changes to one layer without requiring to change the others

Performance

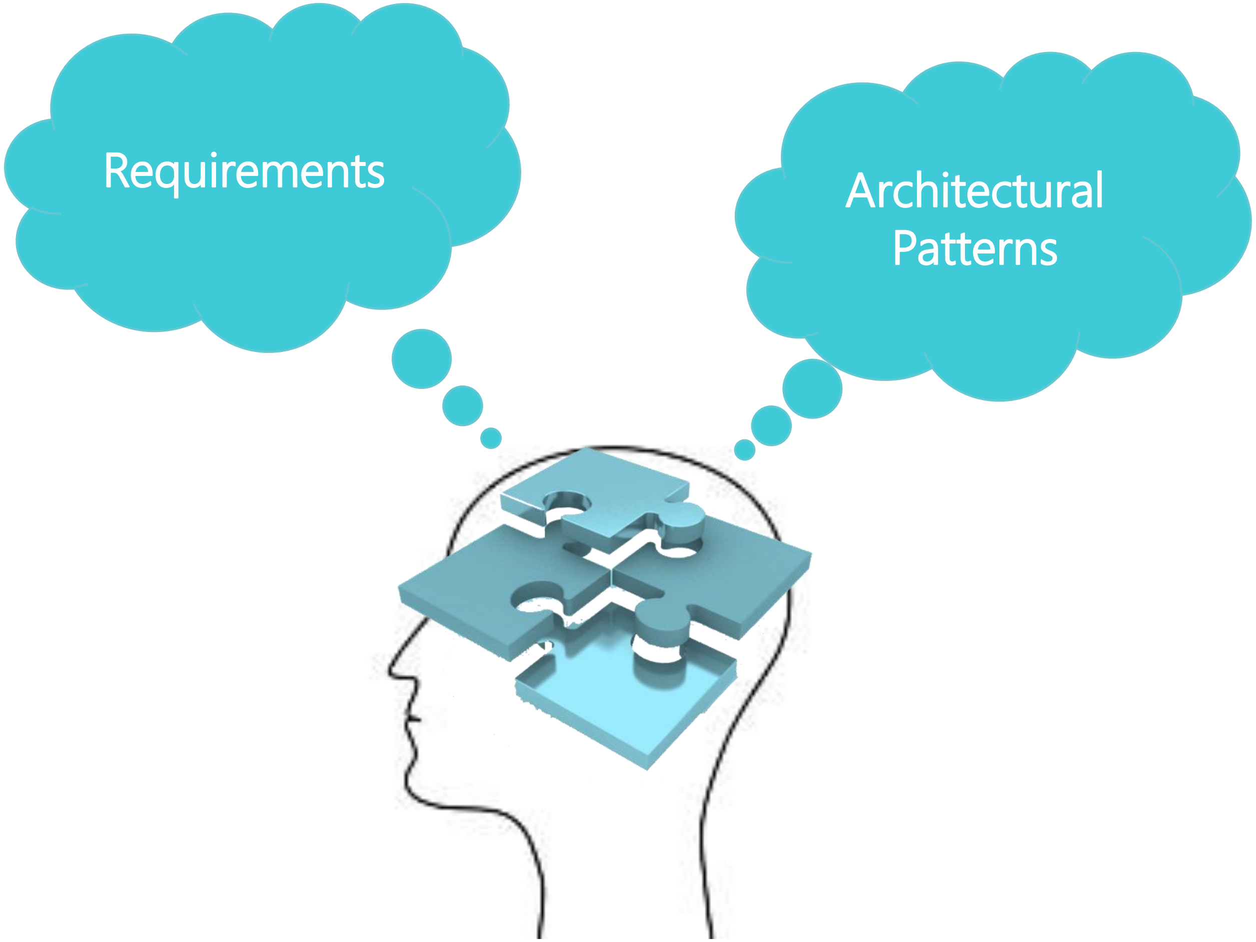
- High performance: each server can process thousands of simultaneous requests
- New client requests can be processed by servers with lower work loads

Scalability

- Servers can be cloned
- Several instances of the server can run on the same machine or different machines
- Potential bottleneck: Data management (DBMS)

Architectural Description

Architectural Decisions

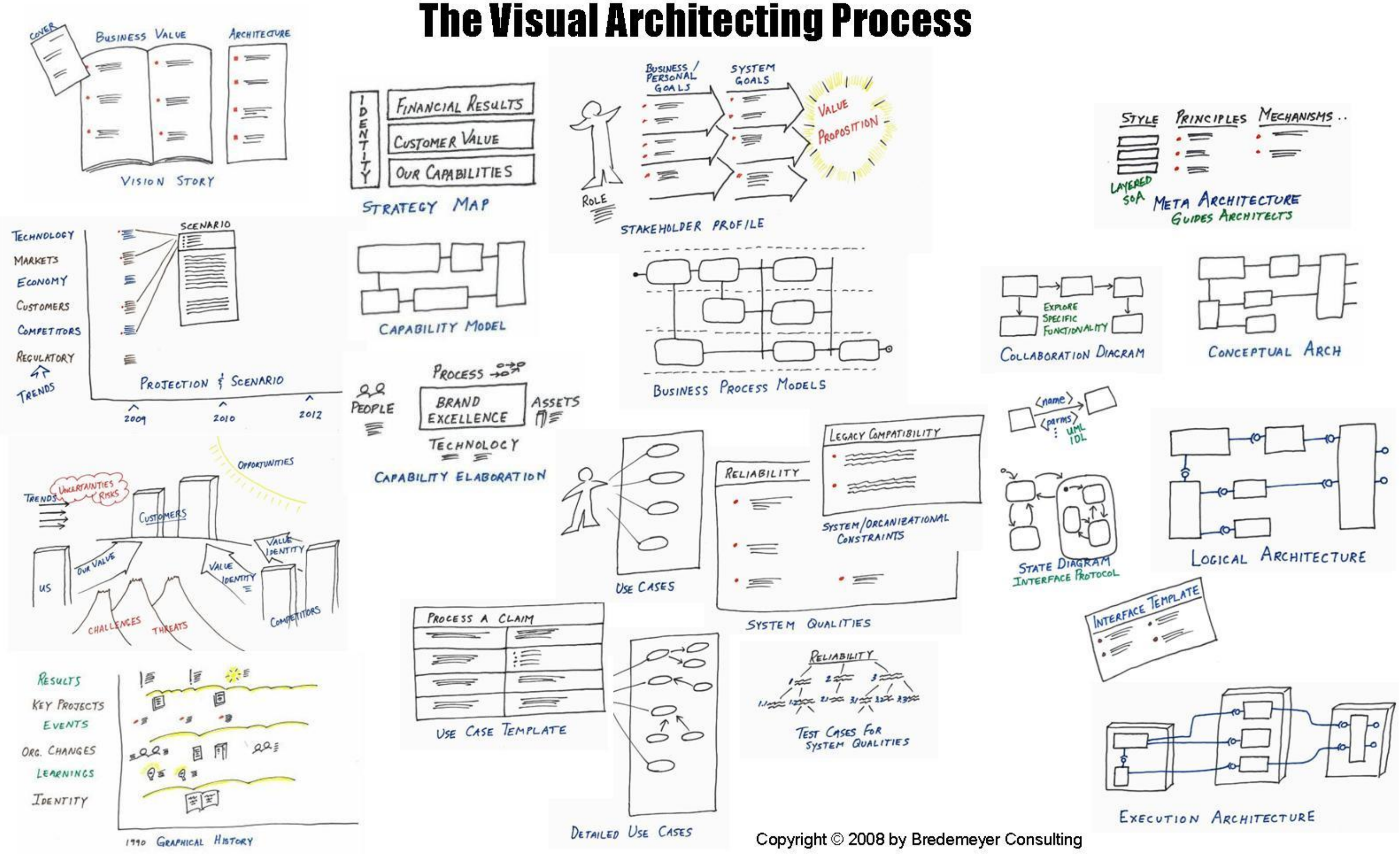


Requirements

Architectural Patterns

Software Architect

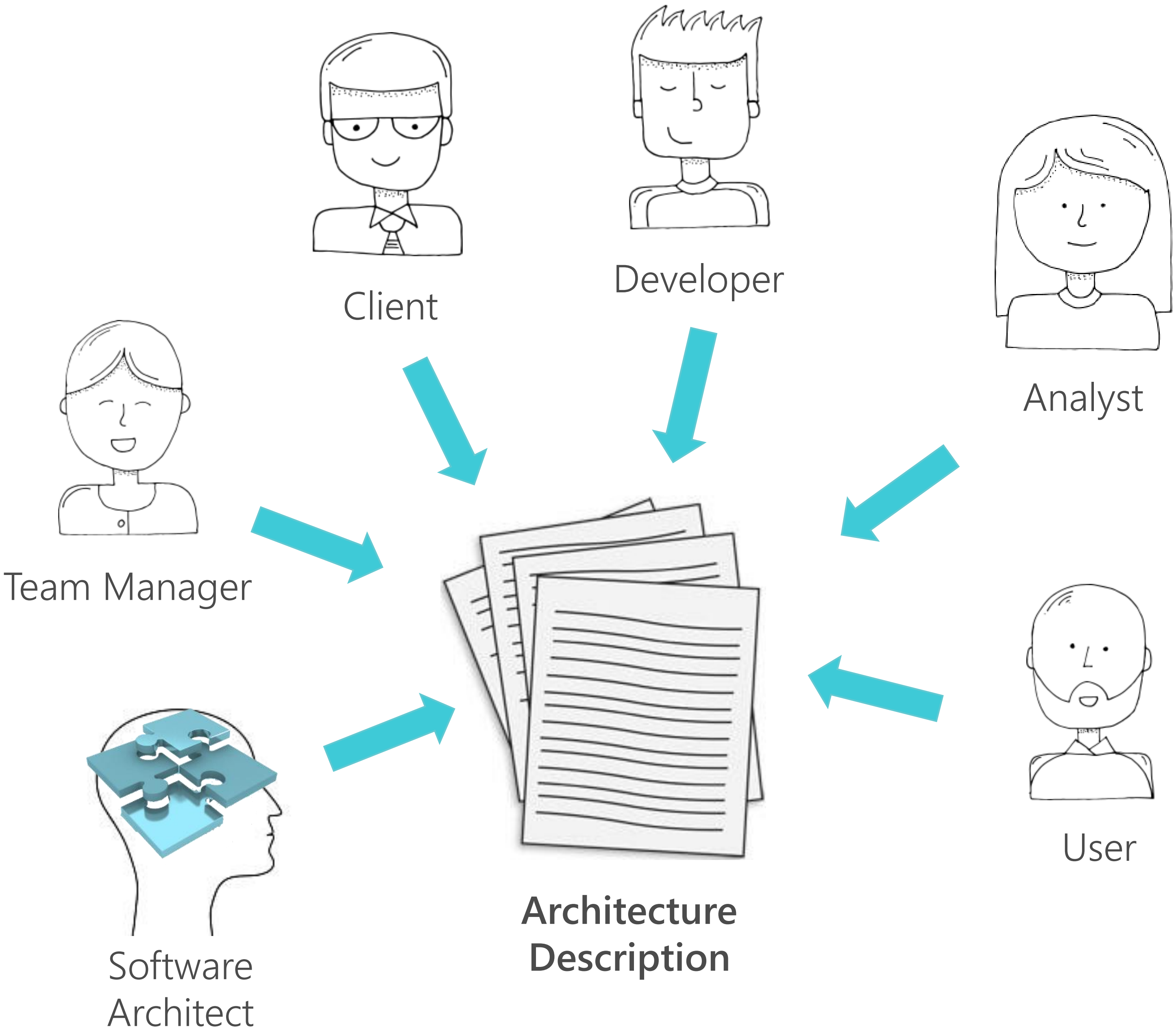
The Visual Architecting Process



Copyright © 2008 by Bredemeyer Consulting

Architecture Description

- Main artifact expressing the software architecture
- Applications:
  - Communicating and sharing architectural knowledge
  - Assessing and analyzing systems qualities
  - Evolving software systems
- Impacts on feasibility, usability, and maintainability of software systems



Client

Developer

Analyst

Team Manager

User

Software Architect

Architecture Description

Architecture Description

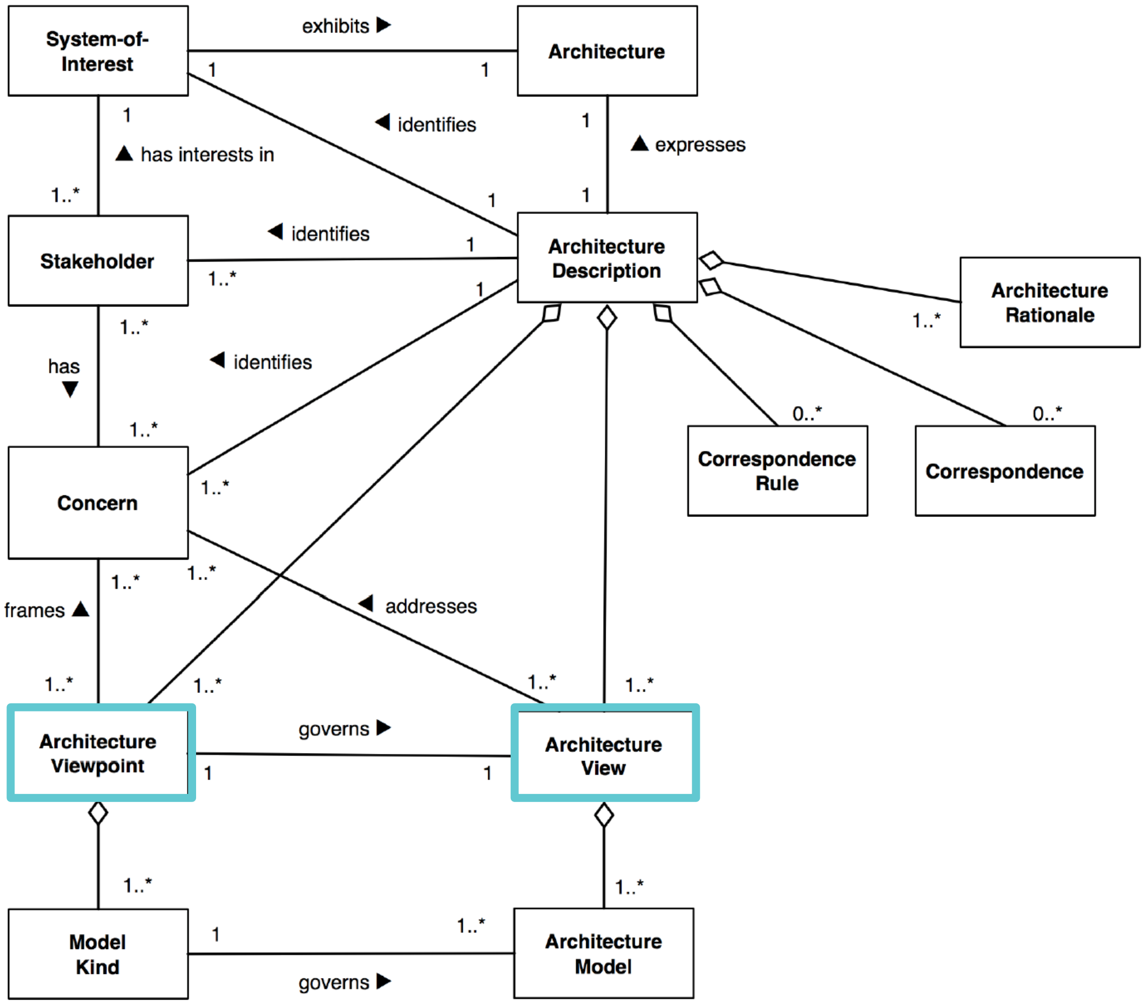
- Targeted for specific stakeholders
- Addresses different concerns
  - Functionality, security, cost, performance, among others
- Different views
  - Each of them conforms to a given viewpoint

Architecture Description Language (ADL)

- Mechanisms for expressing composition, abstraction, reusability, configuration, and analysis of software architectures
- Challenges for describing software architectures:
  - Runtime perspective
  - Dynamic perspective
  - Mobile perspective

Conceptual Model of an Architecture Description

ISO/IEC/IEEE 42010



Architecture Description

ISO/IEC/IEEE 42010

Viewpoint

- Artifact establishing the conventions (i.e., model kinds) for the construction, interpretation and use of architecture views to frame specific system concerns

View

- Artifact expressing the architecture from the perspective of specific system concerns

Architecture Framework

Establishes a common practice for creating, interpreting, and analyzing architecture descriptions for a particular *domain* or stakeholders community

4+1 Views

- Logical Viewpoint
- Process Viewpoint
- Development Viewpoint
- Physical Viewpoint
- Use Case Viewpoint

Views & Beyond

- Module Viewpoint
- Components and Connectors Viewpoint
- Deployment Viewpoint

Architecture Description

- The set of viewpoints describing an architecture can vary for each system
  - Takes into account stakeholders’ concerns
  - Takes into account architect’s goals
- Each viewpoint can highlight a particular element and/or relationship in the system, e.g.:
  - A layer view can be useful for describing portability
  - A deployment view can be useful for describing performance and reliability

ADLs Traditional Definitions

- [ADLs] provide mechanisms for expressing composition, abstraction, reusability, configuration, and analysis of software architectures (Shaw and Garlan, 1994)
- An ADL must explicitly model components, connectors, and their configurations; furthermore, to be truly usable and useful, it must provide tool support for architecture-based development and evolution (Medvidovic and Taylor, 2001)

ADLs Characteristics

- Architecture building blocks
  - Components
  - Connectors
  - Configurations
- Tool Support
  - Enable automated analyses on the architecture description

ADLs Characteristics

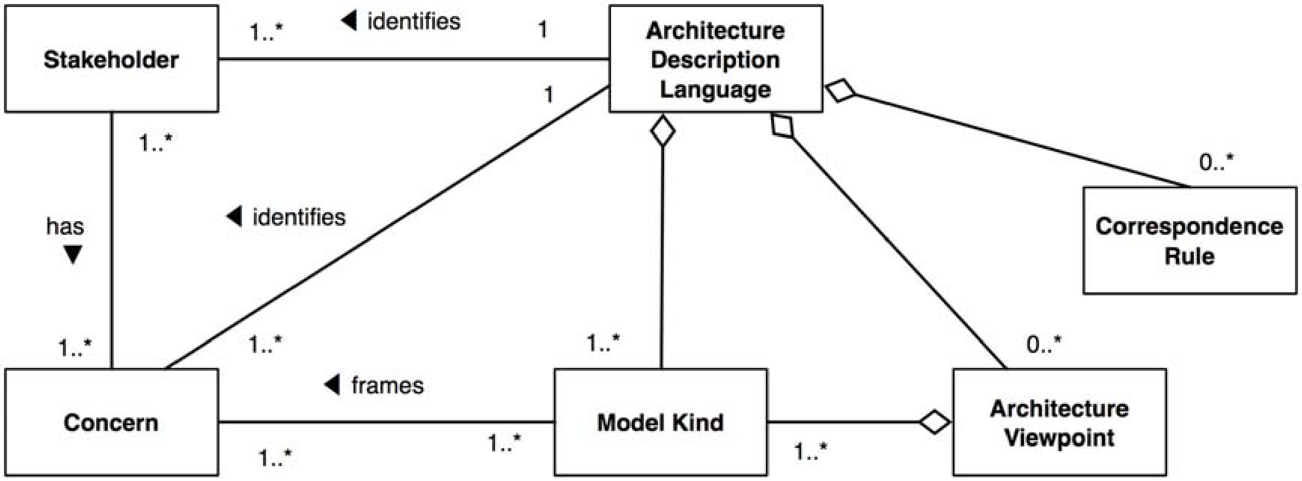
- Components and Connectors
  - Interface
  - Type
  - Semantics
  - Constraints
  - Evolution
  - Non-functional properties
- Tool Support
  - Active specification
  - Multiple views
  - Analysis
  - Refinement
  - Implementation generation
  - Dynamism

- (Architectural) Configuration
  - Understandability
  - Compositionality
  - Refinement and traceability
  - Heterogeneity
  - Scalability
  - Evolution
  - Dynamism
  - Constraints
  - Non-functional properties

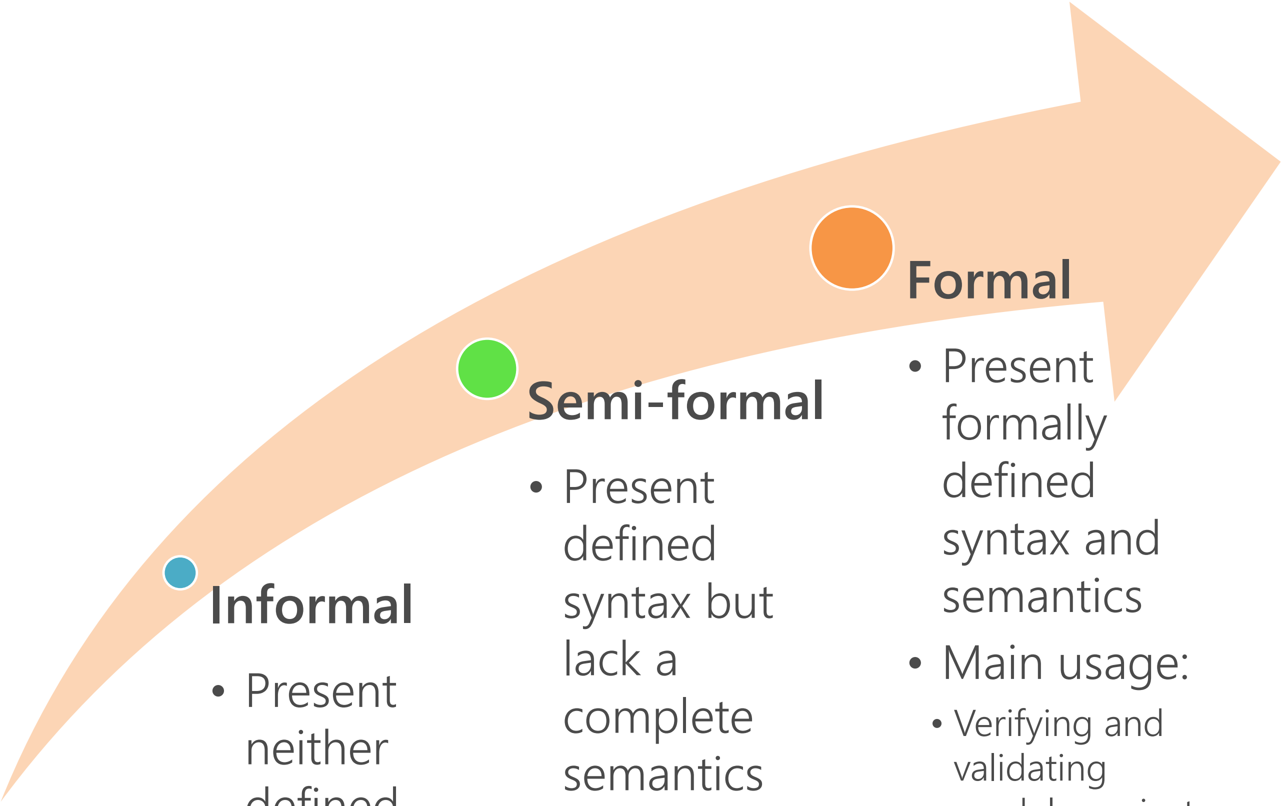
Source: Medvidovic, N. and Taylor, R. N., 2000.

ADL Conceptual Model

ISO/IEC/IEEE 42010



ADL Formalism Level



Informal

- Present neither defined syntax or semantics
- Main usage:
  - Illustrating or exemplifying concepts

Semi-formal

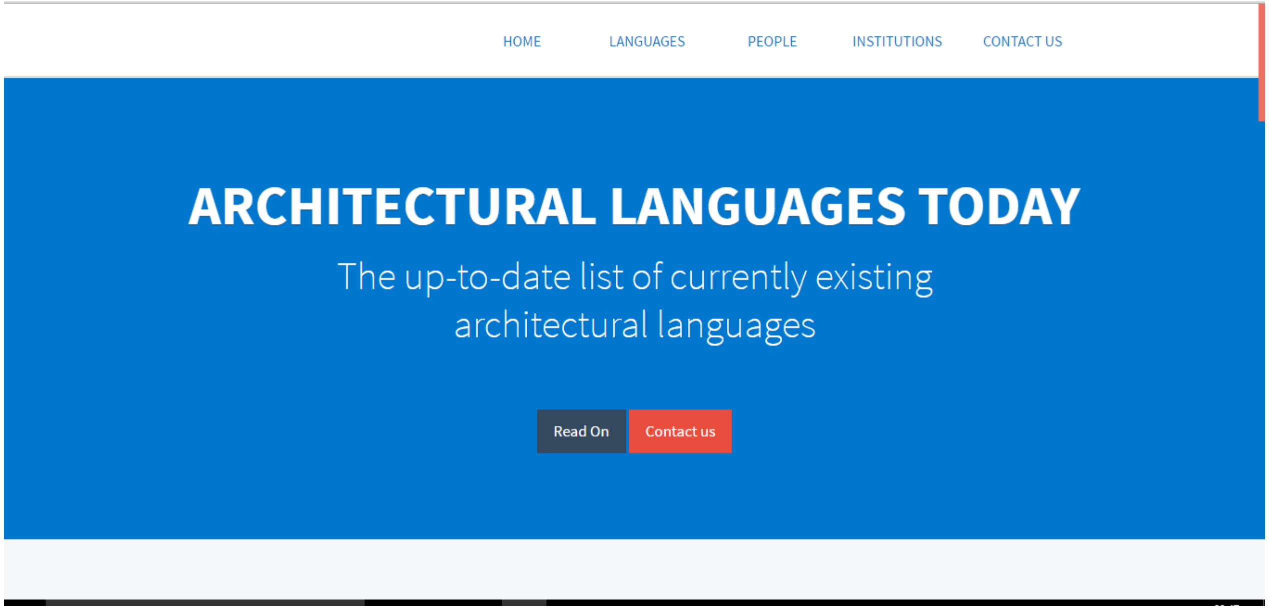
- Present defined syntax but lack a complete semantics
- Main usage:
  - Supporting communication among stakeholders

Formal

- Present formally defined syntax and semantics
- Main usage:
  - Verifying and validating models against properties and quality attributes

ADL Example

- Many, many, many ADLs…
  - 123!!

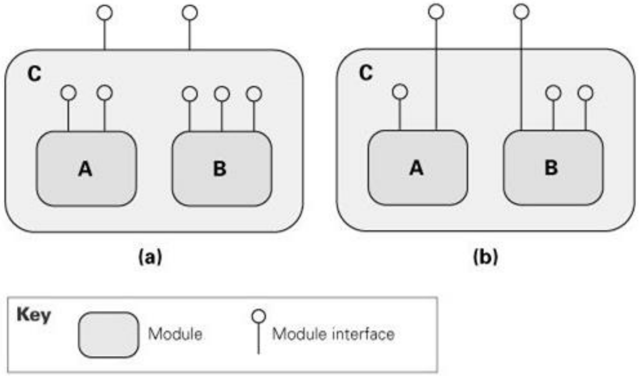


ARCHITECTURAL LANGUAGES TODAY

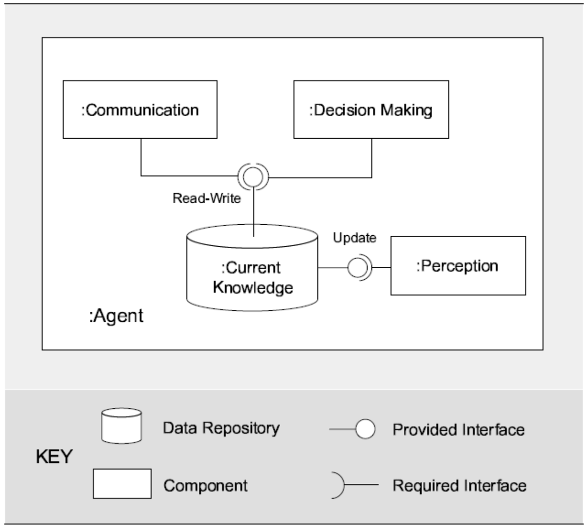
The up-to-date list of currently existing architectural languages

Source: http://www.di.univaq.it/malavolta/al/

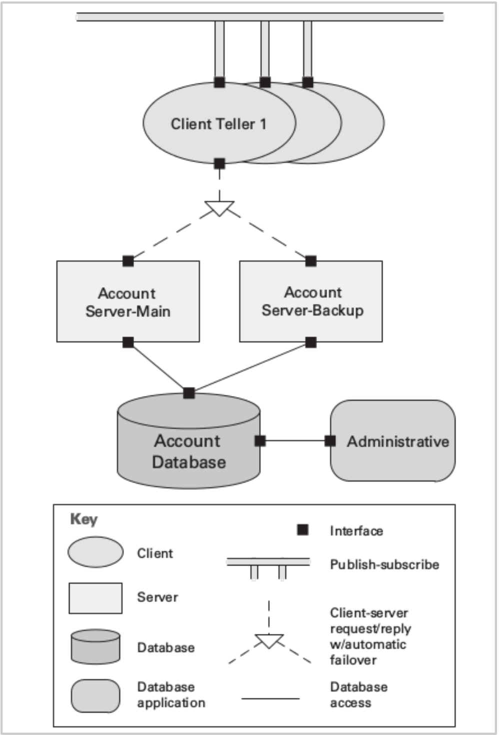
Informal ADL Example



*1. Modules can **(a)** provide interfaces, hiding other modules, or **(b)** exposing some interfaces of internal modules*



3. Shared data view of an agent



2. A bird's-eye-view of a system as it appears at run-time.

Source:
1,2 Clements, P. et al., 2011
3 Weyns, D. An Architecture-Centric Approach for Software Engineering with Situated Multiagent Systems. PhD Thesis. 2006. Available at: http://www.cs.kuleuven.be/publicaties/doctoraten/cw/CW2006_09.abs.html

Semi-formal ADL Example

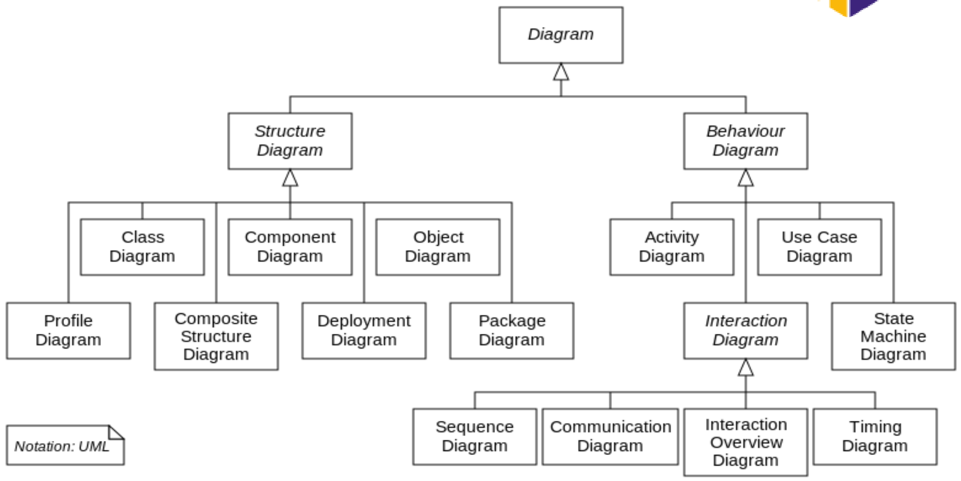
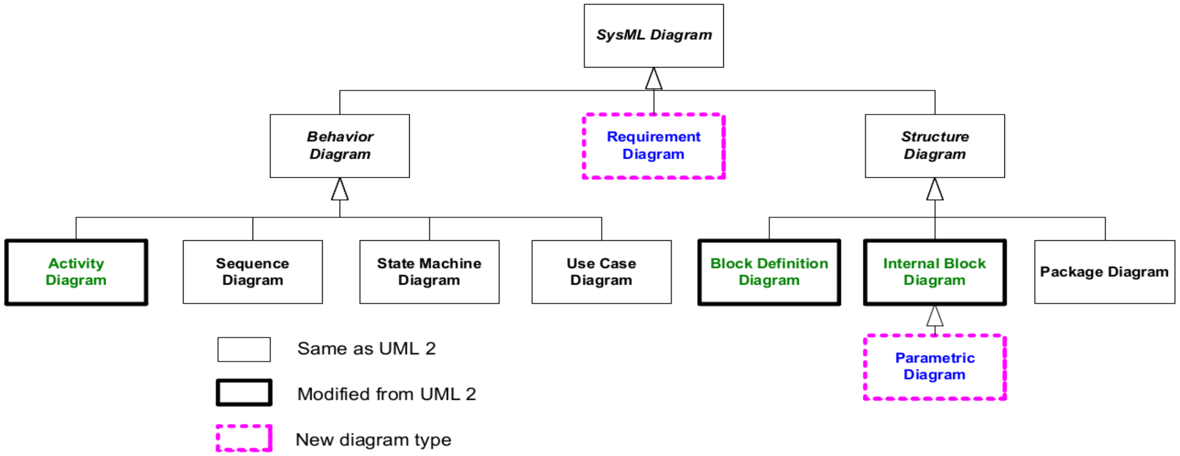


Diagram
Structure Diagram
Behaviour Diagram
Class Diagram
Component Diagram
Object Diagram
Profile Diagram
Composite Structure Diagram
Deployment Diagram
Package Diagram
Activity Diagram
Use Case Diagram
Interaction Diagram
State Machine Diagram
Sequence Diagram
Communication Diagram
Interaction Overview Diagram
Timing Diagram
Notation: UML

1. UML 2.x diagram types

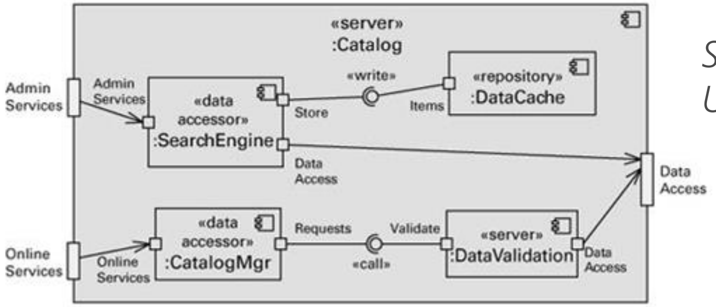


SysML Diagram
Behavior Diagram
Requirement Diagram
Structure Diagram
Activity Diagram
Sequence Diagram
State Machine Diagram
Use Case Diagram
Block Definition Diagram
Internal Block Diagram
Package Diagram
Parametric Diagram
Same as UML 2
Modified from UML 2
New diagram type

2. SysML 1.x diagram types

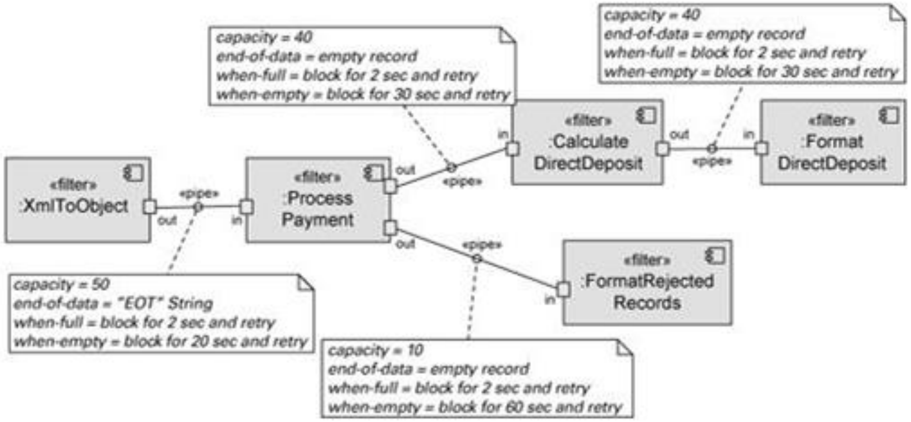
Source:
1 http://www.omg.org/spec/UML/2.5/
2 http://www.omg.org/spec/SysML/1.4/

Semi-formal ADL Example



Substructure of a UML component

UML diagram of a pipe-and-filter view

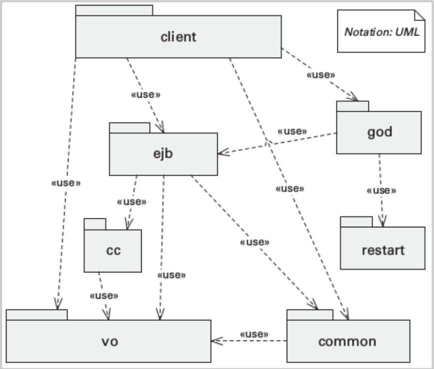


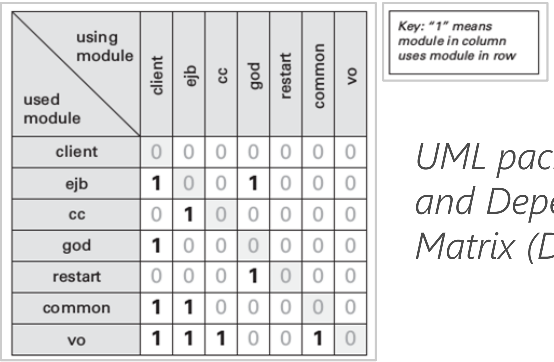
capacity = 40
end-of-data = empty record
when-full = block for 2 sec and retry
when-empty = block for 30 sec and retry

capacity = 40
end-of-data = empty record
when-full = block for 2 sec and retry
when-empty = block for 30 sec and retry

capacity = 50
end-of-data = "EOT" String
when-full = block for 2 sec and retry
when-empty = block for 20 sec and retry

capacity = 10
end-of-data = empty record
when-full = block for 2 sec and retry
when-empty = block for 60 sec and retry





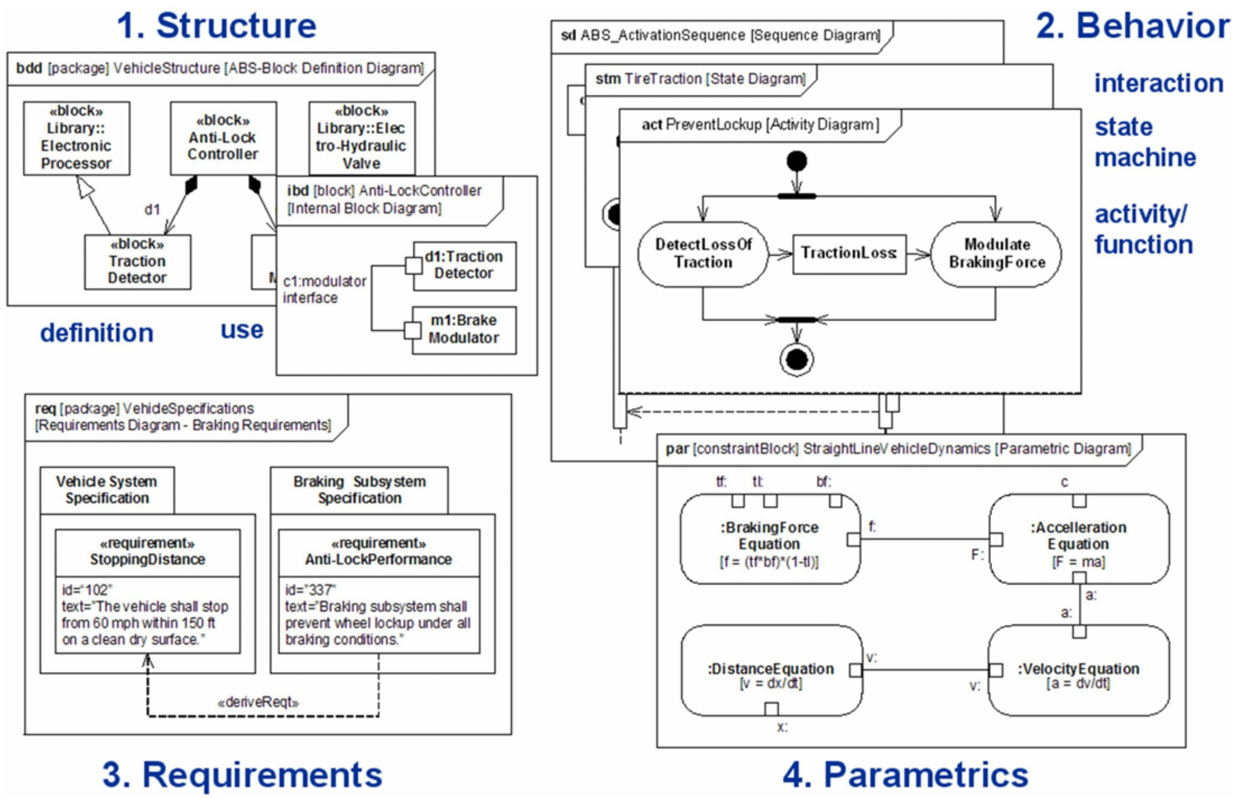
using module / used module	client	ejb	cc	god	restart	common	vo
client	0	0	0	0	0	0	0
ejb	1	0	0	1	0	0	0
cc	0	1	0	0	0	0	0
god	1	0	0	0	0	0	0
restart	0	0	0	1	0	0	0
common	1	1	0	0	0	0	0
vo	1	1	1	0	0	1	0

Key: "1" means module in column uses module in row

UML package diagram (left) and Dependency Structure Matrix (DSM) (right)

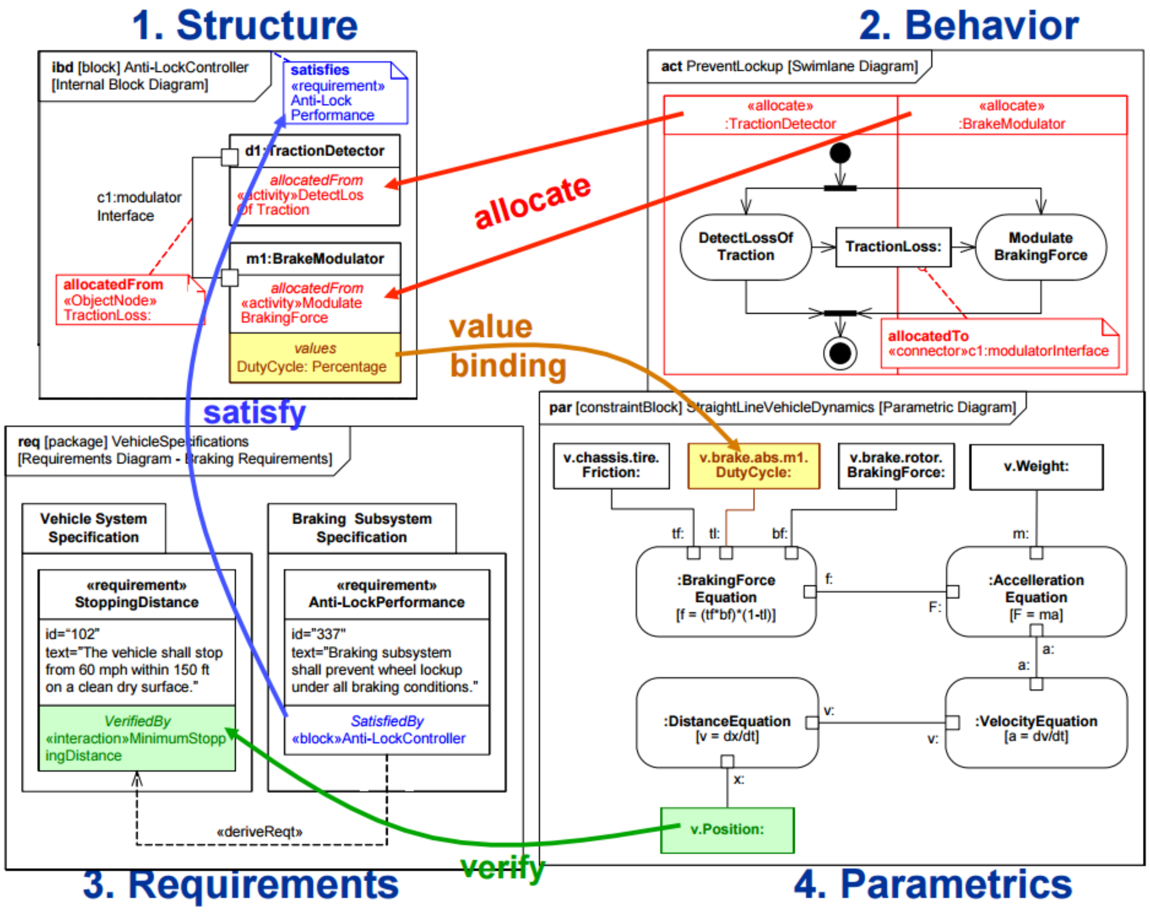
Source: Clements, P. et al., 2011

Semi-formal ADL Example: SysML


1. Structure
bdd [package] VehicleStructure [ABS-Block Definition Diagram]
«block» Library:: Electronic Processor
«block» Anti-Lock Controller
«block» Library::Electro-Hydraulic Valve
d1
«block» Traction Detector
ibd [block] Anti-LockController [Internal Block Diagram]
c1:modulator interface
d1:Traction Detector
m1:Brake Modulator
definition
use
2. Behavior
sd ABS_ActivationSequence [Sequence Diagram]
stm TireTraction [State Diagram]
act PreventLockup [Activity Diagram]
DetectLossOf Traction
TractionLoss
Modulate BrakingForce
interaction
state machine
activity/ function
req [package] VehicleSpecifications [Requirements Diagram - Braking Requirements]
Vehicle System Specification
Braking Subsystem Specification
«requirement» StoppingDistance
id="102"
text="The vehicle shall stop from 60 mph within 150 ft on a clean dry surface."
«requirement» Anti-LockPerformance
id="337"
text="Braking subsystem shall prevent wheel lockup under all braking conditions."
«deriveReqt»
3. Requirements
par [constraintBlock] StraightLineVehicleDynamics [Parametric Diagram]
tf:
tl:
bf:
:BrakingForce Equation
[f = (tf*bf)*(1-tl)]
f:
F:
c
:Accelleration Equation
[F = ma]
a:
a:
:DistanceEquation
[v = dx/dt]
v:
v:
:VelocityEquation
[a = dv/dt]
x:
4. Parametrics

Source: http://www.omgsysml.org/

Semi-formal ADL Example: SysML



1. Structure

ibd [block] Anti-LockController [Internal Block Diagram]

satisfies «requirement» Anti-Lock Performance

d1:TractionDetector

allocatedFrom «activity»DetectLossOf Traction

c1:modulator Interface

m1:BrakeModulator

allocatedFrom «activity»Modulate BrakingForce

values DutyCycle: Percentage

allocatedFrom «ObjectNode» TractionLoss:

2. Behavior

act PreventLockup [Swimlane Diagram]

«allocate» :TractionDetector

«allocate» :BrakeModulator

DetectLossOf Traction

TractionLoss:

Modulate BrakingForce

allocatedTo «connector»c1:modulatorInterface

allocate

value binding

satisfy

3. Requirements

req [package] VehicleSpecifications [Requirements Diagram - Braking Requirements]

Vehicle System Specification

«requirement» StoppingDistance

id="102"
text="The vehicle shall stop from 60 mph within 150 ft on a clean dry surface."

VerifiedBy «interaction»MinimumStoppingDistance

Braking Subsystem Specification

«requirement» Anti-LockPerformance

id="337"
text="Braking subsystem shall prevent wheel lockup under all braking conditions."

SatisfiedBy «block»Anti-LockController

«deriveReqt»

verify

4. Parametrics

par [constraintBlock] StraightLineVehicleDynamics [Parametric Diagram]

v.chassis.tire. Friction:

v.brake.abs.m1. DutyCycle:

v.brake.rotor. BrakingForce:

v.Weight:

tf: tl: bf: m: f: F: a: a: v: v: x:

:BrakingForce Equation [f = (tf*bf)*(1-tl)]

:Accelleration Equation [F = ma]

:DistanceEquation [v = dx/dt]

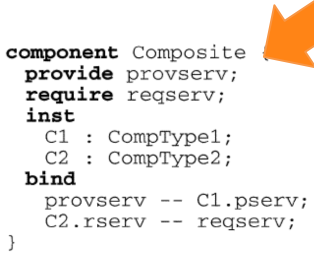
:VelocityEquation [a = dv/dt]

v.Position:

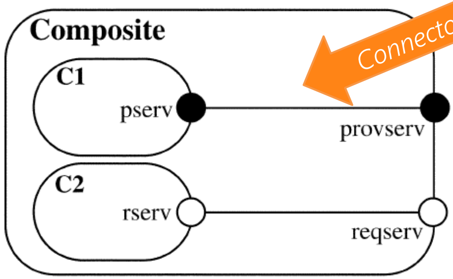
Source: http://www.omgsysml.org/

Formal ADL Example

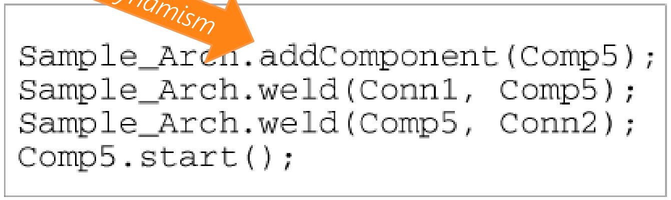
Connectors



```
component Composite {
  provide provserv;
  require reqserv;
  inst
    C1 : CompType1;
    C2 : CompType2;
  bind
    provserv -- C1.pserv;
    C2.rserv -- reqserv;
}
```



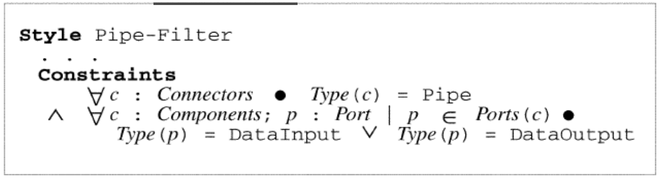
A composite component specified in *Darwin* (top) and (bottom) the graphical view of the component



```
Sample_Arch.addComponent(Comp5);
Sample_Arch.weld(Conn1, Comp5);
Sample_Arch.weld(Comp5, Conn2);
Comp5.start();
```

Dynamic insertion of a component into a *C2SADEL* architecture.

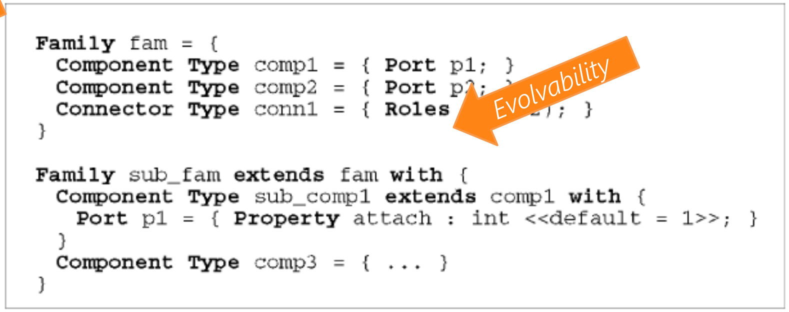
Constraints



```
Style Pipe-Filter
  . . .
  Constraints
      ∀c : Connectors • Type(c) = Pipe
  ∧  ∀c : Components; p : Port | p ∈ Ports(c) •
         Type(p) = DataInput ∨ Type(p) = DataOutput
```

The pipes-and-filters style declared in *Wright*.

Configurations


Evolvability

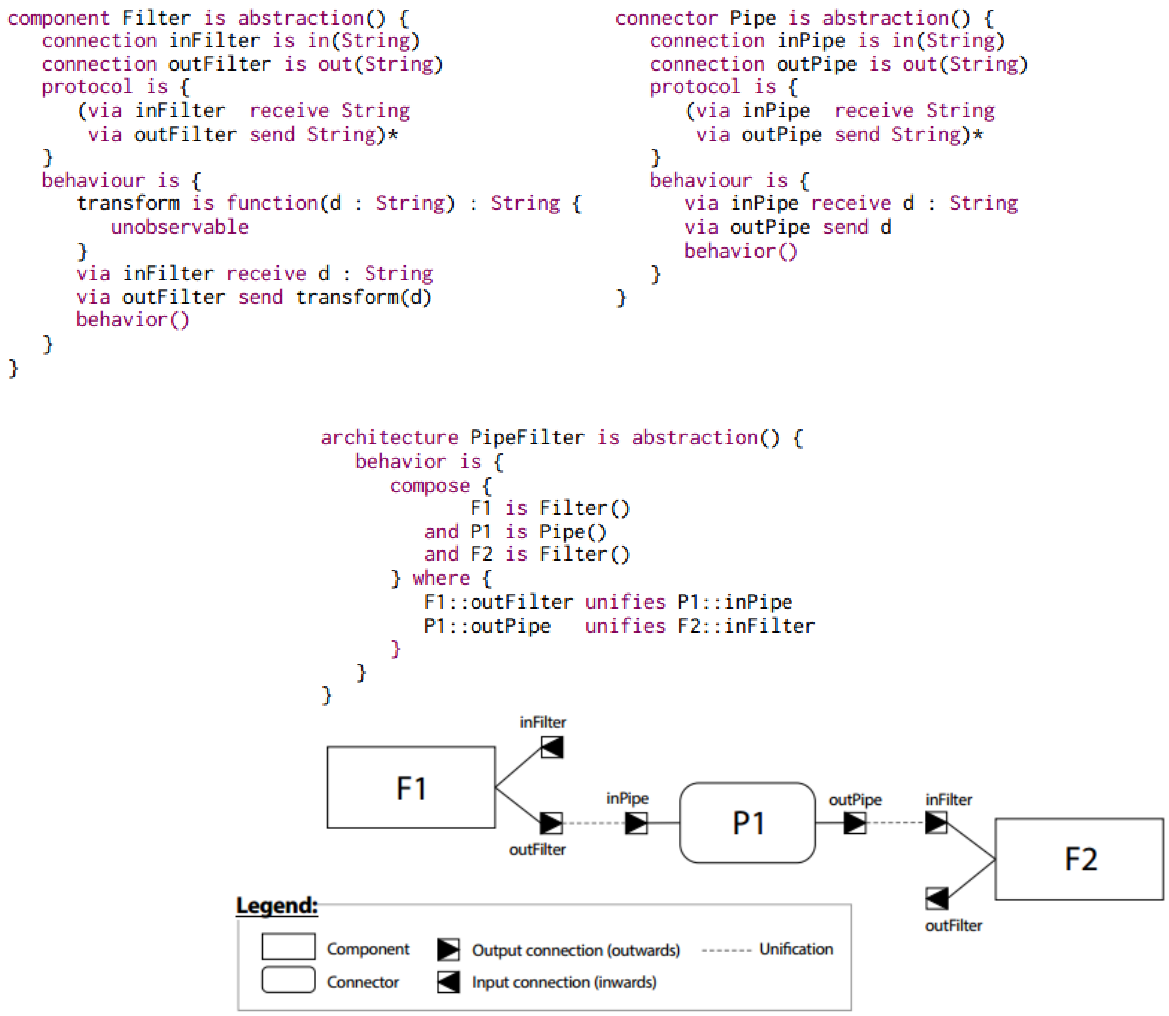
```
Family fam = {
  Component Type comp1 = { Port p1; }
  Component Type comp2 = { Port p2; }
  Connector Type conn1 = { Roles {r1; r2}; }
}

Family sub_fam extends fam with {
  Component Type sub_comp1 extends comp1 with {
    Port p1 = { Property attach : int <<default = 1>>; }
  }
  Component Type comp3 = { ... }
}
```

Declaration in *ACME* of a family of architectures, ***fam***, and its subfamily, ***sub_fam***, which has new components and properties

Source: Medvidovic, N. and Taylor, R. N., 2000.

Formal ADL Example: π-ADL



```
component Filter is abstraction() {
   connection inFilter is in(String)
   connection outFilter is out(String)
   protocol is {
      (via inFilter  receive String
       via outFilter send String)*
   }
   behaviour is {
      transform is function(d : String) : String {
         unobservable
      }
      via inFilter receive d : String
      via outFilter send transform(d)
      behavior()
   }
}
```

```
connector Pipe is abstraction() {
   connection inPipe is in(String)
   connection outPipe is out(String)
   protocol is {
      (via inPipe  receive String
       via outPipe send String)*
   }
   behaviour is {
      via inPipe receive d : String
      via outPipe send d
      behavior()
   }
}
```

```
architecture PipeFilter is abstraction() {
   behavior is {
      compose {
             F1 is Filter()
         and P1 is Pipe()
         and F2 is Filter()
      } where {
         F1::outFilter unifies P1::inPipe
         P1::outPipe   unifies F2::inFilter
      }
   }
}
```

Description of a simple pipeline architecture

Source: Cavalcante, E., Oquendo, F., Batista, T. Architecture-Based Code Generation: From π-ADL Architecture Descriptions to Implementations in the Go Language. ECSA 2015.

Why formal?

- Formalizing software architecture descriptions
  - Models must be scalable
  - Multiple formal methods must be supported
    - using multiple ADLs to model a single system
    - formalizing different aspects of a system in a single ADL
  - Incremental formalization must be supported
    - how do you formalize in the face of incompleteness?
    - **Formalize only and exactly as much as necessary**
  - Analysis results must be transferable to design and implementation
    - what good is deadlock detection at architecture alone?

Source: http://sunset.usc.edu/classes/cs599_2000/September7b.pdf

What industry needs from architectural languages?

- 48 practitioners
- Use of ADLs:
  - 86% use UML or an UML profile,
  - 9% use ad hoc or in-house languages (e.g., AADL, ArchiMate)
  - 5% do not use any ADL
- Needs of ADLs:
  - Design (~66%), communication support (~36%), and analysis support (~30%)
  - Code generation and deployment support (~12% percent) and development process and methods support (~18%)
- Limitations of ADLs:
  - Insufficient expressiveness for non-functional properties (~37%)
  - Insufficient communication support for non-architects (~25%)
  - Lack of formality (~18%)

Source: Malavolta, I. et al. 2013.

What industry needs from architectural languages?

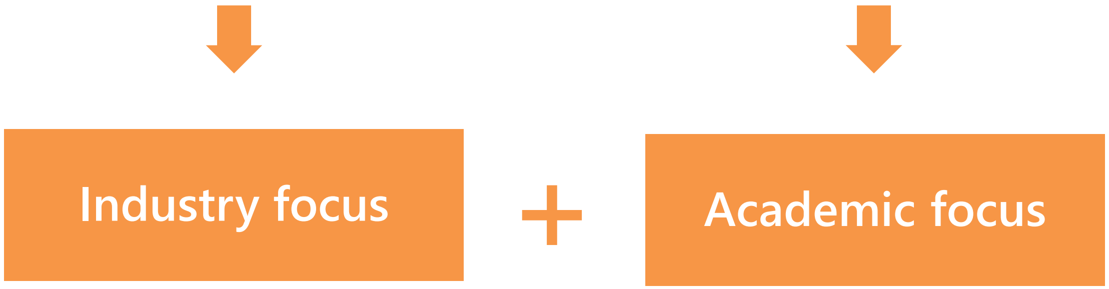
Extrovert VS Introvert

Extrovert

- Communicates the architecture to the stakeholders involved in the architecting phase
- ADLs must be simple and intuitive

Introvert

- Analyzes the architectural design
- ADLs must enable formality so to drive analysis and other automatic tasks



Industry focus + Academic focus

Source: Malavolta, I. et al., 2013.

Architectural Evaluation

Architectural Evaluation

- Architectures are not inherently good or bad, they are only well-suited or not with respect to a particular set of goals
- Questions:

  a. Will the solution meet the quality requirements?
  b. Do we have sufficient resources for developing the solution?
  c. Did we take the right architectural decisions?

and many more…

Architectural Evaluation

Architecture Evaluation

Checks

Architectural-significant decisions

Against

Architectural-significant requirements

The sooner the better

Architectural Evaluation Types

- Quantitative: How much …?
  - Estimation
  - Analytical or simulation models
  - Measurements on feasibility prototypes or products
- Qualitative: What if …?
  - Questioning techniques: questionnaires & checklists
  - Based on scenarios: e.g., ATAM, SAAM, …
  - Prototyping (proof-of-concept)
- Evaluation mostly uses **scenarios**[1] to verify quality attributes

[1] Short statement describing an interaction of one of the stakeholders with the system

Architectural Evaluation

- **When?**
  - Architecture is defined and before or after implementation is completed
    - Before: iterative evaluation of architecture decisions
    - After: Encompasses understanding legacy systems and checking if they meet quality requirements
- **Who?**
  - Domain and technical stakeholders should participate. The evaluation team should not be drawn from the project staff
- **Input**
  - Architecture description
    - Completeness and reliability of the evaluation depends on the description
- **Outputs**
  - Prioritized list of quality requirements
  - ~~Good/bad~~, ~~6/10~~, where are the risks

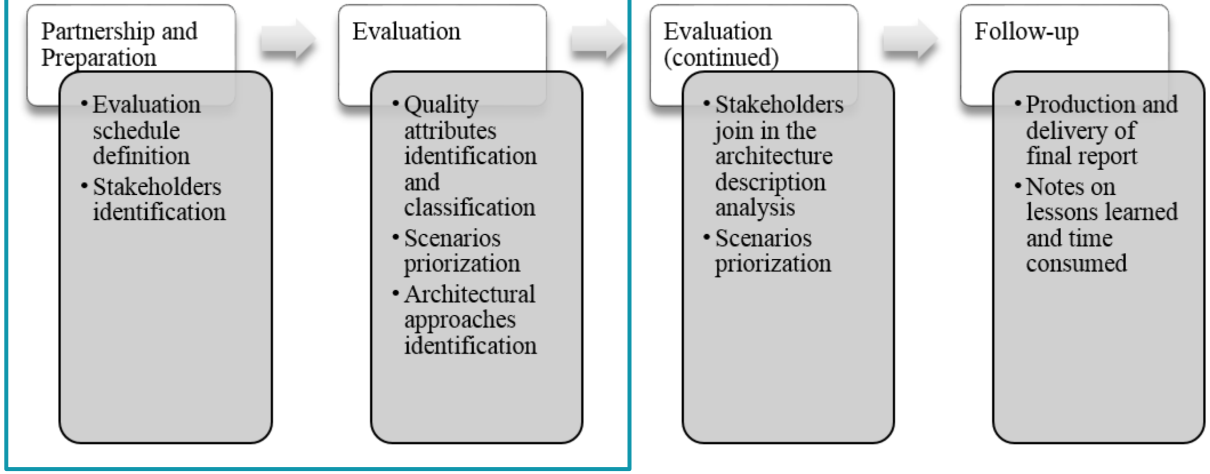
Architecture Tradeoff Analysis Method (ATAM)

1st meeting

Who: evaluation meeting and project decision makers

2nd meeting

Who: evaluation meeting, project decision makers, and all stakeholders



Partnership and Preparation

- Evaluation schedule definition
- Stakeholders identification

Evaluation

- Quality attributes identification and classification
- Scenarios priorization
- Architectural approaches identification

Evaluation (continued)

- Stakeholders join in the architecture description analysis
- Scenarios priorization

Follow-up

- Production and delivery of final report
- Notes on lessons learned and time consumed

Trending Topics in Software Architecture

- Reference Architectures
- Architectural Evolution
- Models @ Runtime
- Sustanaible Architectures
  - Green, Technical Debt

and many more… 😊

Systems-of-Systems

Part II

SoSs

- Independent constituent systems
  - Action and decision making
- Geographic distribution
- Evolutionary development
- Emergent behavior

Image Source: https://ec.europa.eu/digital-single-market/system-systems

SoSs

- Open systems
  - Top
    - Continually open for addition of new applications and systems, whithout any top-level system defining the SoS
    - *Emergent behavior*
  - Bottom
    - The lowest level of the SoS (e.g., communication stack) may be changed at any time
    - *Interoperability*
  - Continually evolving
    - **An SoS is never complete** as it evolves **at run-time** according to changes in the surrounding environment

Source: Abbott, 2006.

SoSs Potential Pitfalls

1. Acquisition management and staffing
2. **Requirements/architecture feasibility**
3. Achievable software schedules
4. **Supplier integration**
5. **Adaptation to rapid change**
6. Systems and software quality factor achievability
7. Product integration and electronic upgrade
8. Commercial off-the-shelf (COTS) software and reuse feasibility
9. **External interoperability**
10. Technology readiness

Source: Boehm, B. et al. 2004

Global Earth Observing System of Systems (GEOSS)

GROUP ON EARTH OBSERVATIONS

SoSs Example

GEOSS

- GEOSS is to be a global, coordinated, comprehensive and sustained system of Earth observing systems
  - Promote coordinated access to data and products produced amongst all contributing systems
- Introduces consistency of content through guidelines to data providers for the appropriate characterization of the observing systems and their derived products
  - Adoption of **standardized best practices**

Source: http://earthobservations.org

SoSs Example

GEOSS

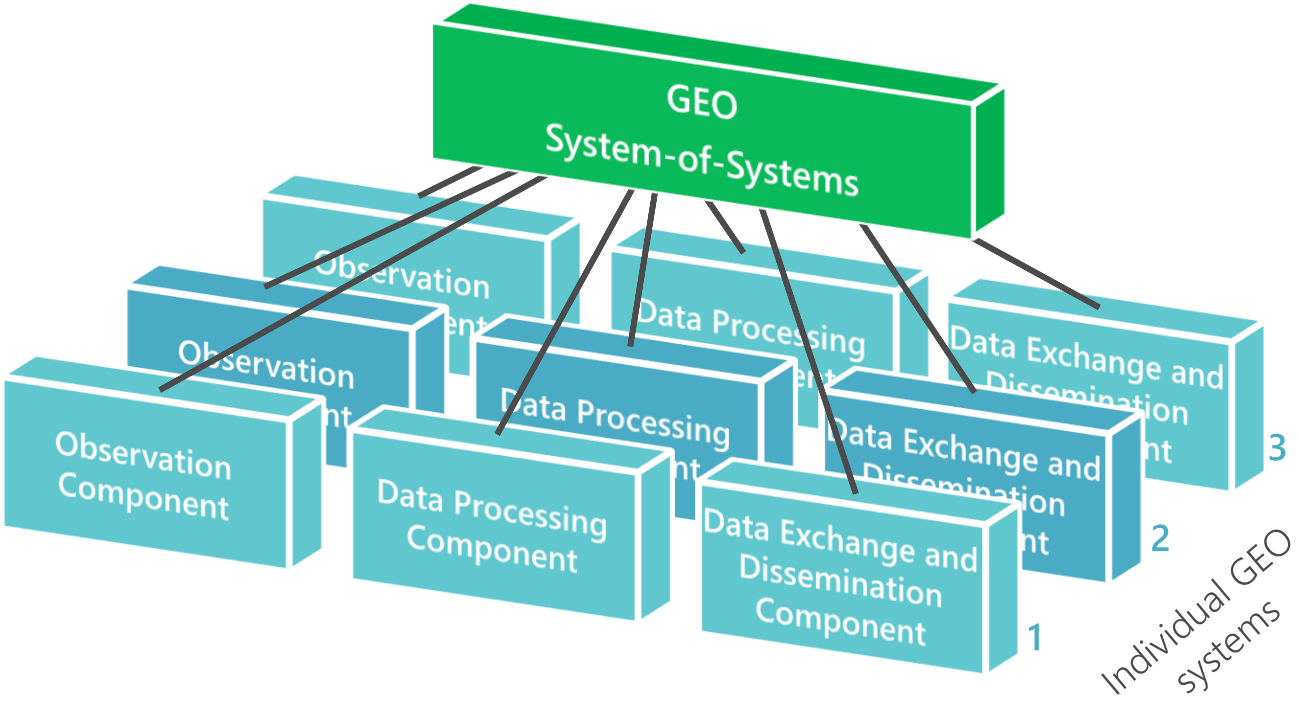
- Variety of users
- Various communities with their own cultures
- Distributed system
  - No new single architecture imposed to everyone
  - Preserve the existing infrastructures as much as possible
  - Enforce simple and robust interfaces and formats
- Dynamic, open system
  - Grow and attract third-party data and service providers and accepts intermitent participation with disconnected/connected modes without disruption
- Comprehensive information flow
  - End-to-end: product order, planning, acquisition, processing, archiving, and distribution

Source: http://earthobservations.org

SoSs Example

GEOSS

Architecture

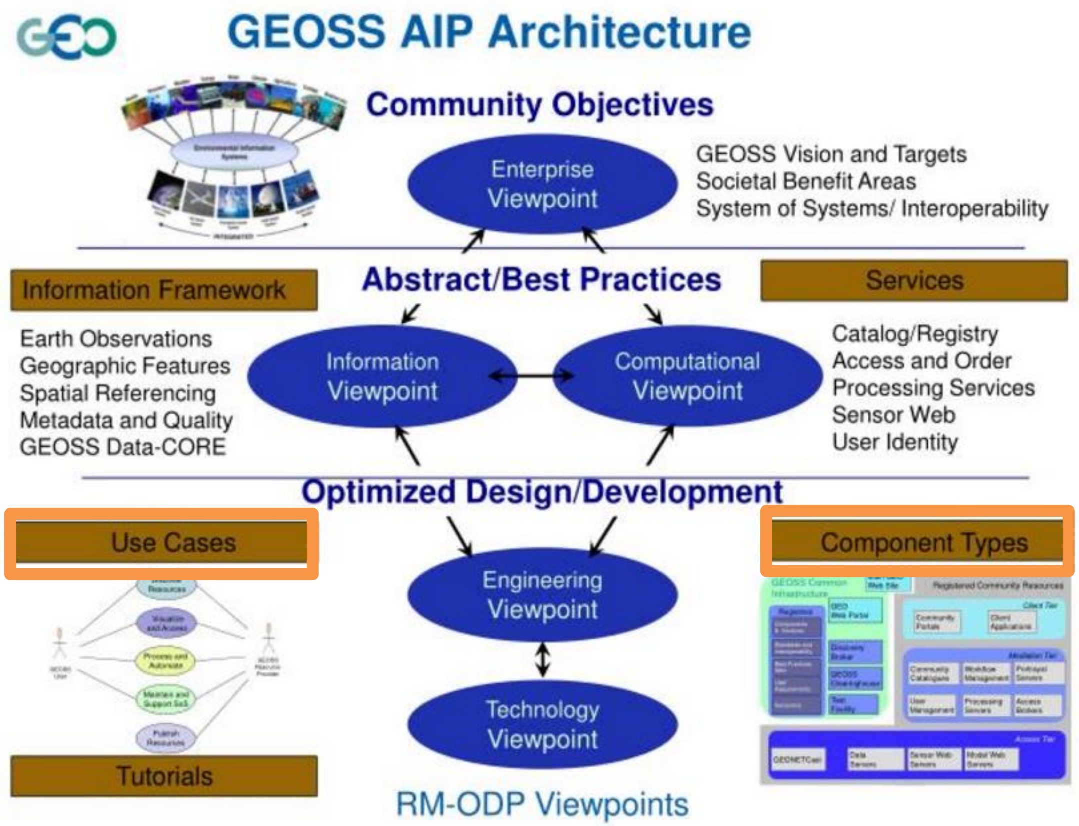

GEO
System-of-Systems
Observation Component
Observation Component
Observation Component
Data Processing Component
Data Processing Component
Data Processing Component
Data Exchange and Dissemination Component
Data Exchange and Dissemination Component
Data Exchange and Dissemination Component
3
2
1
Individual GEO systems

GEOSS defines best practices to ensure data integrability and interoperability

Source: http://earthobservations.org

SoSs Example

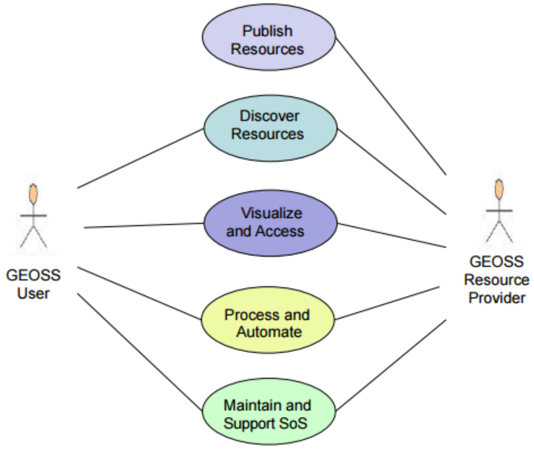
GEOSS Architecture Implementation Pilot (AIP)



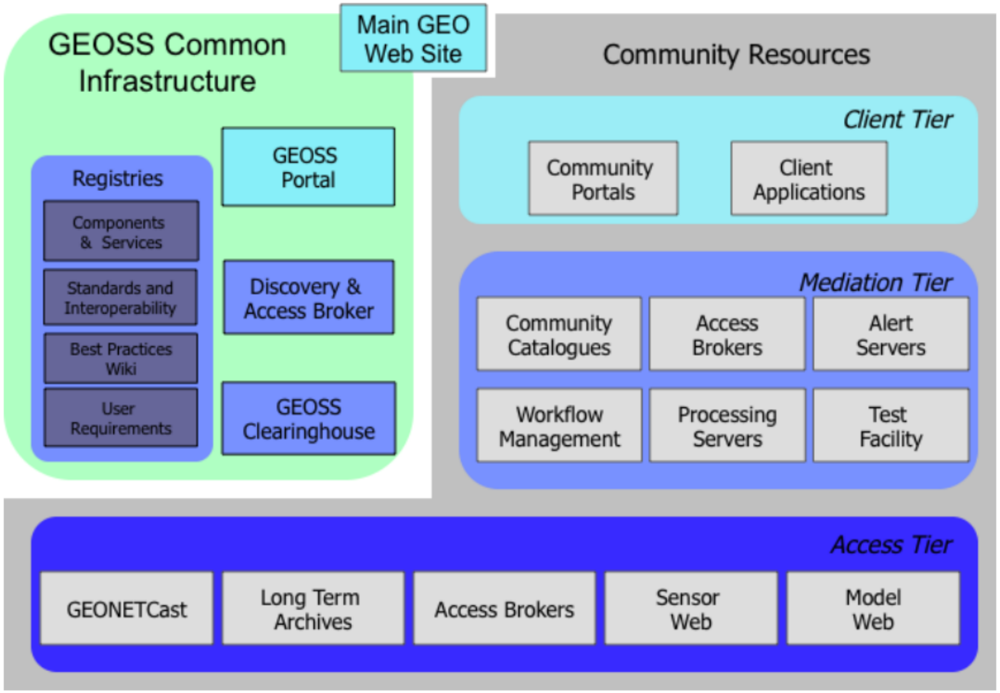
Source: http://earthobservations.org

SoSs Example

GEOSS Architecture Implementation Pilot (AIP)



Use Cases



Engineering components with services

Source: https://www.earthobservations.org/documents/cfp/201501_geoss_cfp_aip8_architecture.pdf

SoSs Example

GEOSS

- Interoperability through open interfaces and reference methods
  - Interoperability specifications agreed to among contributing systems
  - Access to data and information through service interfaces
- Open standards and intellectual property rights
  - Preference for formal international standards
  - Multiple software implementations compliant with the open standards should exist

Source: http://earthobservations.org

SoSs Example

GEOSS

- Build upon existing systems and historical data
  - National, regional or international agencies that subscribe to GEOSS but retain their ownership and operational responsability
- Implementation plan must address cost effectiveness, technical feasibility, and institutional feasibility
- To be sustained over a long period of time, GEOSS needs to be adjustable, flexible, adaptable, and responsive to changing needs
  - Capture future capabilities through open architecture

SOA is configurable and scalable to customer needs and leverages robust systems and processes for global interoperability

Source: http://earthobservations.org

SoSs Description

SoSs Description

- Two levels
  - Mission
    - Identifies required capabilities for constituents, operations, connections, emergent behavior, etc.
  - Architecture
    - Describes structure, behavior, and properties about the SoS

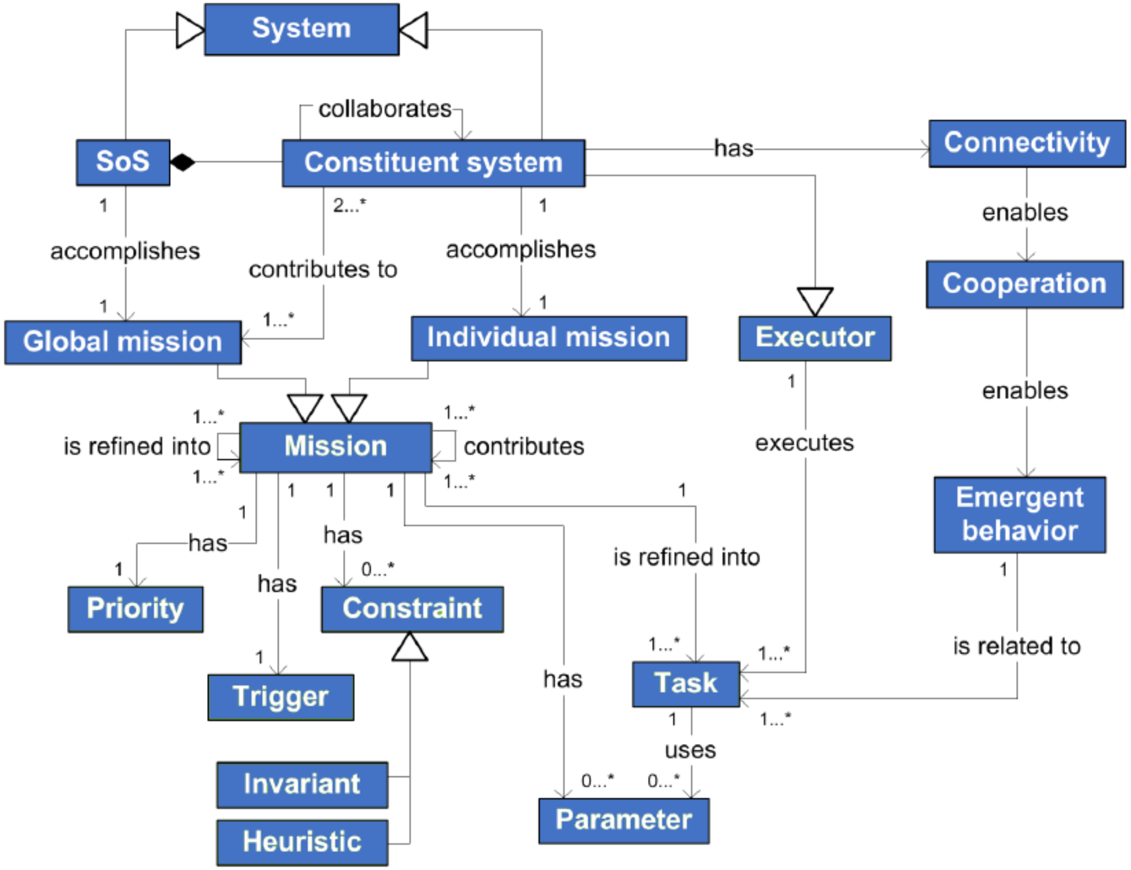
Mission

- Definition
    - Higher functionality that cannot be performed by any constituent alone
        - Accomplished by *emergent behaviors*
    - Guides the whole SoS development process
- mKAOS
    - Language for describing mission models
    - Tool: mKAOS Studio

Source: Silva, E. et al., 2015.

Mission

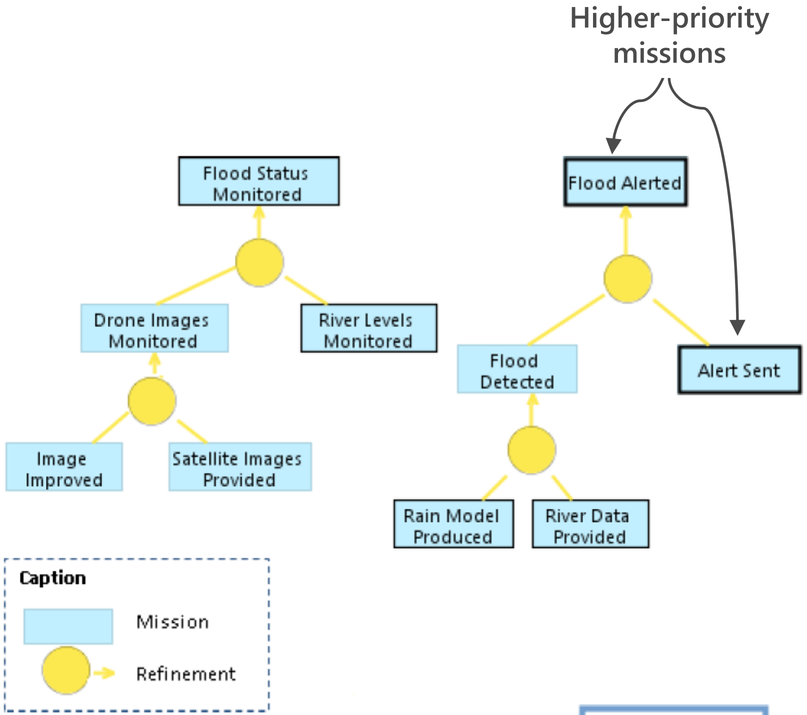
Conceptual model



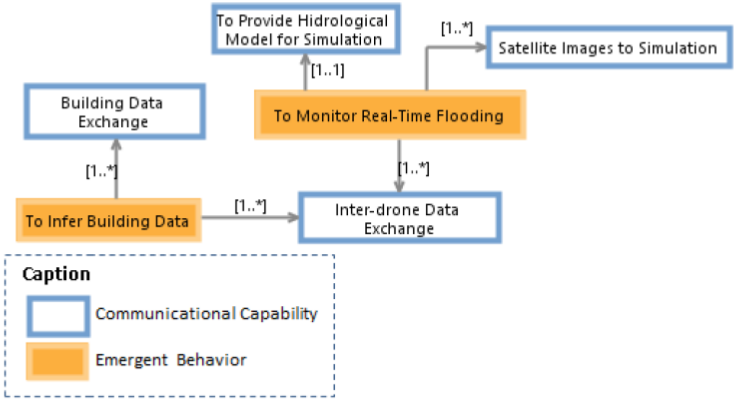
Source: Silva, E. et al., 2015.

Mission

Higher-priority missions



Mission model in mKAOS


To Provide Hidrological Model for Simulation
[1..*]
Satellite Images to Simulation
[1..1]
Building Data Exchange
To Monitor Real-Time Flooding
[1..*]
[1..*]
To Infer Building Data
[1..*]
Inter-drone Data Exchange
Caption
Communicational Capability
Emergent Behavior

Emergent behavior model in mKAOS

Source: Silva, E. et al., 2015.

SoSs Architectural Description

"To gain confidence that an SoS architecture will respect key properties, it is paramount to have a precise model of the constituents and the connectors between them, the properties of the constituents, and the SoSs environment."

Nielsen et al. (2015)

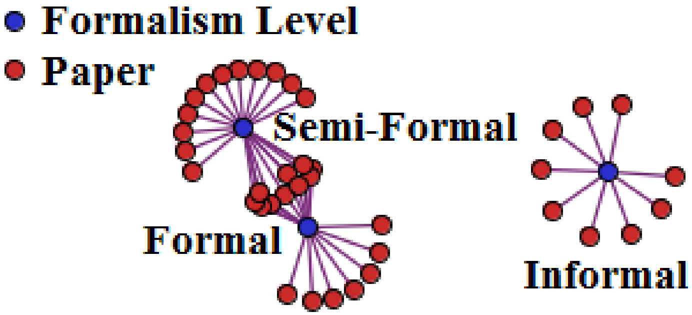
SoSs Architectural Description

- How has the literature addressed the architecture description of SoS?
- Which are the techniques used in the description of software architectures of SoS?
- Does the primary study focuses on a specific type of SoS?

Source: Guessi, M. et al. 2015a.

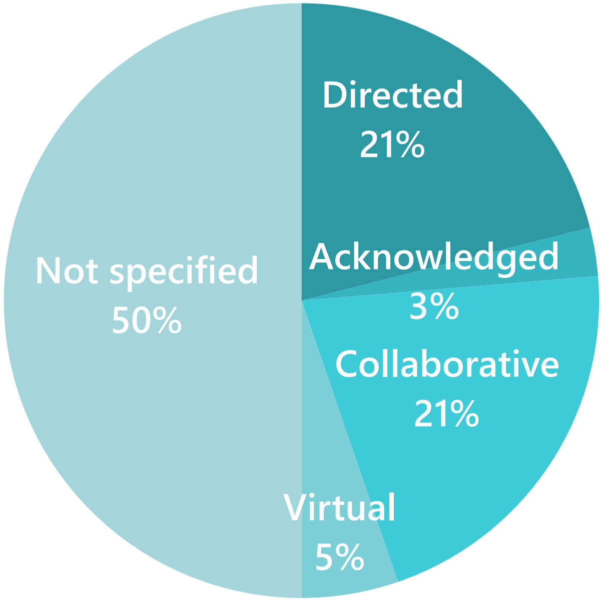
Techniques Used for Describing SoSs Architecture

- Formal languages:
  - **CML**, CFML, FSM, OWL, VDM-SL, among others
- Semi-formal languages:
  - UML, **SysML**, and UPDM
- Combination of formal and semi-formal languages:
  - UML/SysML + Petri nets
  - SysML + VDM-SL



Source: Guessi, M. et al. 2015a.

SoSs Type Described and Concerns



- Main quality characteristics:
  - Interoperability
  - Correctness
  - Integrability
  - Dependability
  - Adaptability
  - Safety

Source: Guessi, M. et al. 2015a.

ADLs for SoSs

SoS characteristics	Do Single System ADLs cope with SoS characteristics?
Operational independence of constituent systems	**No, they do not.** Single system ADLs are based on the notion that components' operation is totally controlled by the system, which is not the case for constituents. Moreover, the concrete components of single systems are known at design-time, which is not necessarily the case of SoSs either.
Managerial independence of constituent systems	**No, they do not.** Single system ADLs are based on the notion of components whose management is totally controlled by the system, which is not the case of SoSs.
Geographical distribution of constituent systems	**No, they do not.** Single system ADLs are based on the notion of logically distributed components. None supports the notion of physical mobility, in particular regarding unexpected local interactions among components that physically move near to each other, as it is the case of SoSs.
Evolutionary development of SoS	**No, they do not.** Single system ADLs are based on the principle that concrete components are known at design-time and that they may possibly enter or leave the system at run-time under the control of the system itself, which is not necessarily the case of SoSs.
Emergent behavior drawn from SoS	**No, they do not.** Single system ADLs have been defined based on the principle that all behaviors are explicitly defined (including global ones). None supports the notion of emergent behavior required in SoSs.

Source: Guessi, M. et al. 2015b. Oquendo, F. 2016a.

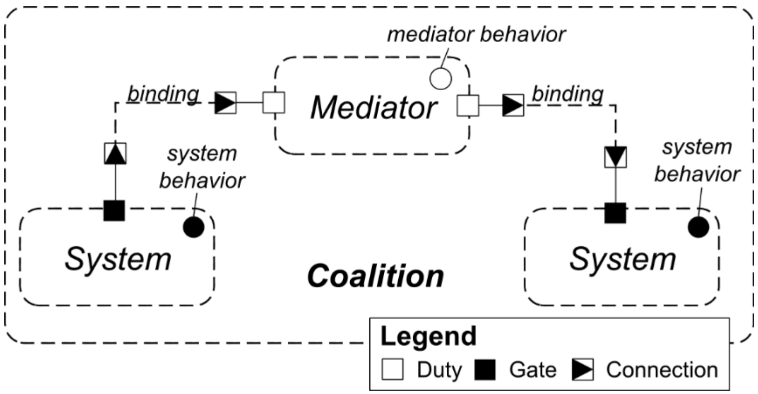
Single System ADLs Weaknesses for SoSs

- Limited expressive power in terms of on-the-fly evolution
- Lack support for open architecture description
  - Concrete constituents are not known at design-time
- Lack mechanism for describing emergent behaviors

Source: Guessi, M. et al. 2015b. Oquendo, F. 2016a.

SosADL
an Architecture Description Language for SoSs

- Description of an abstract architecture for SoS
  - It can be evolutionarily concretized at run-time by identifying and incorporating concrete constituent systems



Coalition represents on-the-fly composition of systems (i.e., constituents)

Source: Oquendo, F. 2016a. Oquendo, F. 2016b.

SoSs Architectural Description

- Analyze trade-offs of alternative designs at early development stages
- Describe contracts that exist between each constituent system and the SoS
- Support evolution
  - Important to keep the architectural design aligned with systems goals and technologies
  - Preserve specified properties under evolution steps
- Support dynamic reconfiguration
  - Run-time modification of architectures and interfaces
- Support emergent behaviors
  - Describe global properties at the SoS level
  - Enable statement and verification of emergence (including desirable and undesirable)

SoSs Research Directions

Research Directions

- Formal ADLs for SoSs
  - Promote correctness, consistency, and completeness of architecture descriptions
  - Support evolutionary development of SoSs
- Desired properties of ADLs for SoSs
  - Understandability,
  - Scalability,
  - Refinement,
  - Traceability, among others others
- Support different phases of SoS life cycle
  - Enforce correctness, consistency, and understandability of architecture descriptions
  - Ensure semantic consistency among heterogeneous models of constituents
  - Interchangeable, complementary techniques should be explored for supporting different abstraction/formalism levels

Bibliography Part I

- Bass, L., Clements, P., and Kazman, R. 2003. Software Architecture in Practice (2ed.). Addison-Wesley Longman Publishing Co.
- Gorton, I. 2006. Essential Software Architecture. Springer-Verlag New York, Inc.
- Kruchten, P. What do software architects really do? In: Journal of Systems and Software, v.81, p.2413-2416. 2008
- Hofmeister, C., Kruchten, P., Nord, R. L., Obbink, H., Ran, A. and America, P. A general model of software architecture design derived from five industrial approaches. In: Journal of Systems and Software, v.80, n.1, p. 106-126. 2007.
- Garland, J. and Anthony, R. 2003. Large-Scale Software Architecture: A Practical Guide Using UML. John Wiley & Sons, Inc., New York, NY, USA.Hofmeister
- ISO/IEC/IEEE 42010:2010 International Standard for Systems and Software Engineering -- Architectural description
- Malavolta, I.; Lago, P.; Muccini, H.; Pelliccione, P. and Tang, A. What Industry Needs from Architectural Languages: A Survey IEEE Transactions on Software Engineering, 2013, v. 39, n. 6, 869-891.
- Lago, P.; Malavolta, I.; Muccini, H.; Pelliccione, P. and Tang, A. The road ahead for architectural languages. IEEE Software, 2014, 32, 98-105.
- Medvidovic, N. and Taylor, R. N. A classification and comparison framework for software architecture description languages. In: IEEE Transactions on Software Engineering, 2000, v. 26, n.1, 70-93.
- Oquendo, F. pi-ADL: An Architecture Description Language based on the Higher Order Typed pi-Calculus for Specifying Dynamic and Mobile Software Architectures. In: ACM Software Engineering Notes, 2004, v. 29, n.3, 15-28.
- Clements, P.; Bachmann, F.; Bass, L.; Garlan, D.; Ivers, J.; Little, R.; Merson, P.; Nord, R.; and Stafford, J. Documenting Software Architectures: Views and Beyond. Addison-Wesley, 2011.
- Shaw, M. and Garlan, D. Characteristics of Higher-Level Languages for Software Architecture. Carnegie Mellon University, 1994. http://www.sei.cmu.edu/reports/94tr023.pdf

Bibliography Part II

- Boehm, B.; Brown, W.; Basili, V. & Turner, R. Spiral Acquisition of Software-Intensive Systems-of-Systems. In: Crosstalk, 2004, p. 4-9
- Guessi, M.; Neto, V. V. G.; Bianchi, T.; Felizardo, K. R.; Oquendo, F. & Nakagawa, E. Y. A systematic literature review on the description of software architectures for systems of systems. In: ACM/SIGAPP SAC' 2015, 2015a, p. 1442-1449
- Guessi, M., Cavalcante, E., and Bueno, L.B.R. Characterizing ADLs for Software-Intensive SoS. In: SeSoS at ICSE' 2015. 2015b. p. 12-18.
- Medvidovic, N. and Taylor, R. N. A classification and comparison framework for software architecture description languages. In: IEEE Transactions on Software Engineering, 2000, v. 26, n.1, 70-93.
- Nielsen, C. B.; Larsen, P. G.; Fitzgerald, J.; Woodcock, J. & Peleska, J. Systems of Systems Engineering: Basic Concepts, Model-Based Techniques, and Research Directions. In: ACM Comput. Surv., 2015, v. 48, p. 1-41
- Oquendo, F. Formally Describing the Software Architecture of Systems-of-Systems with SosADL. In: SoSE' 2016, 2016a, p.1-6
- Oquendo, F. $-Calculus for SoS: A Foundation for Formally Describing Software-intensive Systems-of-Systems. In: SoSE' 2016, 2016b, p. 1-6
- Silva, E.; Batista, T. & Oquendo, F. A Mission-Oriented Approach for Designing System-of-Systems. In: SoSE' 2015, p. 346-351.
- Ulieru, M. & Doursat, R. Emergent engineering: a radical paradigm shift. In: Int. J. Autonomous and Adaptive Communications Systems, 2011, v. 4, n.1, p. 39-60.