Software Architectures and Quality

Documenting Software Architectures

Javier González Huerta

javier.gonzalez.huerta@bth.se

Objectives

- Discuss the concept of architectural documentation, and architectural views
- Introduce the most common architecture view models

Documenting Software Architectures

Importance of Software Architecture Documentation

Importance of Software Architecture Documentation

- Understanding and educating
- Communicating
- Vertebrate Discussion
- Trade-Off Support
- Support for Evaluation and Analysis

Software Architecture Documentation

- Abstract enough to understand
- Detailed enough to analyze
- Prescribes constraints
- Recounts decisions
- Fulfills different stakeholder needs

Roles and Interests

Role	Use of the SA Documentation
Architect	Negotiation, Trade-offs between competing requirements and design solutions
Developer	Understand implementation rules and constraints Understand division of work
Project Manager	Budget and Schedule Resource allocation
Maintainer	Understand the system Impact analysis
Tester	Basis for tests specification based on the behavior and interactions between software elements
Customer	Understanding about the system and the development process
Evaluator	Evaluate the conformance with regard to the quality requirements

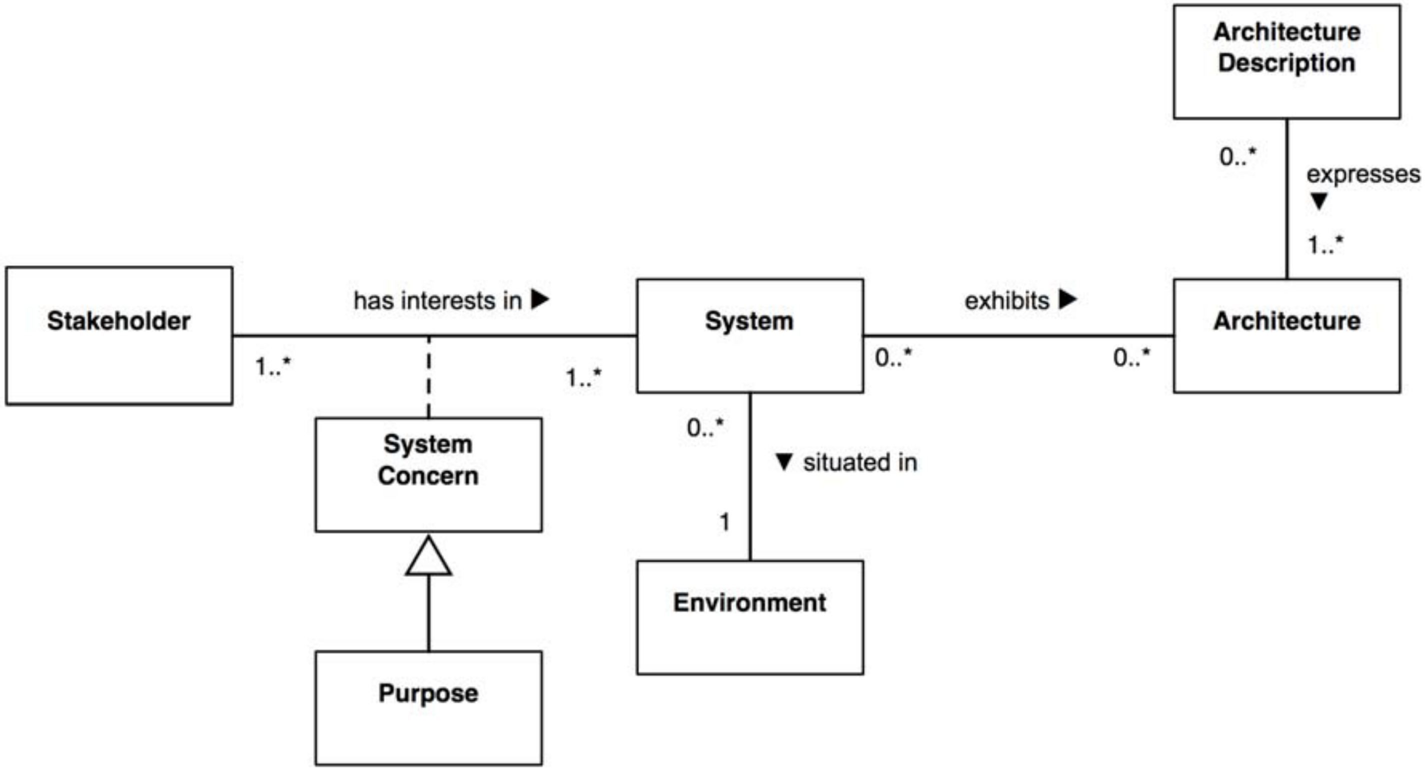
Documenting Software Architectures

What is an Architectural View?

Architectural Views

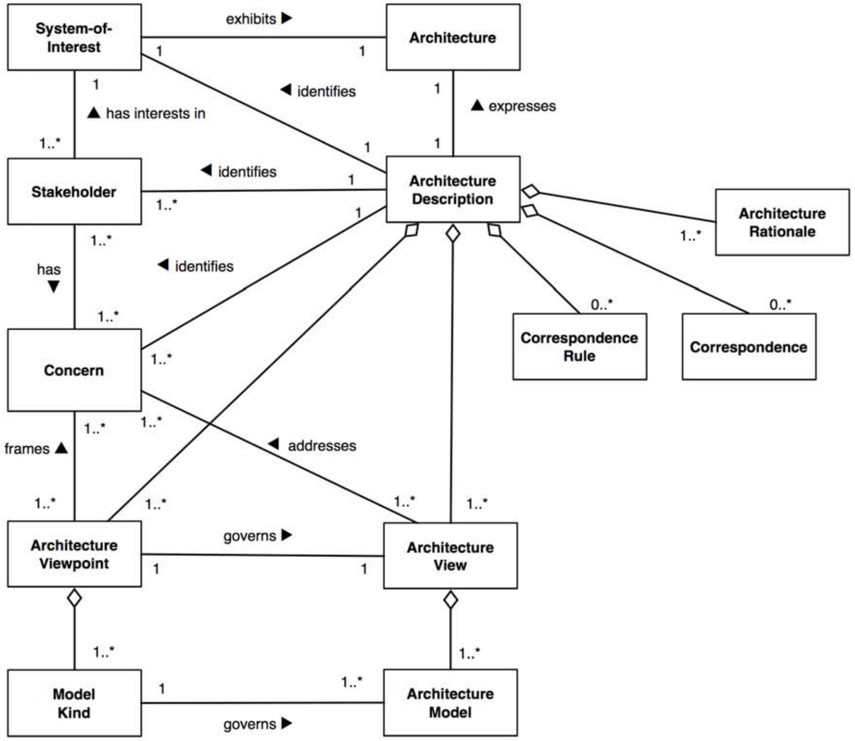
- *A software architecture is a complex entity that cannot be described in an unidimensional fashion*
- The software architecture of a software system comprises many different concerns
  - Development
  - Testing / Quality
  - Deployment
  - Maintenance
- Not all the architectural / system information is going to be applicable to one concern
- We simplify the showing only what is relevant to each concern
- Divide and conquer

Terminology: Concerns and Views



ISO / IEC / IEEE , “ISO / IEC / IEEE 42010:2011 Systems and software engineering,” 2011.

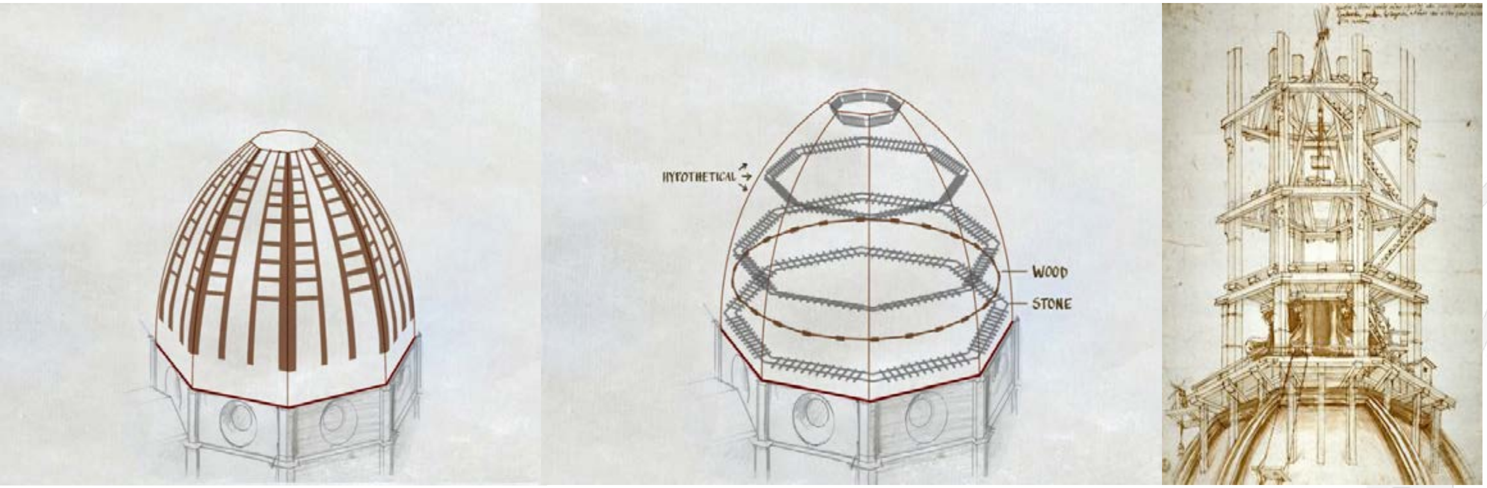
Terminology: Views and Models



ISO / IEC / IEEE , "ISO / IEC / IEEE 42010:2011 Systems and software engineering," 2011.

Architectural Views

- No single view is the architecture, all together form the architectural description of the system
  - Each view describes a certain concern of the system

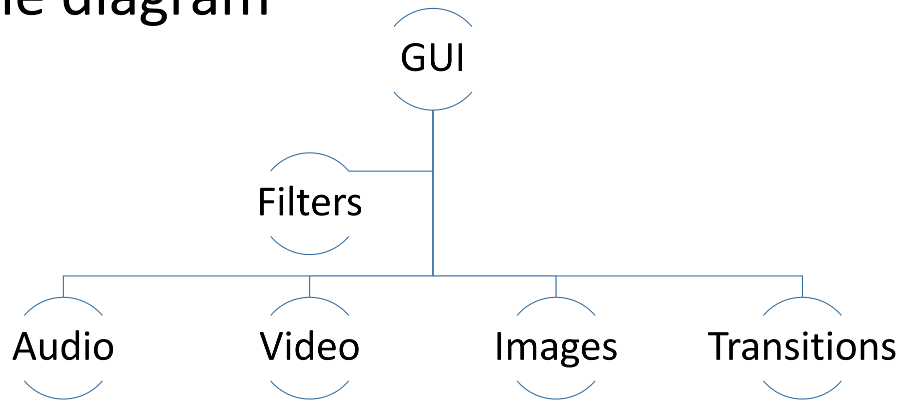


What is a view?

- A view has:
  - Elements
  - Relationships among elements
  - Properties of elements and / or relations
  - Constraints
  - A selection criterion which specifies the elements, relations and properties we consider and the ones we exclude
- The selection criteria should be clear and unambiguous

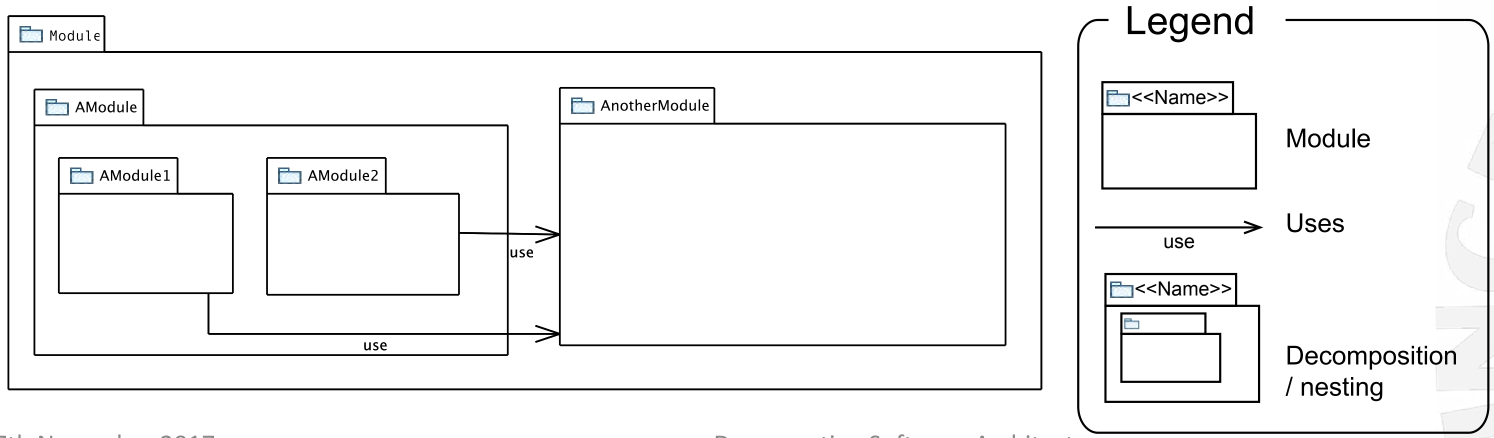
What is a view? Beyond Boxes and Lines

- To be useful, every view in the architecture documentation have to include a key / legend that describes the meaning of all the symbols shown
  - In case of pure text views, an explanatory text and example can help as key / legend
- Adding the legend to the view provides a clear and comprehensible semantics to the diagram



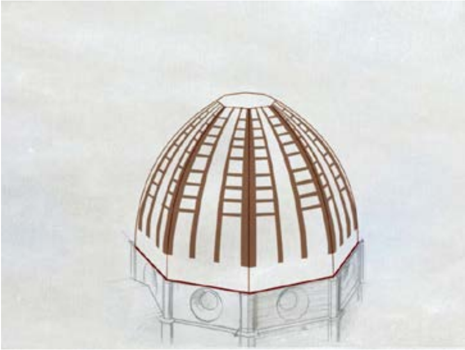
Example of a view diagram

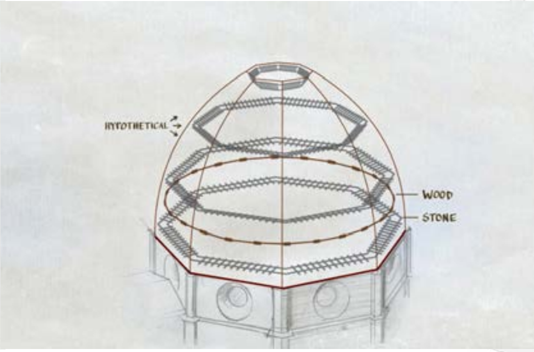
- Elements: Modules
- Relationships: Uses, Decomposition
- Properties: Module Name
- Selection Criteria: All modules on the 3 first levels

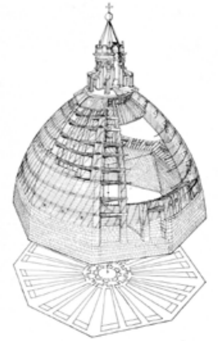


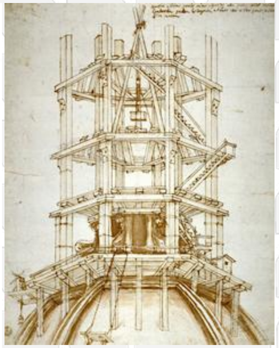
Projection

- A software architecture view / model can be seen as a projection of the architecture using as a filter the view definition
- We can make the analogy with the construction drawing view
  - It is a projection of all the abstractions needed to implement the system (in that case build the house)
- For example the electricity schemas is not usually drawn together with the plumbing (only if you need to make specific analysis)







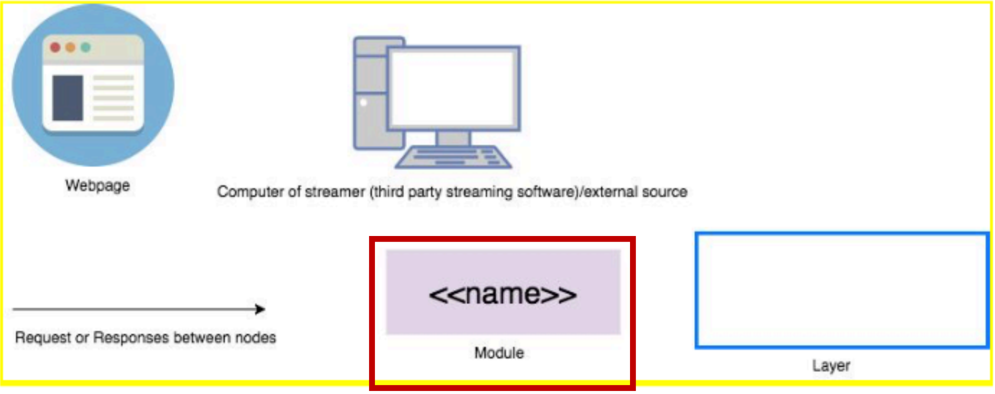


Consistency: Legend - View Definition

- A Legend is consistent with the view definition when:
  - All elements / relationships / properties shown on the legend are defined in the view definition
  - NO additional elements, relationships, properties that are not described in the view definition appear on the legend

Consistency: Legend - View DefinitionCounter Example

- **Elements:** Webpage, External Source(Ex. Computer of streamer), Layer
- **Relations:** Request or Response
- **Property:** Name

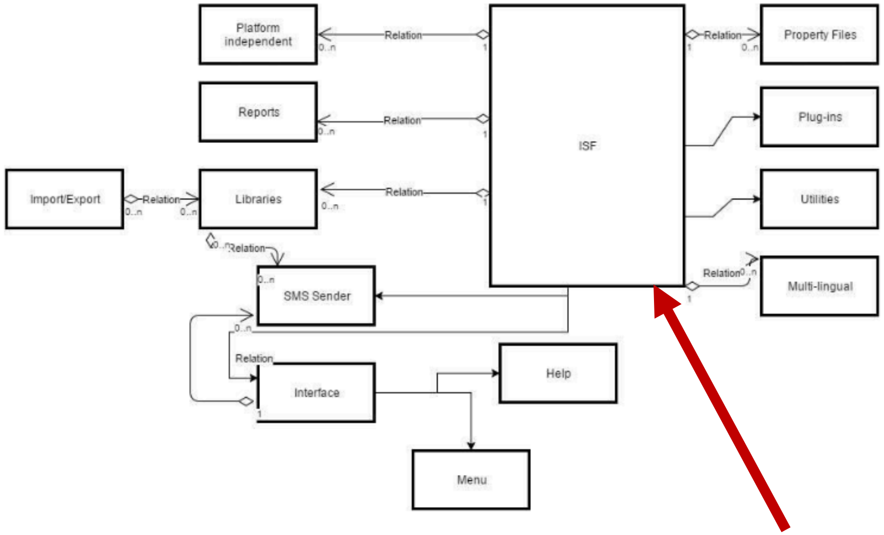


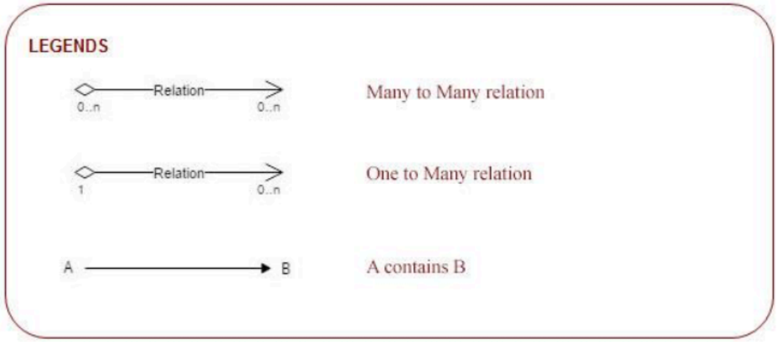
Webpage
Computer of streamer (third party streaming software)/external source
Request or Responses between nodes
<<name>>
Module
Layer

Consistency: View - Legend

- A View is consistent with the Legend:
  - All elements / relationships / properties shown on the view match with their corresponding graphical notation described in the legend
  - NO additional elements, relationships, properties that have no corresponding graphical notation or that have a appear on the view

Consistency: View – Legend - Counter Example



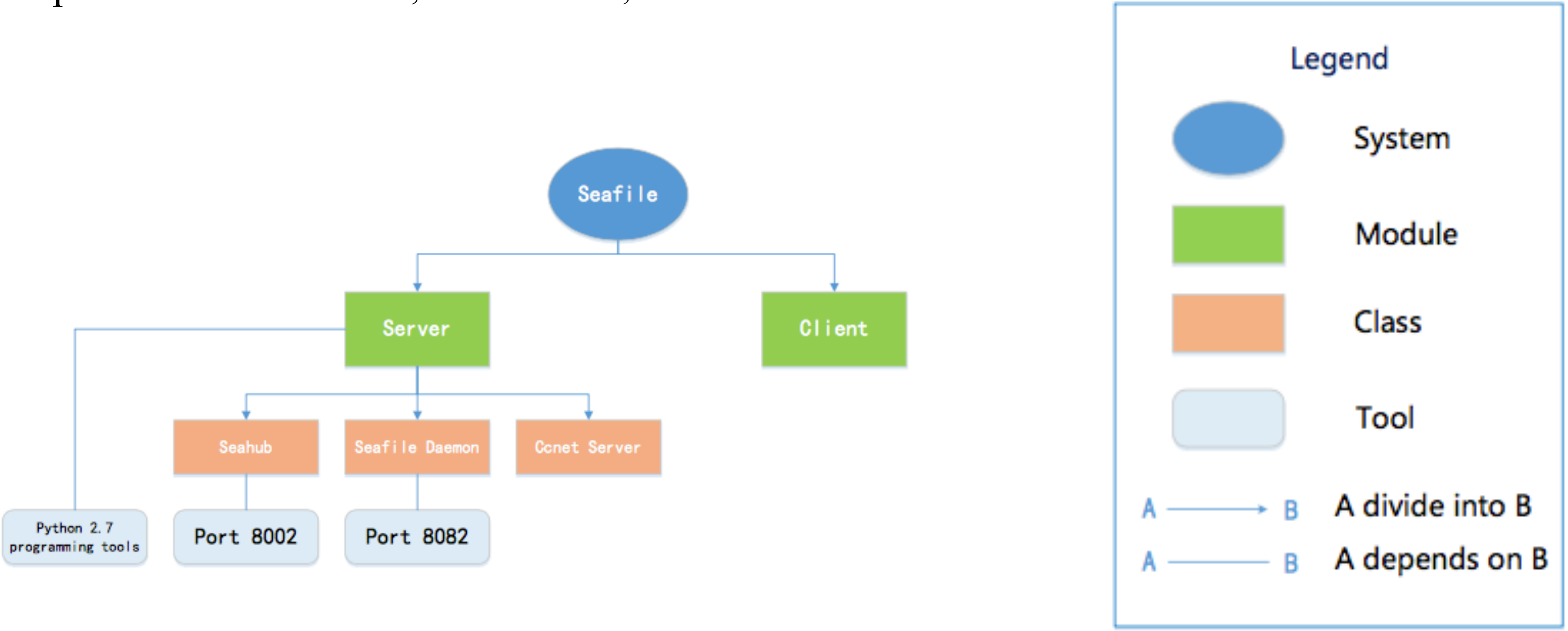


Consistency: View – View Definition

- A View is consistent with its definition when:
  - All elements that fulfill the selection criteria are shown on the view
  - All relations that fulfill the selection criteria are shown on the view
  - All properties that fulfill the selection criterion are shown on the view
  - NO additional elements, relationships, properties that do not meet the selection criteria are shown on the view

Consistency: View – View Definition - Counter Example

Elements: Module
Relations: include, dependency
Properties: Module’s name, Class’s name, Tool’s name



Counter Examples

- These are anti-patterns (bad solutions to common recurring problems)
- This is not what you are expected to do in your assignments: Don’t reinvent the wheel, if you know there are good representations for views, please use them!

Documenting Software Architectures

Architectural Views

Typical Views Set

- Conceptual View
- Modular View [1]
- Component View [1]
- Allocation View [1]

[1] P. C. Clements *et al.*, *Documenting software architectures: views and beyond*, 2nd Edition. Pearson Education, 2010

Note: The information is also summarized in the course book, chapter 18.

Conceptual View

- The conceptual view describes the system in terms of domain-level concepts and abstractions
- Is a coarse grained description of the system at hand
- It should be relatively independent of particular software and hardware solution
- **Elements:** Concepts / Roles / Classes
- **Relationships:** Association / Composition / Aggregation
- **Properties:** Name, Responsibilities

Module Views

- A module is an implementation unit (e.g., Classes + Interfaces)
  - Provides a coherent set of responsibilities.
- **Elements:** Software structures (C programs, C++, Java or C# classes) and their grouping (Packages or Namespaces)
- **Relationships:**
  - Is Part of: Describes the part vs whole relationship between the module and *submodule* the part
  - Depends on: Dependency relationship between two elements. *Uses, Allowed to Use* are examples of depend on relationship
  - Is a: Generalization / Specialization

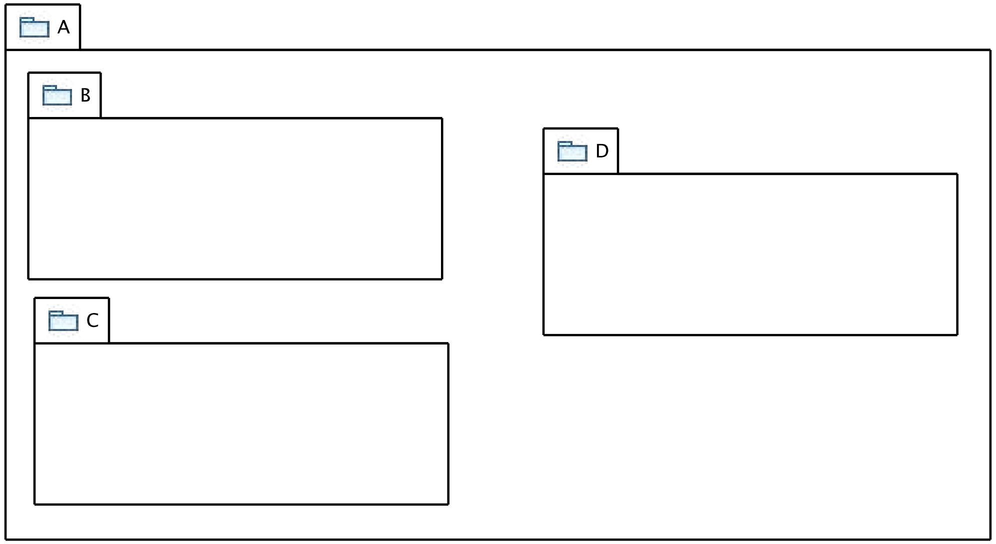
Properties in Module Views

- Name
- Responsibilities
- Visibility/scope of interfaces
- Implementation details
  - Language,
  - Mapping to source code artifact
  - Test info
  - Author
  - Revision history

Styles of Module View

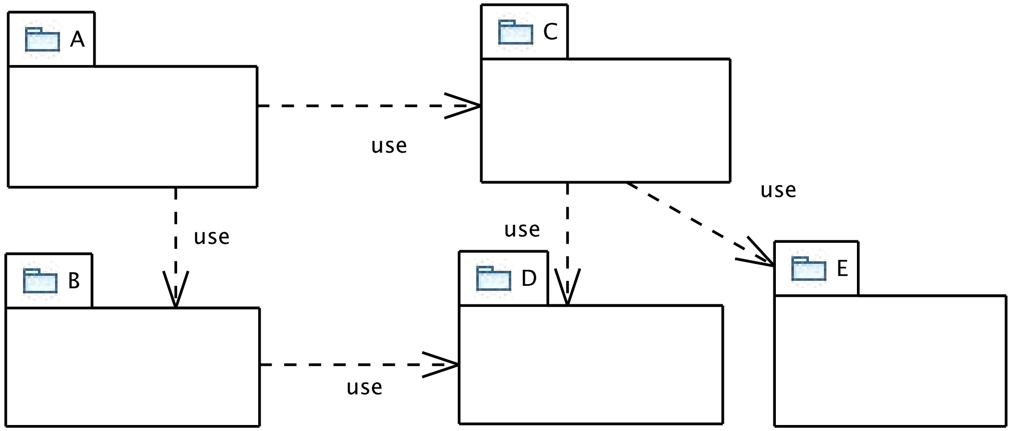
- Decomposition
- Uses
- Layers
- …

Module View: Decomposition



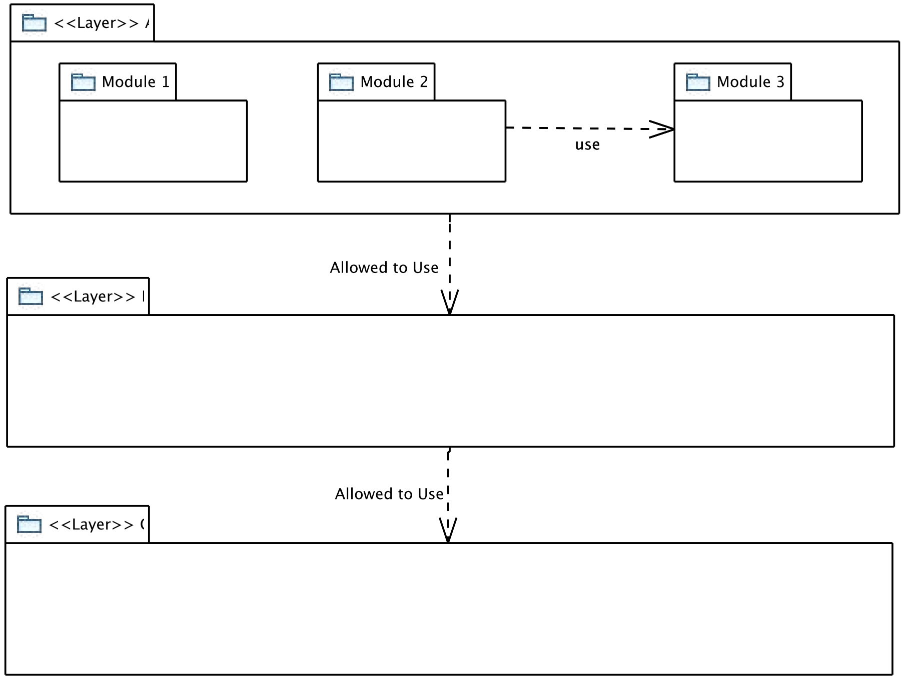
- Focuses on the *is_part_of* relationship
- Describes how the system is organized into submodules
  - How the responsibilities are partitioned across them
- Divide and Conquer

Module View: Uses



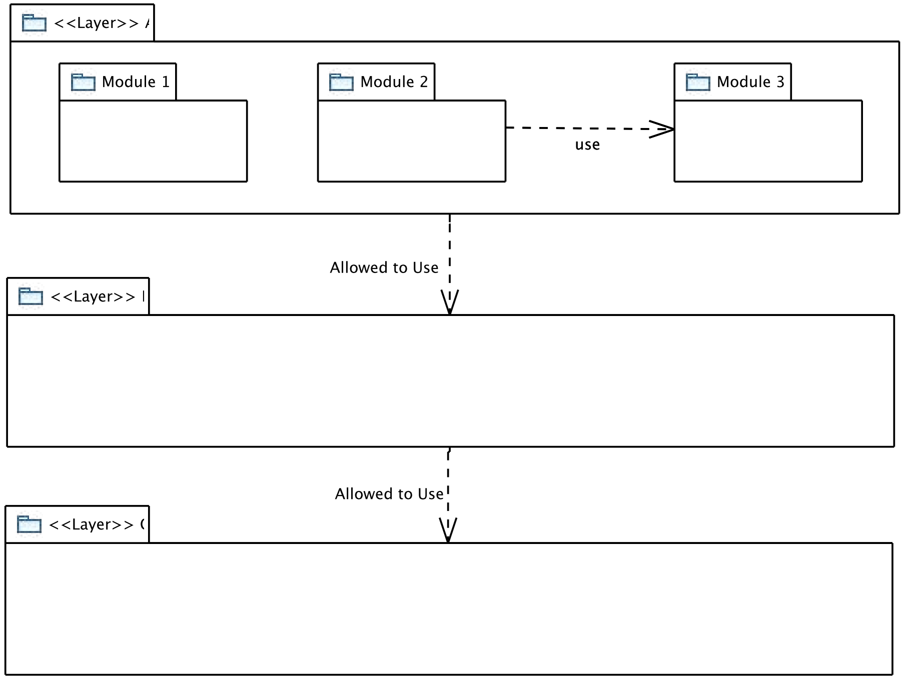
- This style describes the build dependencies
- Focuses on the *depends_on* relationship
- A module uses B module if it requires part of its exposed functionality
- A´s correctness depends_on B´s correctness

Module View: Layers



- A layer is a grouping of modules that together offer a set of services to other layers
- Layers completely partition the system and each partition, through interfaces

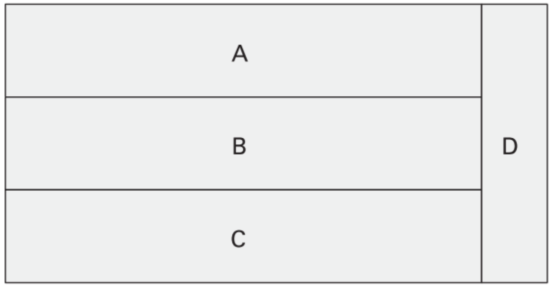
Module View: Layers

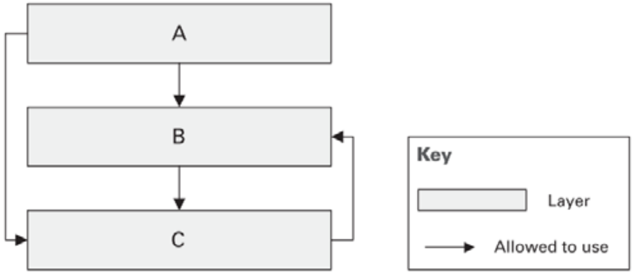


- The main relationship is the *allowed_to_use*
  - Strictly ordered
  - Unidirectional

Reflection

- **Discuss during 2 minutes in groups of 2-3 persons: Are these two examples of layered architectures?**





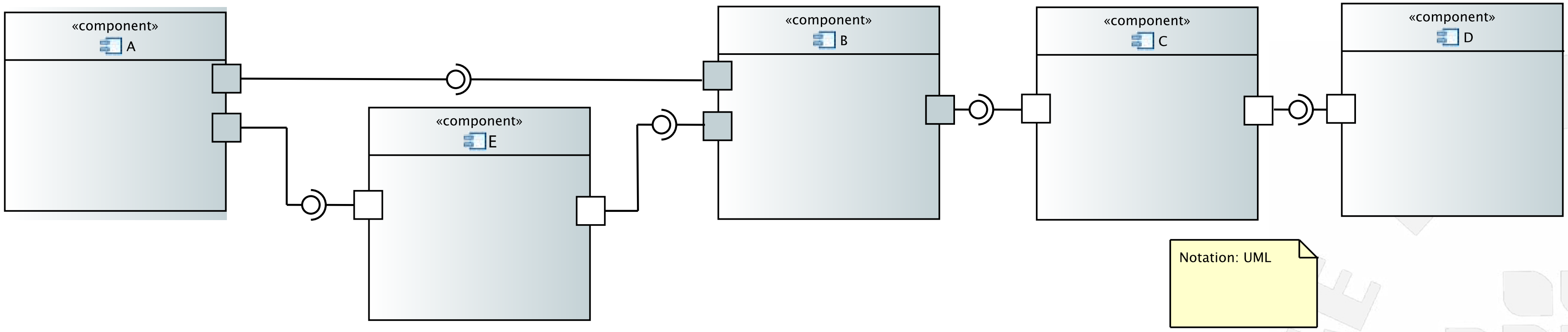
Component and Connector Views

- Describe the system in terms of runtime software elements (typically threads and processes, but also objects and data stores)
- Component have interfaces called ports that define their interaction with their environment
- Ports of the same type can be replicated to offer different input or output channels at runtime

Component and Connector Views

- **Elements:**
  - Components: Runtime elements as Processes and / or Threads
  - Connectors: Are defined as the forms of interaction between components (synchronous, asynchronous, complex transactions)
- **Relations:** The relations are pathways for interaction and communication. Usually the *Attachment* between a component and a connector
- **Properties:** Execution related properties
  - Name
  - Latencies
  - Scheduling
  - Access Rights
  - Concurrence

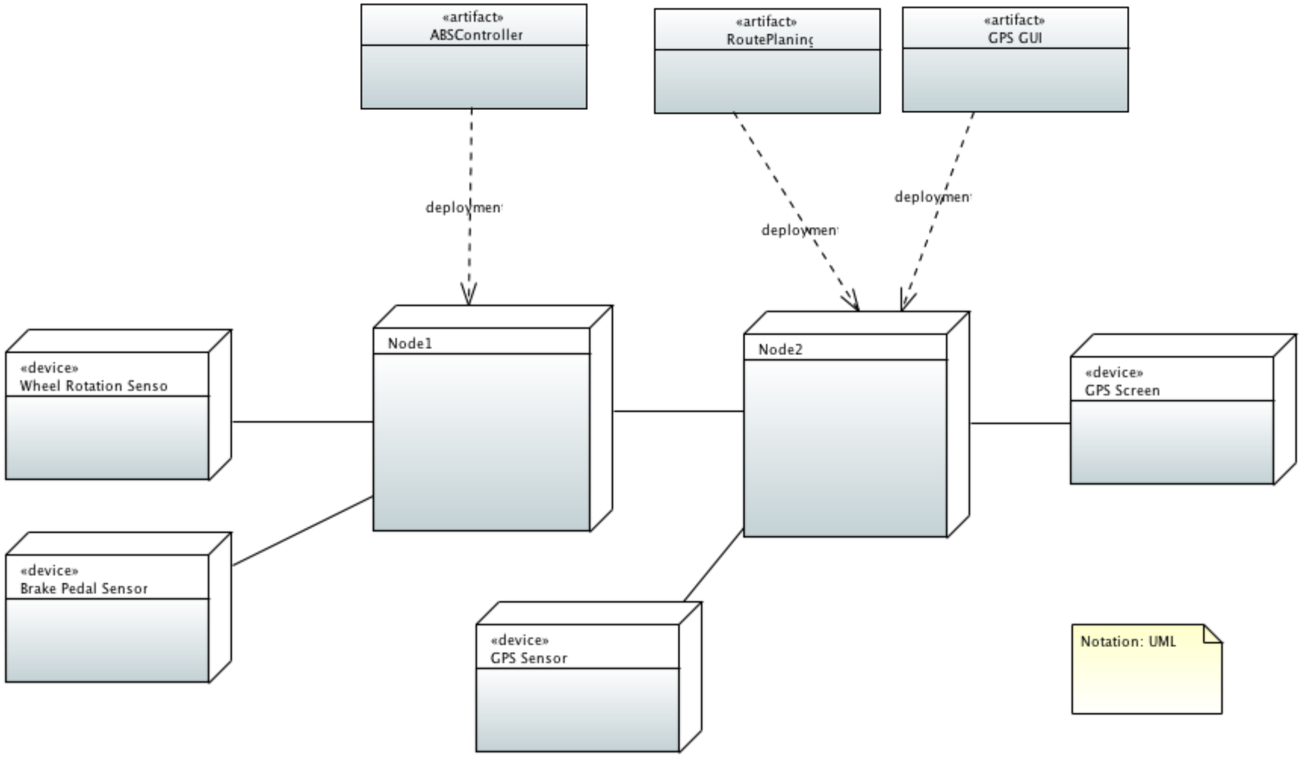
Component and Connector Views



Allocation Views: Deployment View

- Describe the mapping of software elements into non-software (typically hardware) elements
- The most typical allocation view is the *deployment view*
- **Elements:**
  - Software Elements: elements from the C&C View
  - Environmental Elements: Hardware of the computing platforms
- **Relationships:**
  - Allocated-to: Describes the physical units where the software elements will be executed
  - Migrates-to, Copy-migrates-to or Execution-migrates-to to express migration tactics in case of failure / need of backup

Allocation View



Notation: UML

Mapping between Views

- The different views have a direct mapping among them (although not always explicit)
- The conceptual view shows how the application is organized into smaller solutions (modules)
- The component view describes the main processes, but the mapping does not necessarily have to be 1 to 1
- The allocation view describes how the processes and threads of the component view are mapped into hardware elements (in this case the mapping is explicit

Reflection

- **Discuss during 2 minutes in groups of 2-3 persons: What are the most relevant views? How many views are ”good enough” to have?**

Which are the relevant views?

- This totally depends:
  - On the System nature
  - The goal you pursue when documenting the architecture
- Different views expose different quality attributes to different extent
- How many? Occam´s razor – only the necessary ones
  - Less and updated is better than many obsolete and that nobody cares to read

Qualities and Views

- To analyze software architecture for specific qualities we need the information in the views
- Some views are more appropriate to a certain quality than others
- E.g. Latency is often related to networking and execution and therefore more relevant information is found in component and deployment views than in the module view.

Documenting Software Architectures

Documenting the Views

Documenting the views

- The views can be documented using three categories of notations (taking into account their degree of formality):
  - Informal notations (boxes and lines)
  - Semi-formal notations (like UML or SysML)
  - Formal notations using formal languages or Architectural Description Languages

Informal Notations (boxes and lines)

- Views are described by using informal graphical notations
  - Without any sort of syntactic rules to draw the diagrams
  - Without clear semantics
- The visual conventions should be defined while creating the views
- Since there is no semantics no conformance checking can be performed
- If any form of analysis is required, should be done manually, since the views are not machine-readable

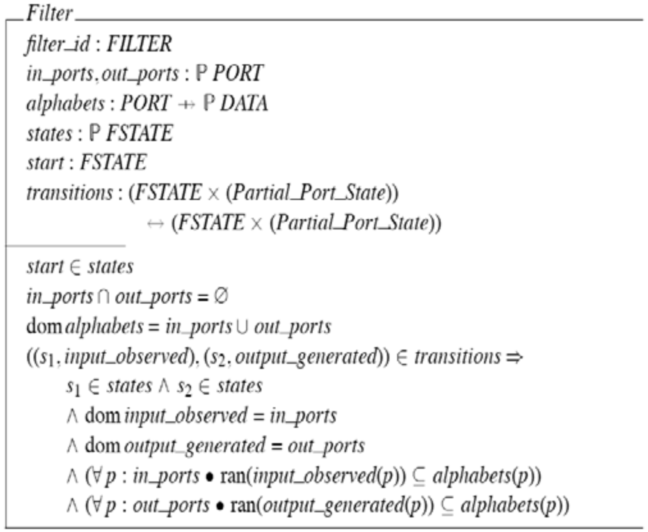
Use of Structured, Semi-Formal Notations

- Views described by using structured, sometimes standard notations like UML or SysML (Systems Modeling Language)
- UML is a standardized modeling language for modeling software designs
  - It can be used to document software architectures but some object-oriented abstractions are not well-suited to describe architectural concerns
- SysML is neither a Architecture Description Language but offers the required abstractions to describe software architectures
- Conformance checks can be automatically to check the consistency with the languages

Formal Notations

- Mathematical specifications (Z, VDM)

A system using Pipes & Filters in Z



Filter

$filter\_id : FILTER$

$in\_ports, out\_ports : \mathbb{P}\ PORT$

$alphabets : PORT \rightharpoonup \mathbb{P}\ DATA$

$states : \mathbb{P}\ FSTATE$

$start : FSTATE$

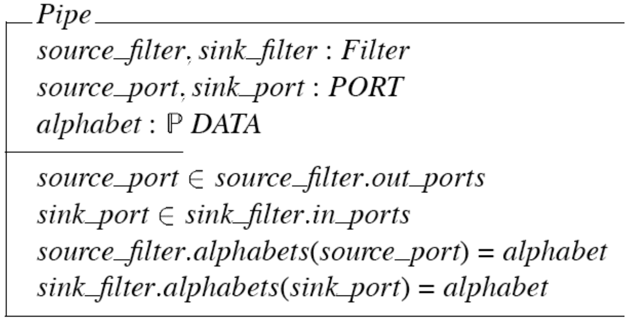
$$transitions : (FSTATE \times (Partial\_Port\_State)) \\ \leftrightarrow (FSTATE \times (Partial\_Port\_State))$$

$start \in states$

$in\_ports \cap out\_ports = \emptyset$

$\mathrm{dom}\, alphabets = in\_ports \cup out\_ports$

$$\begin{aligned}&((s_1, input\_observed), (s_2, output\_generated)) \in transitions \Rightarrow \\ &\quad s_1 \in states \land s_2 \in states \\ &\quad \land\ \mathrm{dom}\, input\_observed = in\_ports \\ &\quad \land\ \mathrm{dom}\, output\_generated = out\_ports \\ &\quad \land\ (\forall p : in\_ports \bullet \mathrm{ran}(input\_observed(p)) \subseteq alphabets(p)) \\ &\quad \land\ (\forall p : out\_ports \bullet \mathrm{ran}(output\_generated(p)) \subseteq alphabets(p))\end{aligned}$$



Pipe

$source\_filter, sink\_filter : Filter$

$source\_port, sink\_port : PORT$

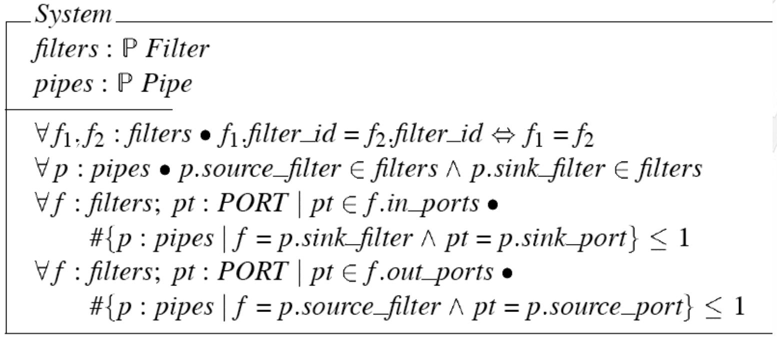
$alphabet : \mathbb{P}\ DATA$

$source\_port \in source\_filter.out\_ports$

$sink\_port \in sink\_filter.in\_ports$

$source\_filter.alphabets(source\_port) = alphabet$

$sink\_filter.alphabets(sink\_port) = alphabet$



System

$filters : \mathbb{P}\ Filter$

$pipes : \mathbb{P}\ Pipe$

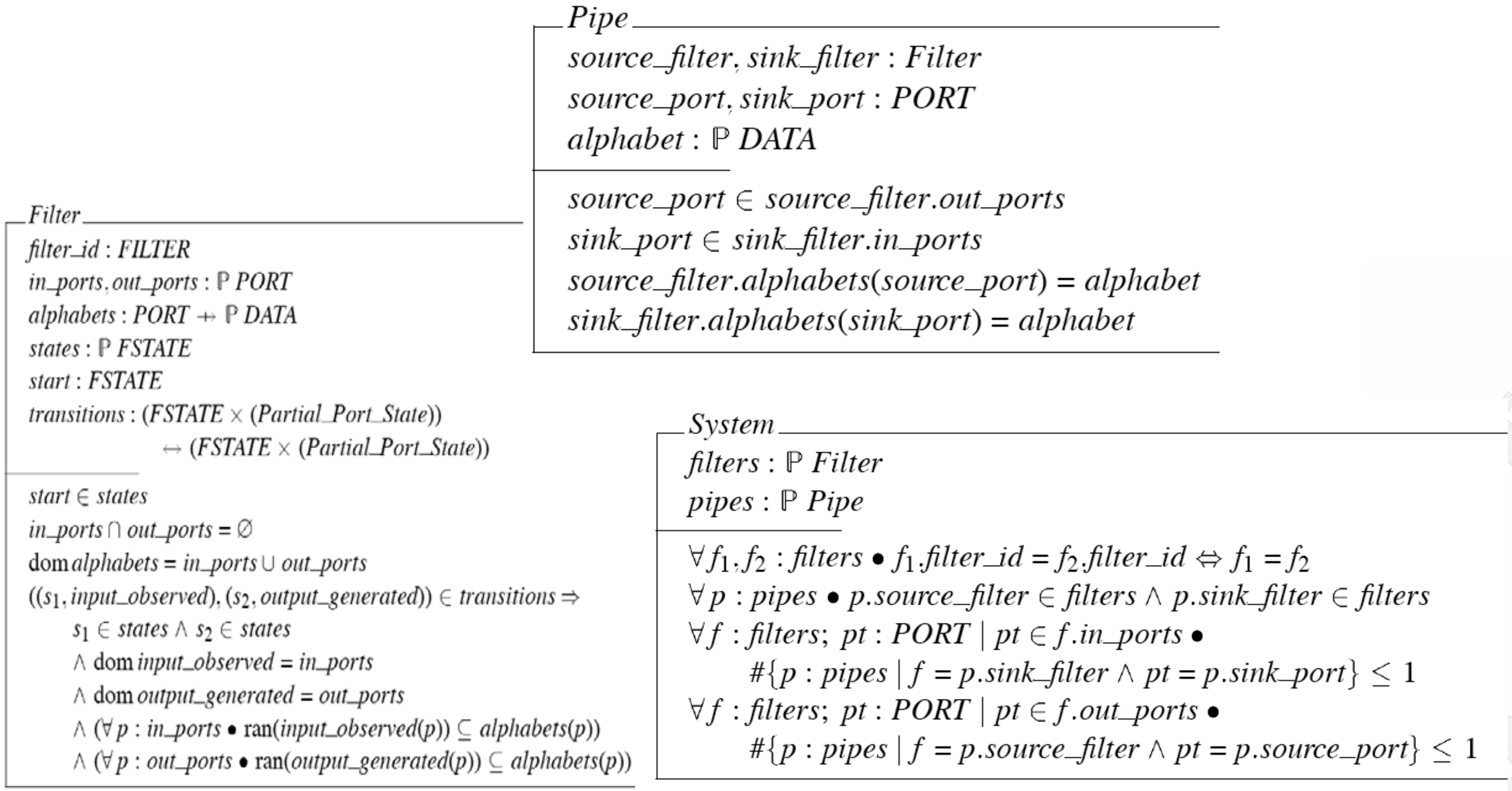
$\forall f_1, f_2 : filters \bullet f_1.filter\_id = f_2.filter\_id \Leftrightarrow f_1 = f_2$

$\forall p : pipes \bullet p.source\_filter \in filters \land p.sink\_filter \in filters$

$$\begin{aligned}&\forall f : filters;\ pt : PORT \mid pt \in f.in\_ports \bullet \\ &\quad \#\{p : pipes \mid f = p.sink\_filter \land pt = p.sink\_port\} \leq 1\end{aligned}$$

$$\begin{aligned}&\forall f : filters;\ pt : PORT \mid pt \in f.out\_ports \bullet \\ &\quad \#\{p : pipes \mid f = p.source\_filter \land pt = p.source\_port\} \leq 1\end{aligned}$$

A system using Pipes & Filters in Z



Pipe

$source\_filter, sink\_filter : Filter$

$source\_port, sink\_port : PORT$

$alphabet : \mathbb{P}\ DATA$

$source\_port \in source\_filter.out\_ports$

$sink\_port \in sink\_filter.in\_ports$

$source\_filter.alphabets(source\_port) = alphabet$

$sink\_filter.alphabets(sink\_port) = alphabet$

Filter

$filter\_id : FILTER$

$in\_ports, out\_ports : \mathbb{P}\ PORT$

$alphabets : PORT \nrightarrow \mathbb{P}\ DATA$

$states : \mathbb{P}\ FSTATE$

$start : FSTATE$

$transitions : (FSTATE \times (Partial\_Port\_State))$
$\quad \leftrightarrow (FSTATE \times (Partial\_Port\_State))$

$start \in states$

$in\_ports \cap out\_ports = \varnothing$

$\mathrm{dom}\, alphabets = in\_ports \cup out\_ports$

$((s_1, input\_observed), (s_2, output\_generated)) \in transitions \Rightarrow$
$\quad s_1 \in states \wedge s_2 \in states$
$\quad \wedge\ \mathrm{dom}\, input\_observed = in\_ports$
$\quad \wedge\ \mathrm{dom}\, output\_generated = out\_ports$
$\quad \wedge\ (\forall p : in\_ports \bullet \mathrm{ran}(input\_observed(p)) \subseteq alphabets(p))$
$\quad \wedge\ (\forall p : out\_ports \bullet \mathrm{ran}(output\_generated(p)) \subseteq alphabets(p))$

System

$filters : \mathbb{P}\ Filter$

$pipes : \mathbb{P}\ Pipe$

$\forall f_1, f_2 : filters \bullet f_1.filter\_id = f_2.filter\_id \Leftrightarrow f_1 = f_2$

$\forall p : pipes \bullet p.source\_filter \in filters \wedge p.sink\_filter \in filters$

$\forall f : filters;\ pt : PORT \mid pt \in f.in\_ports \bullet$
$\quad \#\{p : pipes \mid f = p.sink\_filter \wedge pt = p.sink\_port\} \leq 1$

$\forall f : filters;\ pt : PORT \mid pt \in f.out\_ports \bullet$
$\quad \#\{p : pipes \mid f = p.source\_filter \wedge pt = p.source\_port\} \leq 1$

Formal Notations

Advantages:

- Avoids ambiguities
- Produces precise behavioural models
- Permits rigorous analyses

Disadvantages:

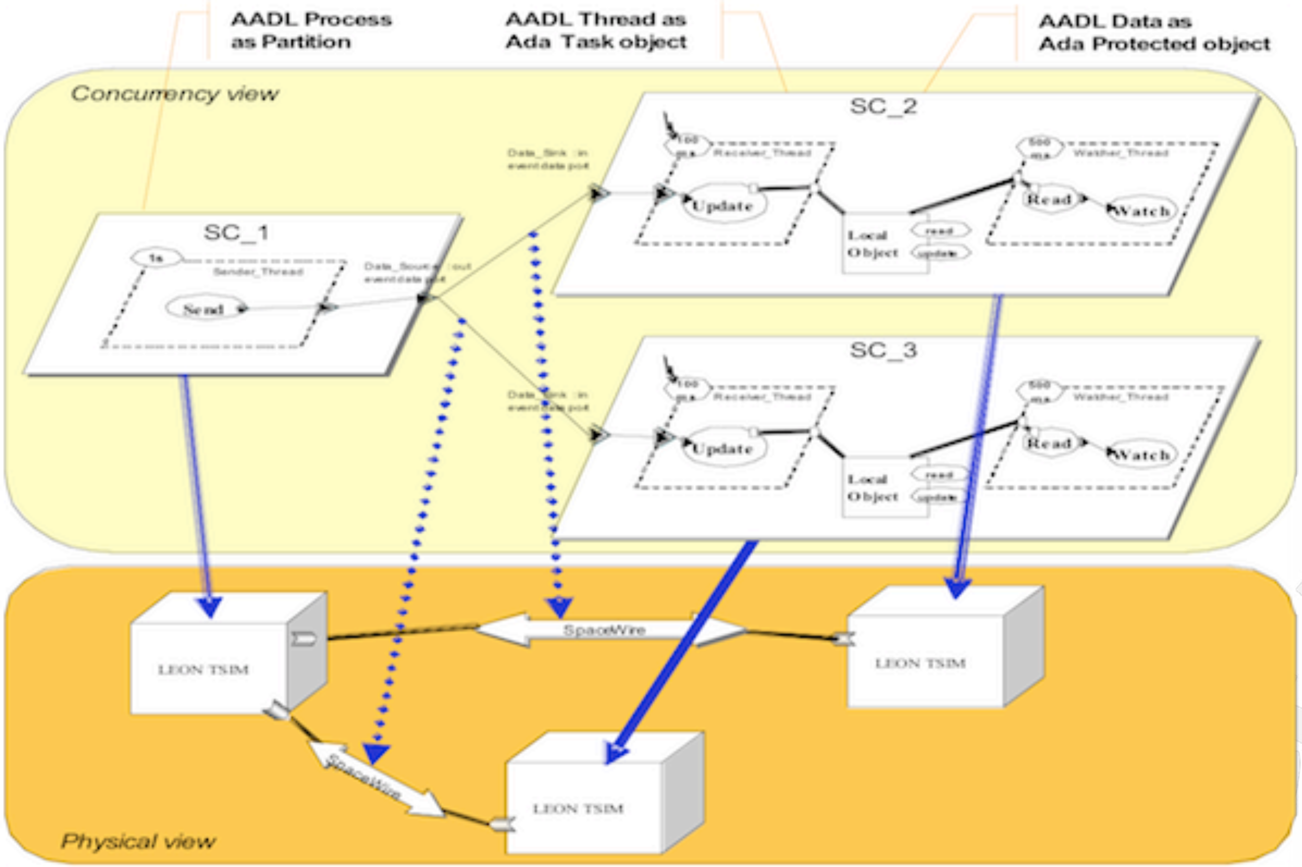
- We lose the ability of communicate with stakeholders

Architectural Description Languages

- Architectural Description Languages (ADLs) are *[usually]* graphical notations providing semantics
  - Help to describe, analyze and reason about architectures
- The majority of the times are domain specific
- The outcome are formal specifications
- Automatic conformance checking can be performed
- Certain qualities can be automatically assessed taking the architectural descriptions as input

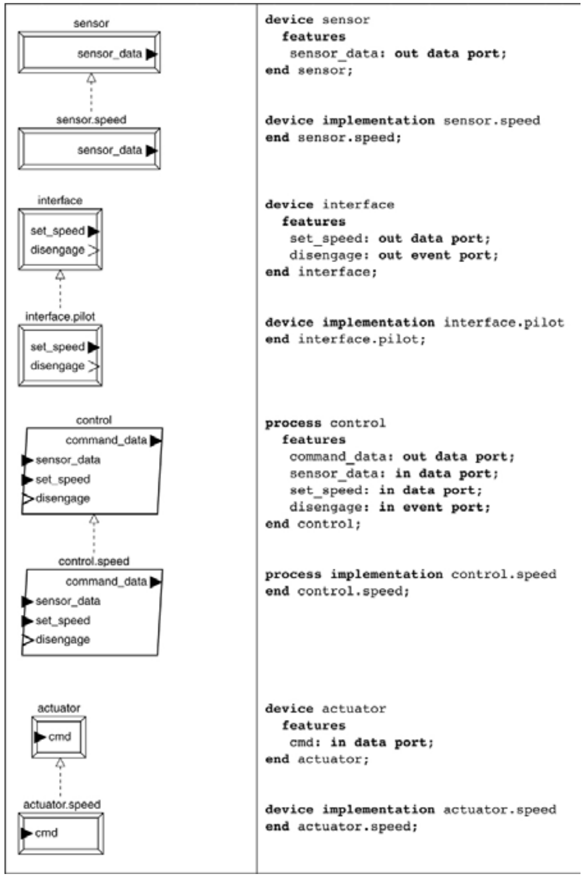
- Examples:
  - AADL: safety critical automotive and avionics systems SAE standard
  - EAST-ADL2: automotive industry standard compliant with the AUTOSAR Reference architecture
  - ACME, Darwin, Rapide, Wright, TASM, Aesop

Example of ADL: AADL

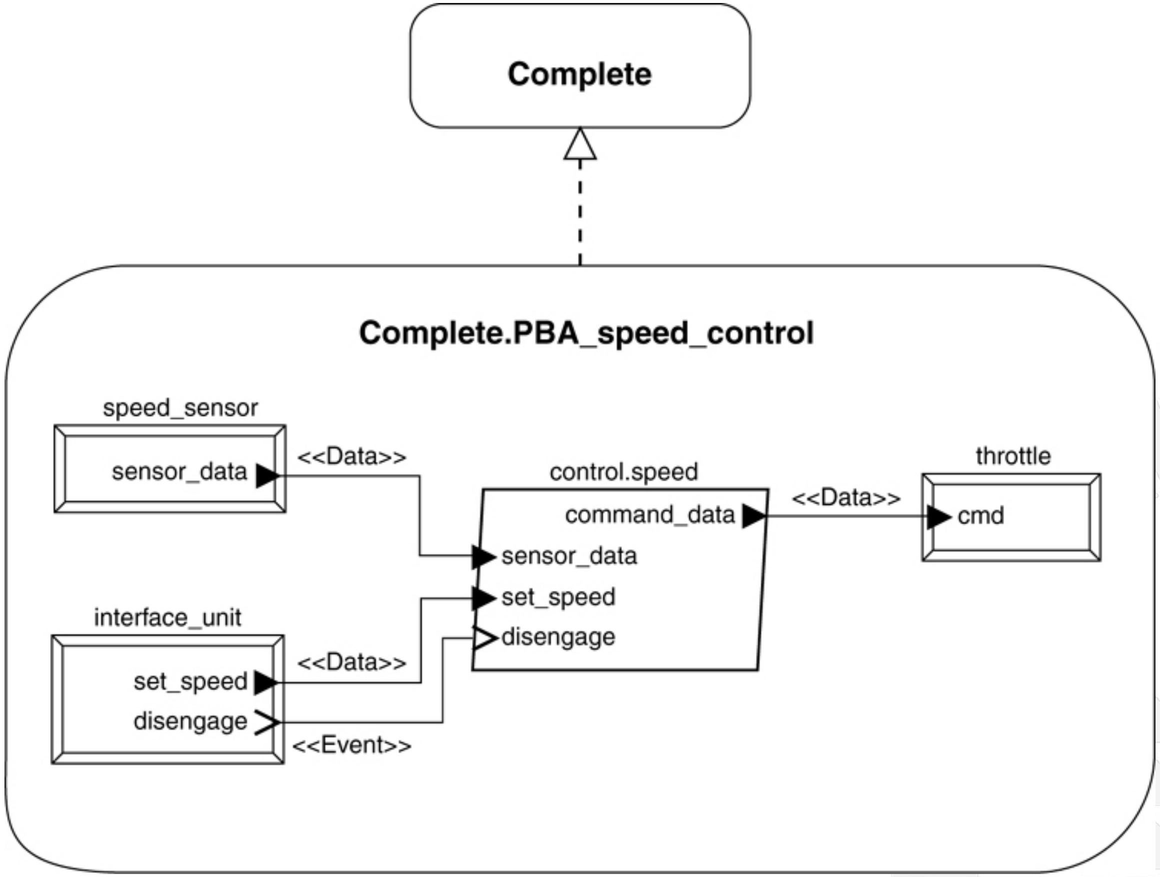


[1] P. H. Feiler and D. P. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language (SEI Series in Software Engineering)*. Addison-Wesley Professional, 2012.

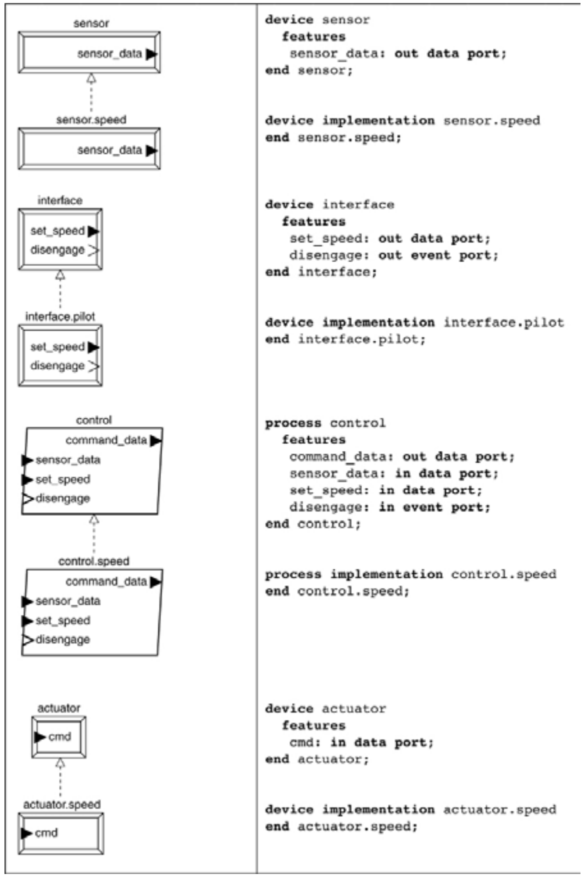
AADL Specification: Power Boat Speed Control



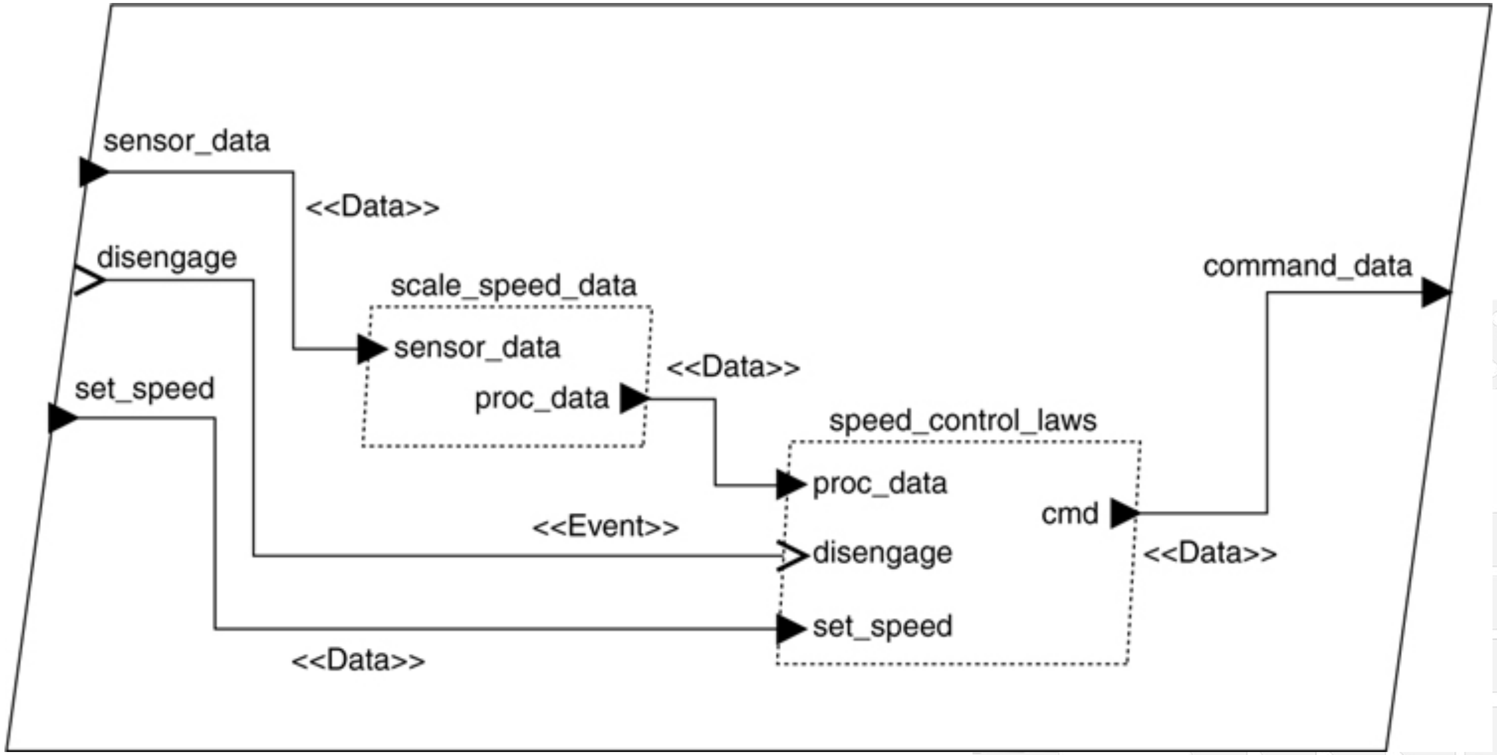
```
device sensor
  features
   sensor_data: out data port;
end sensor;

device implementation sensor.speed
end sensor.speed;

device interface
  features
   set_speed: out data port;
   disengage: out event port;
end interface;

device implementation interface.pilot
end interface.pilot;

process control
  features
   command_data: out data port;
   sensor_data: in data port;
   set_speed: in data port;
   disengage: in event port;
end control;

process implementation control.speed
end control.speed;

device actuator
  features
   cmd: in data port;
end actuator;

device implementation actuator.speed
end actuator.speed;
```



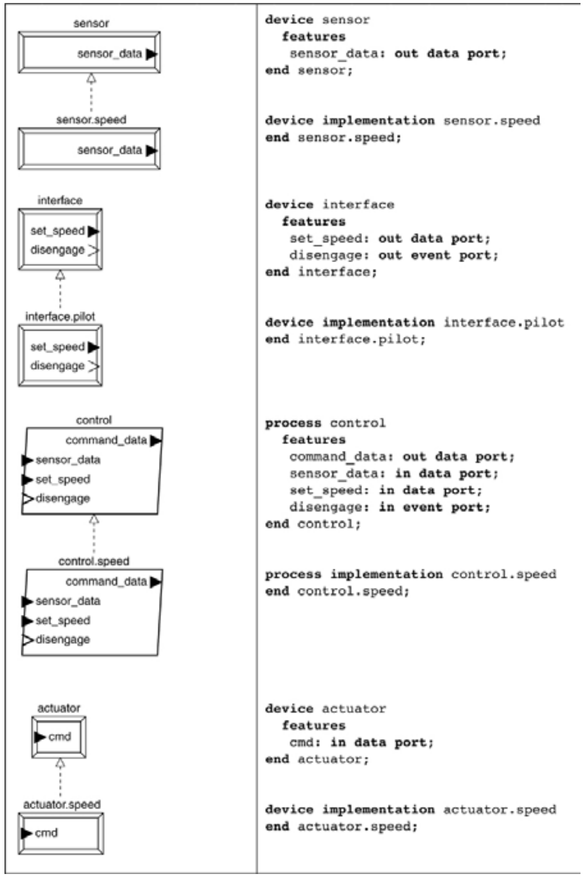
AADL Specification: Power Boat Speed Control



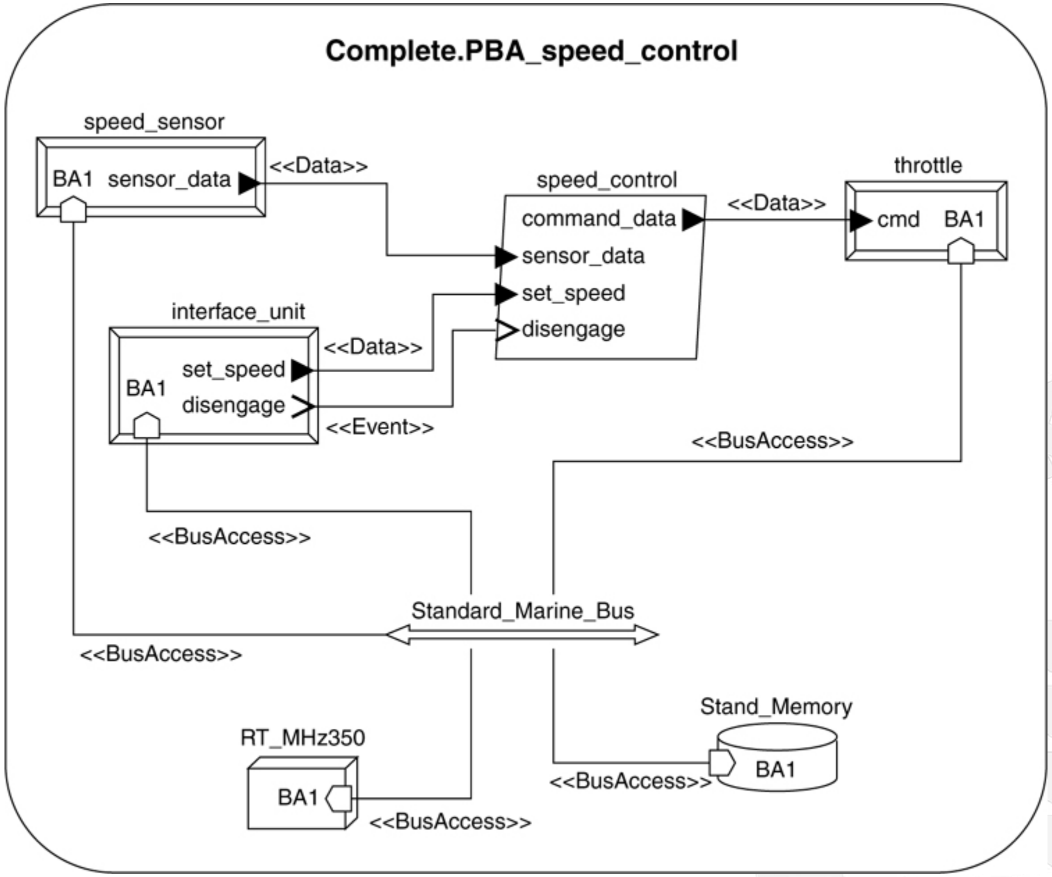
```
device sensor
  features
   sensor_data: out data port;
end sensor;

device implementation sensor.speed
end sensor.speed;

device interface
  features
   set_speed: out data port;
   disengage: out event port;
end interface;

device implementation interface.pilot
end interface.pilot;

process control
  features
   command_data: out data port;
   sensor_data: in data port;
   set_speed: in data port;
   disengage: in event port;
end control;

process implementation control.speed
end control.speed;

device actuator
  features
   cmd: in data port;
end actuator;

device implementation actuator.speed
end actuator.speed;
```



AADL Specification: Power Boat Speed Control


sensor
sensor_data
sensor.speed
sensor_data
interface
set_speed
disengage
interface.pilot
set_speed
disengage
control
command_data
sensor_data
set_speed
disengage
control.speed
command_data
sensor_data
set_speed
disengage
actuator
cmd
actuator.speed
cmd

```
device sensor
  features
   sensor_data: out data port;
end sensor;

device implementation sensor.speed
end sensor.speed;

device interface
  features
   set_speed: out data port;
   disengage: out event port;
end interface;

device implementation interface.pilot
end interface.pilot;

process control
  features
   command_data: out data port;
   sensor_data: in data port;
   set_speed: in data port;
   disengage: in event port;
end control;

process implementation control.speed
end control.speed;

device actuator
  features
   cmd: in data port;
end actuator;

device implementation actuator.speed
end actuator.speed;
```



Beyond the Views

- To completely document a software architecture you also need:
  - Development Road-map
  - System overview/introduction
  - Rationale/explanation of why significant choices where made
  - References to relevant supporting documentation

Seven Rules for Sound Documentation

- **Rule 1:** Write the Documentation from the reader´s point of view
  - The documentation will be write once and [hopefully] read many times
  - Don’t make uninformed assumptions. Chances are that these uninformed assumptions can lead to a big misunderstood.
  - Avoid unnecessary jargon & acronyms.
- **Rule 2:** Avoid repetition
  - Makes the document easier to use and change
  - If something is repeated in a slightly different way can lead to confusions
  - Repeat only if something needs to be clarified.

Seven Rules for Sound Documentation

- **Rule 3:** Avoid Ambiguity
  - Occurs when documentation can be interpreted in more than one way and when at least one of them is incorrect
  - The most dangerous is the undetected ambiguity
  - **Solution:** A well-defined notation with precise semantic is a good way to mitigate ambiguity. Always describe the notation used with a key / legend
- **Rule 4:** Use a Standard Organization
  - Helps the reader to navigate, helps the writer to plan and organize the contexts, helps checking for completeness

Seven Rules for Sound Documentation

- **Rule 5:** Record Rationale
  - The purpose of the documentation is not only to record the decisions made but also:
    - Why the decision was made
    - What were the alternatives
    - Why the other alternatives were discarded
  - Avoid architectural knowledge evaporation
- **Rule 6:** Keep it Current but Not Too Current
  - The documentation needs to be up to date,
  - But having to review the documentation to reflect decisions that will not persist is an unnecessary waste
- **Rule 7:** Review Documentation for Fitness of Purpose
  - Only the audience determines that documentation contains the right information presented in the right way

Software Architectures and Quality

Documenting Software Architectures

Javier González Huerta

javier.gonzalez.huerta@bth.se

in real life