Software Architecture Documentation in the Real World

Markus Völter
voelter@acm.org
www.voelter.de

About me

völter.de ingenieurbüro für softwaretechnologie

Markus Völter
voelter@acm.org
www.voelter.de

- Independent Consultant
- Based out of Goeppingen, Germany
- Focus on
  - Model-Driven Software Development/DSLs
  - Software Architecture
  - Product Line Engineering

Founder and Editor of

Software Engineering Radio
the Podcast for Professional Developers
http://se-radio.net

CONTENTS

- What is Software Architecture
- Documenting Software Architectures
  - (Structured) Glossaries
  - Patterns and the Pattern Form
  - Pattern Languages
  - Tutorials and FAQs
  - Diagramming and Modeling
  - Channels
  - What about Code?
- Layout and Typography
- Diagramming Guidelines
- Summary

CONTENTS

- **What is Software Architecture**
- Documenting Software Architectures
  - (Structured) Glossaries
  - Patterns and the Pattern Form
  - Pattern Languages
  - Tutorials and FAQs
  - Diagramming and Modeling
  - Channels
  - What about Code?
- Layout and Typography
- Diagramming Guidelines
- Summary

What is Software Architecture

- **Wikipedia:**
  The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships between them.
- **Eoin Woods:**
  Software architecture is the set of design decisions which, if made incorrectly, may cause your project to be cancelled.
- **Hayes-Roth:**
  The architecture of a complex software system is its "style and method of design and construction".

What is Software Architecture II

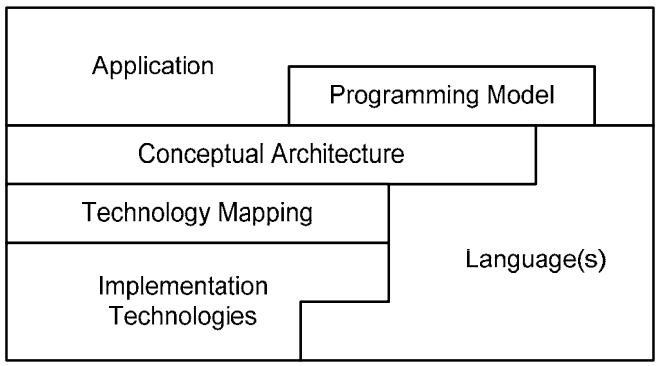
- **Boehm, et al., 1995:**
  A software system architecture comprises
  - A collection of software and system components, connections, and constraints.
  - A collection of system stakeholders' need statements.
  - A rationale which demonstrates that the components, connections, and constraints define a system that, if implemented, would satisfy the collection of system stakeholders' need statements.
- **Other:**
  Architecture is everything that is expensive to change later.
- **Mine:**
  Everything that needs to be consistent throughout a software system (“strategic design” – Eric Evans)

Architecture / System Categories – Focus

- **Small, ad-hoc systems** typically developed by small teams or single people
- **Large systems,** that are developed by larger teams, typically long-lived, strategic
- **Product Lines & Platforms**, i.e. base architectures on top of which a family of systems is built often by several teams, strategic
- We will primarily **focus on large systems & product lines** – since for small ad-hoc systems architecture documentation is often not essential

Aspects of Software Architecture

- This diagram outlines a number of **terms and concepts** we will use in the rest of this presentation.



Architectural Process

- An architecture (conceptual and application) **evolves over time** as we build a system (or over several systems)
  - There may be a more or less appropriate initial idea…
  - … maybe based on architectural styles & patterns …
  - … but it will always evolve over time
- However, at any given time there is the **one-and-only correct** architecture
  - The notion of what this one-and-only correct architecture is changes over time, but at any given time it is well-defined
- So, it is essential that applications are (in the process of becoming) **consistent with that architecture** at any point in time to keep the system consistent
  - Ideally you want to “enforce” the architecture via tools…

What needs to be documented?

- **Conceptual level:**
  - The conceptual architecture
  - Stakeholders and their needs
  - Rationales why the conceptual architecture is as it is
  - The programming model
  - The technology mapping
- **Application Level:**
  - The application architecture
  - Stakeholders and their needs
  - Rationales why the application architecture is as it is
- We will **focus** mainly on the **conceptual level**

CONTENTS

Documentation Fundamentals for all Artifacts

- For each artifact, define and state the **target audience** – and make sure the content is relevant to that audience
- Use a suitable **medium / channel** (see below)
- Document only as **little as possible**
- **Avoid duplication!** Document every aspect **in one place only** – and use links (not just references!) to connect related topics
- Just as with code, put documentation into the **Version Control System** (and not on some strange Web Collaboration Platform)
  - That's true for the development of the docs
  - There might be a different publishing channel

Documentation Fundamentals for all Artifacts II

- Always document top down
  - provide **progressively more details** only for those readers who want to actually know them
  - Make sure **concepts and the big picture is understandable** without rummaging through all the details!
- Try to **structure** an architecture (or at least its documentation) into **layers**, or **levels**, or **rings**
  - First cover only the basic layer
  - Then add more and more layers to the picture
  - This makes things easier to comprehend
- For important things, use **several** descriptions, representations, formulations, channels, renderings…
- **Visualize!** … see later.

CONTENTS

Glossaries

- A glossary **lists the relevant architectural concepts** and their meaning and relationships
- It is useful to **introduce the basic ideas** and familiarize readers with the terms used in the architecture
- To make the glossary less abstract, make sure an **example** is provided for each of the introduced terms
- It can be used for the conceptual architecture and the application architecture – but it is **more important for the conceptual** architecture
- **Target Audience:** Everybody technical

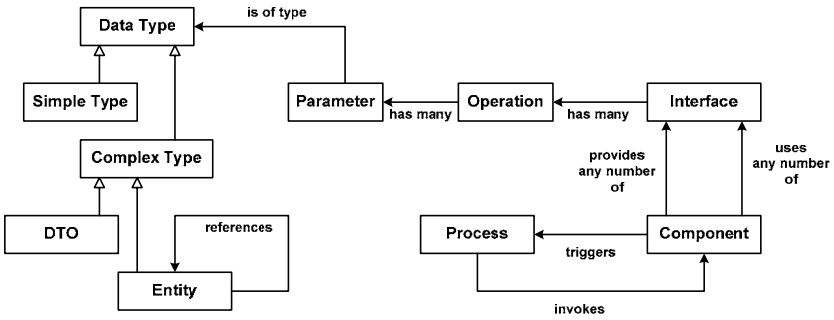
Glossary Example

Data type	Represents a certain chunk of data. Data types can either be simple types (string, int, boolean and the like) or *Complex Types*.
Complex Type	A complex data type is basically a like a struct in that it has named and typed attributes. There are two kinds of complex data types: *Entities* and *Data Transfer Objects*
Entity	persistent entities that have a well-defined identity (and can thus be searched), and that can have relationships to other entities.
Data Transfer Object	Data transfer objects have no identity and are not persistent.
Interface	A contract that contains a number of operations; operations are defined in the usual way (parameters, return type, exceptions)
Component	A component is a well-defined piece of behaviour. It does not implement technical concerns. Each component can provide a number of *Interfaces*. It can also use a number of interfaces (provided by other components). Components are stateless (i.e. cannot "remember" things from one invocation to another)
Process	We also explicitly support business processes. These are considered to be expressable as state machines. Components can trigger the state machine by supplying events to them. In turn, other components can be triggered by the state machine, resulting in the invocation of certain operations defined by one of their provided interfaces.

Structured Glossaries

- Represents the core concepts as a diagram, **highlighting the relationships** between the concepts
- **UML Class Diagrams** are very well suited for this kind of description
- They are **an addition** to normal glossaries, **not a replacement**, since they don't explain concepts – they just show their relationships
- **For modelers:** these are not the same as meta models, since they are less formal, less detailed, and generally not "implementable"

Structured Glossaries Example



CONTENTS

- What is Software Architecture
- Documenting Software Architectures
  - (Structured) Glossaries
  - **Patterns and the Pattern Form**
  - Pattern Languages
  - Tutorials and FAQs
  - Diagramming and Modeling
  - Channels
  - What about Code?
- Layout and Typography
- Diagramming Guidelines
- Summary

Referencing Patterns

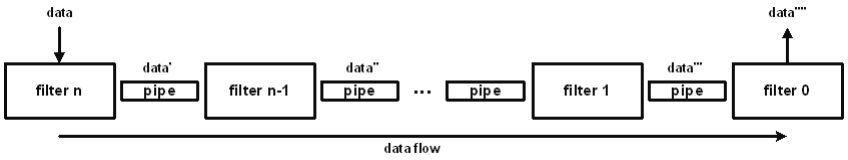
- If you're describing a certain software structure, and that structure has already been documented as a pattern, then it makes sense to **reference that pattern** – your readers might know it!
- There's a **huge body of patterns** in the literature, on topics such as
  - Distributed (Object) Systems [POSA2, POSA4]
  - Remoting Infrastructures [Remoting Patterns]
  - Resource Management [POSA3]
  - Patterns of Enterprise Application Architecture [PoEAA]
  - Enterprise Integration Patterns [EIP], Integration Patterns [IP]

Architectural Patterns

- Architectural patterns can be used to **describe well-working architectural styles** and blueprints.
- Many have been described in the POSA series books, for example, specifically in [POSA1].
- Examples include
  - Blackboard
  - Pipes and Filters
  - Microkernel
  - Components & Connectors
- Many of the same architectures have also been documented as **architectural styles** by the SEI. These can be references, too, of course.

The Pipes and Filters Pattern

- **Thumbnail:**
  - The Pipes and Filters pattern provides a structure for systems that process a stream of data.
  - Each processing step is encapsulated in a filter component.
  - Data is passed through pipes between adjacent filters.
  - Recombining filters allows you to build families of related systems.



- **Known Uses:**
  - Compilers (different stages)
  - UNIX shells
  - CMS Pipelines
  - Image Processing (ALMA)

Architectural Patterns / The Pipes and Filters Pattern II

- **Consequences:**
  - \+ No intermediate files necessary, but possible
  - \+ Flexibility by filter exchange or recombination
  - \+ Reuse of filter components
  - \+ Rapid prototyping of pipelines
  - \+ Possibility of improved efficiency by parallel processing
  - – Shared state may be expensive and complicated
  - – Possible data transformation overhead
  - – Error Handling

Architectural Patterns as Fix Points

- Architectural Patterns serve as **fix points in the design space** of an architecture.
  - You understand the requirements
  - You design an initial architecture
  - You find it resembles a certain architectural pattern
  - You analyze the differences. Are they essential?
  - You then look at the patterns consequences to see if they are acceptable.
  - Then you may want to iterate… until you maybe hit another pattern in the architectural design space.
- When using MDSD, architectural patterns can be used as a **basis for architectureal metamodels** (see below)
  - The solution structure of an architectural pattern can be described as a metamodel.

Writing your own Patterns

- If you come up with certain **recurring best practices** in your domain (technical or functional) you may want to write these down as patterns.
- The pattern forms (there are various forms) all have in common that they **require the author to structure the content very strictly**.
  - This forces the author to think hard about stuff such as applicability, forces or consequences
  - For readers, well-structured content becomes easier to comprehend

Using the Pattern Form

- Even if something is not recurring and hence is not a pattern…
- Writing things up in pattern form **improves the effectiveness of communication**, provides a means to break down complex structures and **generally improves writing style** (and author proficiency).
- Once you're accustomed to the patterns form, **you will use it implicitly** when writing any kind of technical documentation, i.e.
  - Start by setting the context,
  - Explain when and for who the following stuff is interesting
  - Describe problem and solution in increasing levels of detail
  - And then elaborate on the consequences.
  - Finally, you'll point to related material

CONTENTS

The challenge of documenting complex architectures

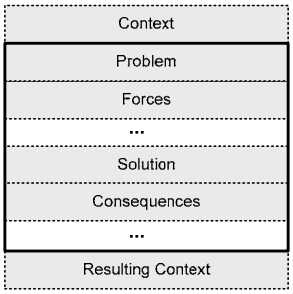
- It is **not enough to simply collect** descriptive data about an architecture
  - e.g. a big UML model or a collection of diagrams or APIs
- rather, communicating an architecture requires a **well-defined, didactic approach**, where
  - You start with a **motivation** of what the general problem is (what is it that the architecture should achieve)
  - Then you provide an **overview** over the solution strategy
  - … and **progressively** provide more and more **details** …
  - Until you've covered all cases incl. border cases

Inner Structures of complex Systems

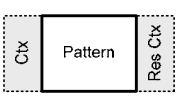
- **Pattern Languages** are collections/sequences of patterns that describe a “whole”,
  - The overall structure of the system is **too complicated** to be described in one step – thus the language.
  - Sometimes there **are alternative sequences** through the pattern language describing various alternatives of the “whole”
  - **Group** patterns into **chapters** to implement the layers/levels/rings mentioned before
- A pattern language thus describes **how to build** a complex system of a certain type
- There are **various examples** of such pattern languages,
  - Many cover middleware technology [Server Component Patterns, Remoting Patterns] , and
  - They are published in various forms

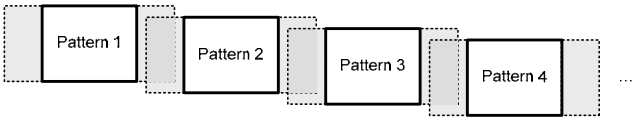
From Patterns to Pattern Languages

- The pattern is the **undividable** entity of knowledge/documentation



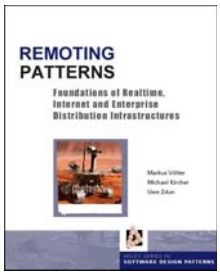
- Pattern Languages are built by having subsequent patterns **solve problems that arise from using a previous pattern.**

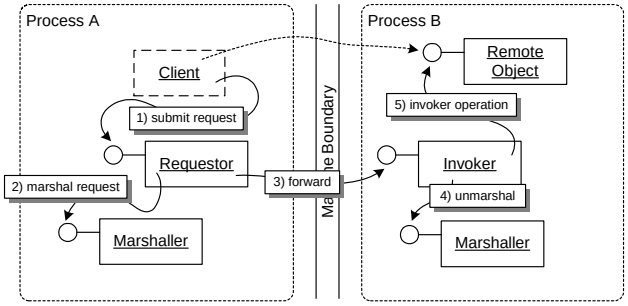




Example: Remoting

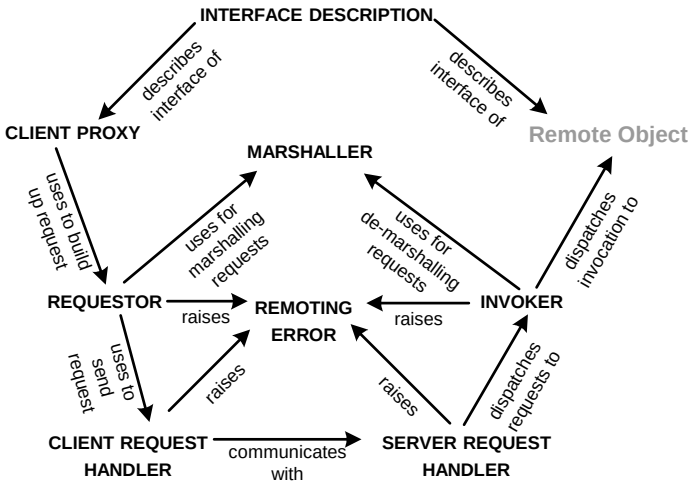
- Describes the internal architecture of **remoting middleware** such as CORBA, WebServices or .NET Remoting
- It can be seen as a pattern language that describes the **internal details of Broker architectures** in industrial practice.





Example: Remoting II

- A **structured glossary** (per chapter!) shows the conceptual relationship between the patterns

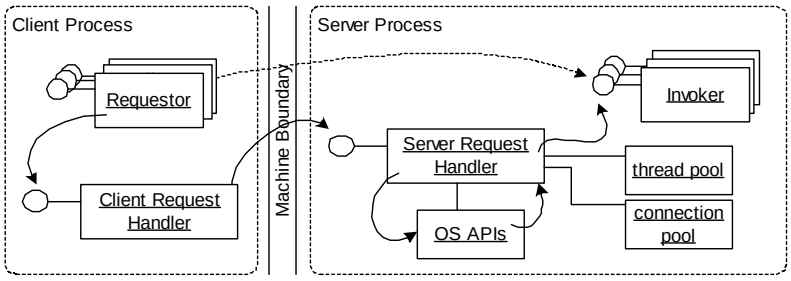


Example: Remoting III: Server Request Handler

- **Context:** You are providing remote objects in a server application, and invokers are used for message dispatching
- **Problem:**
  - The request message has to be received from the network;
  - Managing communication channels efficiently and effectively is essential
  - Network communication needs to be coordinated and optimized
- **Solution:** Server request handler deals with all communication issues of a server application:
  - Receives messages from the network
  - Combines the message fragments to complete messages
  - Dispatches the messages to the correct invoker
  - Manages all the required resources (connections, threads, ...)

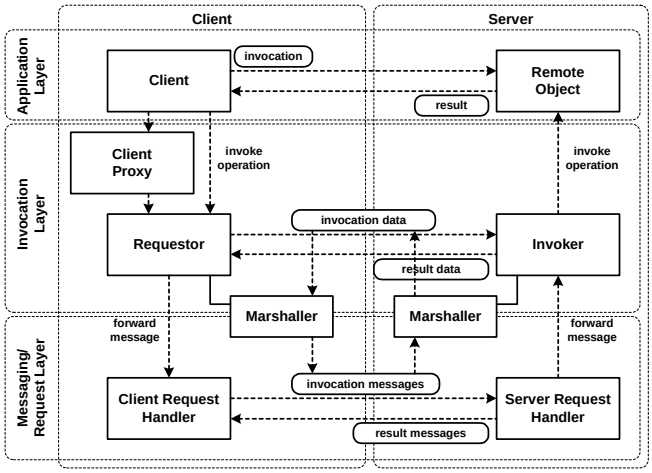
Example: Remoting IV: Server Request Handler 2

- Each pattern in the language is illustrated with a diagram that shows the **relationships** and **interactions** with other building blocks of the overall system.



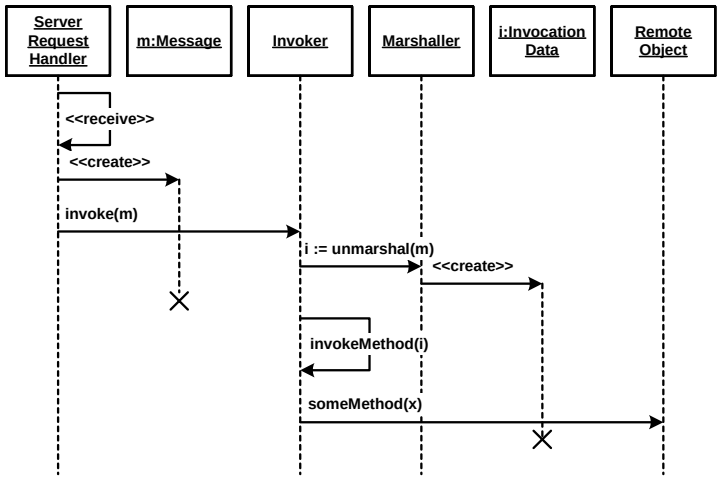
Example: Remoting V

- Here is another view showing the interactions, **grouped into layers**



Example: Remoting VI

- **Interesting interactions** are illustrated with sequence diagrams (typically a couple of diagrams per chapter)



CONTENTS

- What is Software Architecture
- Documenting Software Architectures
  - (Structured) Glossaries
  - Patterns and the Pattern Form
  - Pattern Languages
  - **Tutorials and FAQs**
  - Diagramming and Modeling
  - Channels
  - What about Code?
- Layout and Typography
- Diagramming Guidelines
- Summary

Tutorials & FAQs

- When documenting the **programming model**, the respective documentation
  - Needs to be problem/solution-based
  - Needs to explain common things first, and exceptional things later
  - Needs to provide a step-by-step approach
- Here's what has proven to be useful:
  - **Tutorials** (Walkthroughs) for typical cases of increasing complexity (e.g. 5, 20 and 60 minute tutorial)
  - **FAQs** to illustrate exceptional cases in a problem → solution fashion
- Note that tutorials and FAQs **should not contain too much rationale** for what they explain – rather, refer to other documentation for that. Make it practical!

Examples of what you need to address

- How do I set up the **environment** (IDE, Repository, Build)?
- How do I **acquire and release resources**, who manages the lifecycle of certain artifacts?
- What other **protocols** do I need to follow (e.g. locking)
- In which chunks, and where, do I put my **application logic**?
- What are the constraints wrt. to **concurrency**
- How do I **interact with the platform** and environment?
- Which **aspects** of the underlying programming languages or frameworks are **disallowed?**
- Important **conventions and idioms**, including certain important naming conventions
- Where and how do I write my **unit tests**?

CONTENTS

Models

- **Definition I:** (www.answers.com/topic/model)
  A schematic description of a system, theory, or phenomenon that accounts for its known or inferred properties and may be used for further study of its characteristics
- **Definition II:** (www.ichnet.org/glossary.htm)
  A representation of a set of components of a process, system, or subject area, generally developed for understanding, analysis, improvement, and/or replacement of the process
- **Definition III:** (ecosurvey.gmu.edu/glossary.htm)
  an abstraction or simplification of reality

Diagrams

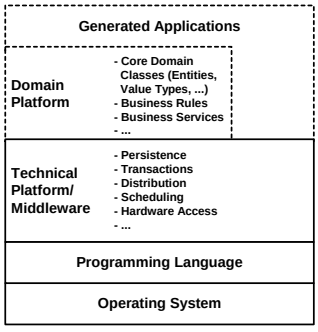
- **Definition I:** (en.wikipedia.org/wiki/Diagram)
  A diagram is a simplified and structured visual representation of concepts, ideas, constructions, relations, statistical data, anatomy etc used in all aspects of human activities to visualize and clarify the topic.
- **Definition II:** (careers.ngfl.gov.uk/help/definitions/14_2_image.html)
  Diagram means a graphical or symbolic representation of something, usually showing the relationship between several items.
- **Definition III:** (www.evgschool.org/Columbus%20vocabulary.htm)
  A diagram is a drawing, sketch, plan, or chart that helps to make something easier to understand

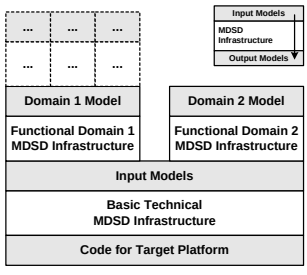
Models vs. Diagrams

- Diagrams are mainly used to **"intuitively communicate"** something to **humans**
- Models are mainly used to **"formally specify"** something to **tools**
- Hence, models need to be **correct** and **complete** wrt. to the aspect, viewpoint or concern they describe.
  - They need to be based on a well-defined language
- **Diagrams** can be used to **represent models**.
- Models, however, can also be represented in other, non-diagram ways (e.g. with textual notations)

Examples of Architectural Diagrams

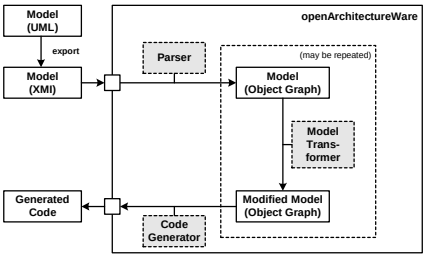
- This diagram shows the **layers** in a typical **distributed system** architecture
  - The visual layers are meant to actually illustrate a strict layers architecture
- **Transformation architecture** of a cascaded MDSD application
  - It is built by recursively applying the atomic building block shown in the top right corner



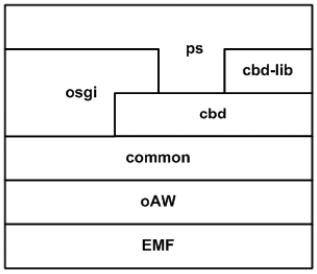


Examples of Architectural Diagrams II

- **Model Transformation architecture** in the tool openArchitectureWare
  - The boxes are hierarchical structures of the tool
  - The arrows represent data flow

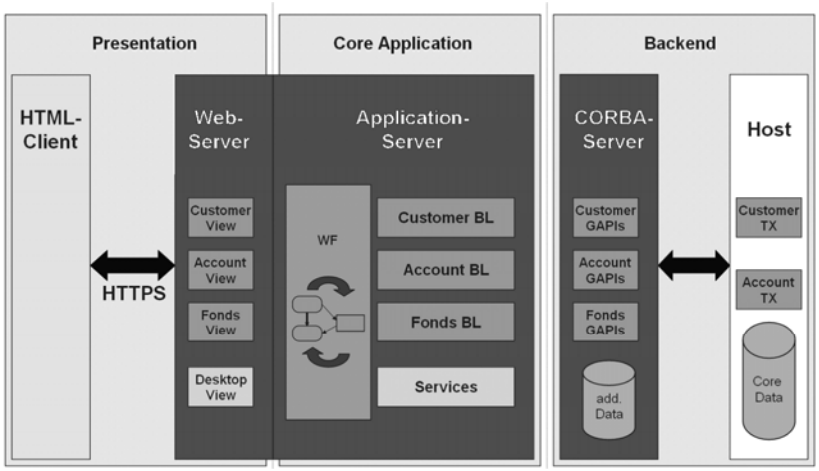


- **Layers** of a product-line architecture
  - If you visually draw layers, make sure this is actually what you want to communicate (i.e. there really is a layering in the system you describe)



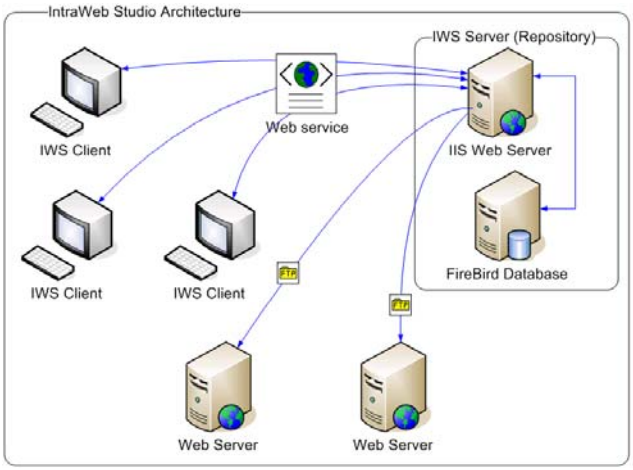
Examples of Architectural Diagrams III

- A three-tier enterprise system. Useful diagram?



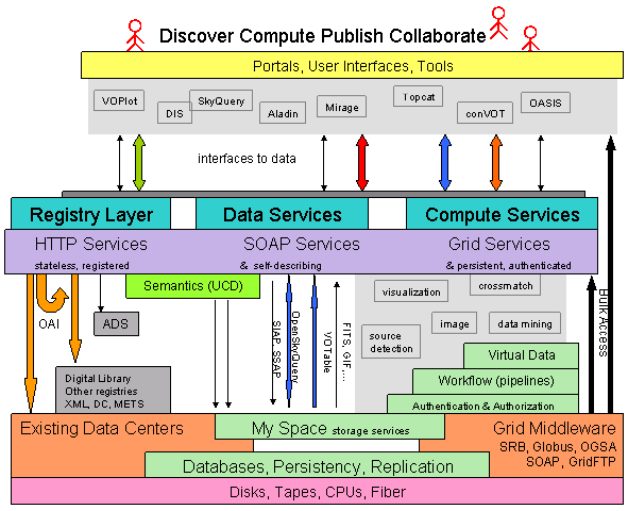
Examples of Architectural Diagrams V

- Some other Architecture. Useful diagram?
  (it is certainly very nice ☺)



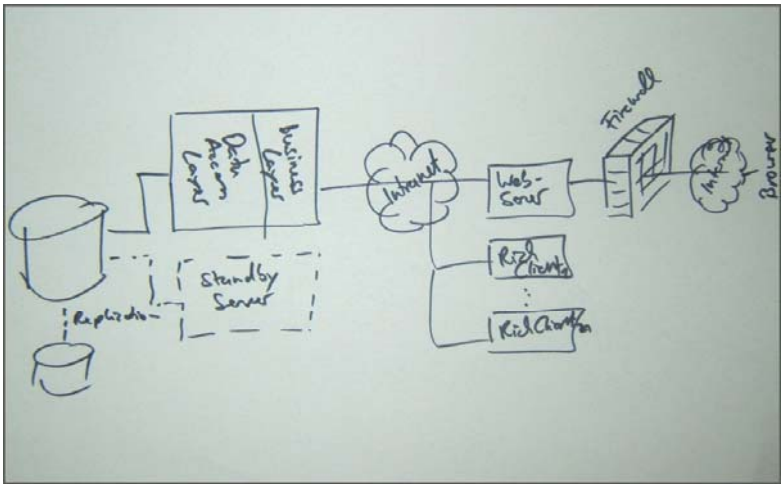
Examples of Architectural Diagrams VI

- One more… Useful? (It is certainly ugly!)



Examples of Architectural Diagrams VII

- And you don't need a fancy tool, you can use a **flipchart** (assuming your handwriting is readable!)

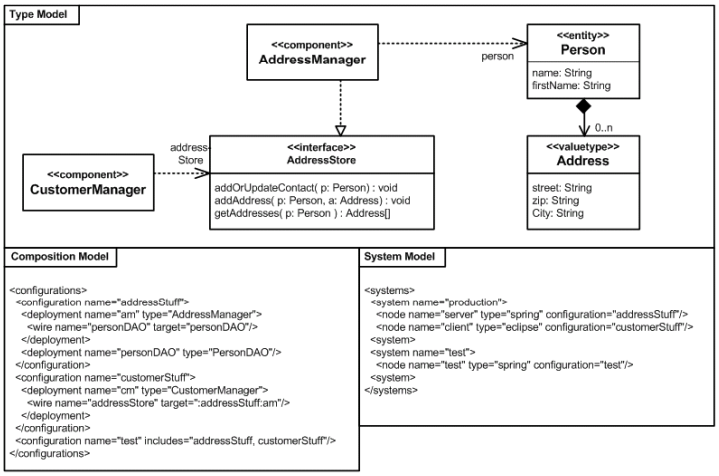


The use of Diagrams

- Diagrams are used to communicate to people.
- They often use **nice, intuitive symbols**, they are (typically) not based on a well-defined (modeling) language.
- Often, the **meaning is not really clear**
  - you need explaining text or somebody talking to you as they draw the diagram
- However, diagrams are **very very useful** in documenting architectures, as long as
  - You **explain** what the diagram **means**
  - And you are **consistent** wrt. the notation among the set of diagrams you use
  - … you might even use a standardized modeling language

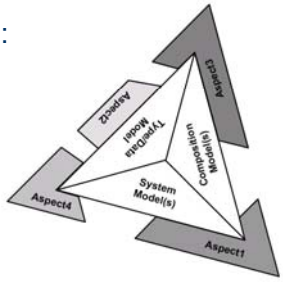
Example of an Architectural Models

- A three-viewpoint model for a **component-based enterprise** system (using UML and XML)



Viewpoints

- When building models, it is essential to define several **viewpoints** of the system
- In the previous example, we used the following three structural viewpoints:
  - **Type Model**: Components, Interfaces, Data Types
  - **Composition Model**: Instances, “Wirings”
  - **System Model**: Nodes, Channels, Deployments
- Often, additional viewpoints are needed:
  - Persistence
  - Security
  - Forms, Layout, Pageflow
  - Timing, QoS in General
  - Packaging and Deployment
  - Diagnostics and Monitoring

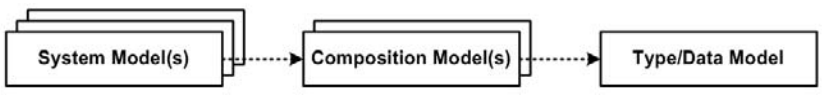


Viewpoints II – the 4+1 Model

- Originally conceived by **Philippe Kruchten**
- **Core Views** used to describe the architecture
  - **Logical View:** Functional requirements (e.g. UML diagrams, structural and behavioral)
  - **Process View:** Non-Functional (concurrency, performance, scalability)
  - **Development View:** file layout, project structure, versioning, packaging
  - **Physical View:** topology, communication, deployment
- **+1:** Scenarios (Use Cases)
- Not too much used in practice…

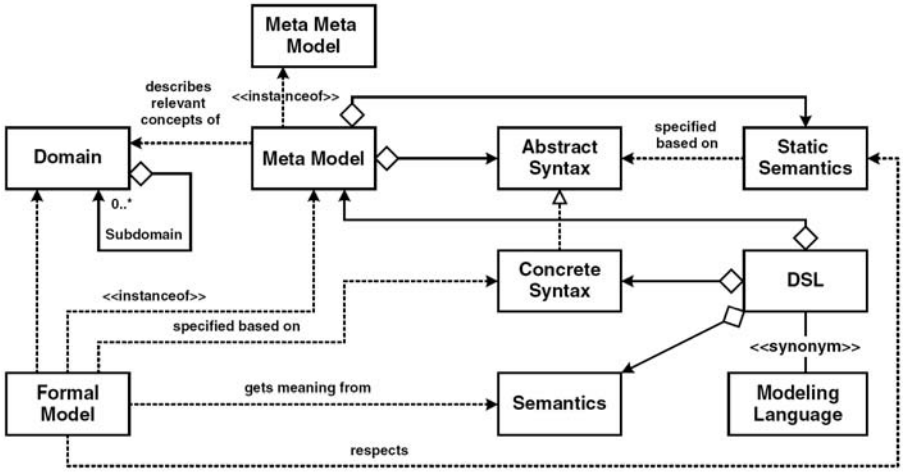
Viewpoints III – connection to modeling

- If you want to use viewpoints in conjunction with modeling, each viewpoints needs it own **modeling language** (or language partition)
- You need to come up with a **meta model** suitable for expressing that viewpoint, and with a **suitable concrete syntax**.
- The meta models (and hence, languages, and viewpoints) need to **depend on each other** in a suitable way.



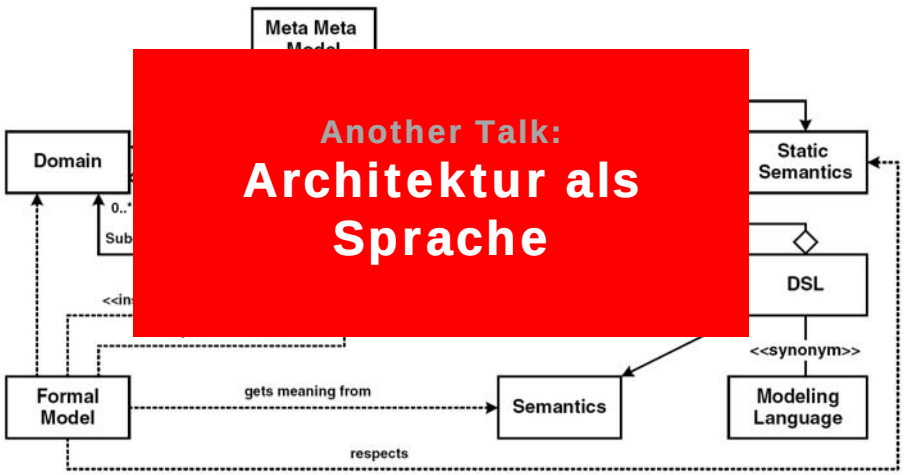
Modeling Languages (DSLs)

- Here is a **structured glossary** of the necessary concepts:



Modeling Languages (DSLs)

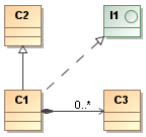
- Here is a **structured glossary** of the necessary concepts:



Another Talk:
Architektur als Sprache

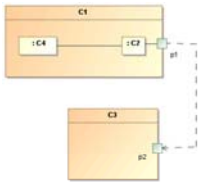
The role of UML

- UML is not specifically tailored for **software *architecture* modeling**, but rather for software modeling in general
  - You can use UML for **diagramming**, as well as for **modeling** – you typically need a profile for the latter.
- The question is, though, which UML diagrams are suitable for architecture descriptions
  - We use green for modeling, red for diagramming
- **Class Diagrams**
  - Useful for architecture meta models
  - And for structured glossaries
  - … and using a profile for every other structural aspect, in principle… but the graphical symbols are very limited. Hence custom diagrams or things like FMC are used.

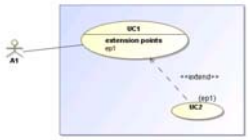


The role of UML II

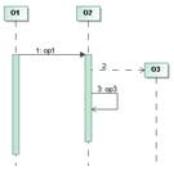
- **Composite Structure Diagrams**
  - Extremely useful for modeling hierarchical structures of components, instances, as well as component connections
  - My favourite kind of diagram in UML ☺



- **Use Case Diagrams**
  - (More or less) useful for describing usage scenarios and requirements towards the architecture

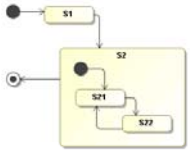


- **Sequence Diagram**
  - Very useful for illustrating the interactions among architectural components
  - Note the sequence diagrams are good for scenarios, not for closed, complete behavioral specification

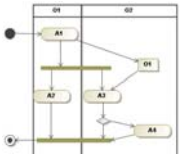


The role of UML III

- **State Diagrams**
  - Very useful illustrating state changes of components, if their behavior is state-based
  - Very useful for defining protocols between components, and for formally specifying state-based behavior

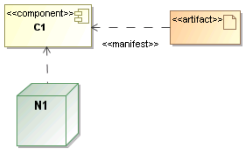


- **Activity Diagrams**
  - Useful for describing activities, their allocation to components and data flow
  - They can be used to formally specify behaviour, but I don't do this very often



The role of UML IV

- **Implementation Diagrams (Component & Deployment)**
  - Moderately useful for modeling the packaging of components into deployment artifacts and runtime processes and executables, and
  - Moderately useful for describing system (hardware) infrastructure and the allocation of processes and components to them



The role of UML V: Summary

- The UML can do **everything** … in principle.
- Tool support is of **varying quality**, but it is getting better.
  - This is especially true for profile support and tool customization!
- Here is how I like to use (or not use) UML in the context of architecture
  - I use it for architecture meta models
  - I define domain specific architecture DSLs and work with these languages for formal modeling
  - I really like composite structure diagrams
  - I use sequence diagrams to illustrate interactions
  - I use informal (Visio-based) notations for illustrations

CONTENTS

Printable Material

- To be **read in one piece** to teach concepts
- **Readability** and Formatting is important
- These days mainly implemented as PDFs
- **Suitable for**
  - Conceptual Architecture (Patterns, Pattern Languages, Glossaries, Meta models, DSLs)
  - Programming Model Tutorials

Online References

- Used for **looking up details**
- Readability and Formatting is not so important, **searchability and indexing** more important
- These days mainly implemented as HTML or Wikis
- **Suitable for**
  - Programming Model APIs and FAQs
  - Glossaries

Blogs

- It is useful if the architecture/platform team sets up an **architecture blog** to keep application developers up-to-date with recent developments.
- **This is useful for**
  - Updates wrt. to the evolution of the platform
  - Tips & Tricks on how to use the architecture
  - Success stories and other news

Flash Demo / Video / Animation

- Here you typically **screen-capture** some activity related to your architecture and record it for replay.
- **Explaining Text** is either recorded (audio) or added later in keys/bubbles.
- This is useful for
  - Programming Model Tutorials
  - … especially if a lot of pointing and clicking, or other “tool use” is required

Podcasts & Video

- Podcasts are **audio files** published via an **RSS feed** in regular episodes (“audio-blog”)
- **This is useful for**
  - General discussions about concepts
  - News and stories in general
- Complex technical concepts **can be explained** in audio only
  - See se-radio.net, the podcast for developers
  - Make sure it’s always **at least two people talking** otherwise it will be boring quickly
  - Make sure things are repeated or clarifying questions are asked
- **Video is useful for**
  - General discussions about concepts – since you can film two guys on the flipcharts

The Back Channel!

- Be sure to **encourage feedback** of the users of your architecture. **Accept** feedback and criticism, and **improve** your documentation accordingly!
  - Create tutorials, FAQs and glossaries as **Wikis**, so that users can contribute, enhance and comment (I am not sure this is useful for the more conceptual stuff)
  - If you use podcasts or videos, **invite users** to “appear on the show”
  - **Exchange architects and developers**, to make sure architects eat their own dog food, and developers understand how complex it is to integrate all the(ir) requirements into the architecture

CONTENTS

What about Code?

- It is useful to **document important APIs** in the code and use tools such as *JavaDoc* or *DoxyGen* to generate online API documentation.
- However, **code (documentation) cannot replace** tutorials, glossaries, rationales, FAQs, or any of the other kinds introduced before – **code does not tell a story**!
  - Of course, tutorials and FAQs contain code to show how to use the programming model
- It is useful to **refer to code** from any of the other artifacts if people want more details.
- Do not document things elsewhere that are **obvious** and **understandable** from the code.

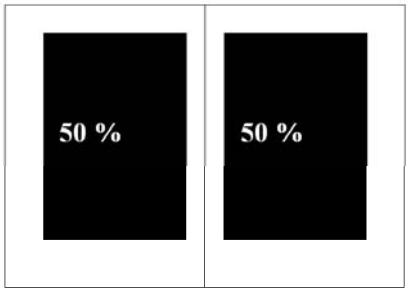
CONTENTS

Page Layout & Typography

- Typography influences the reader when reading the document
- You'll read faster if the **page geometry** is suitable and you've chosen **suitable fonts**
- You should use **document templates**
  - that contain only stylistic aspects, not 25 sections to fill in
  - They are prepared by a small number of people
  - Hence, good layout will become pervasive
- And always use **change marks** for revisions of the documents – otherwise readers will not read anything beyond version 1

Page Layout & Typography II

- **50% Page contents**
  - seems to be too little
  - but is appropriate for the readers' fields of view
  - Typically a good decision for documents

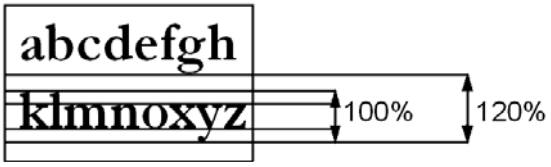


- **2 – 2.5 Alphabets per Line**
  - Long lines are **hard to follow**
  - Short lines require **too many "carriage returns"**
  - Might result in **several columns** in a document

abcdefghijklmnopqrstuvwxyz abcdefghijklmnopqrstuvwxyz

Page Layout & Typography III

- **120% Line Spacing**



- **2 Fonts**
  - Use Serif Font for the text (guides the eye)
  - Use Sans Serif for Headlines
  - ... and maybe Monospaced for Code

Beispiel
Für die Überschrift wurde hier die serifenlose Schrift Frutiger gewählt. Im Haupttext kommt die Serifen-Schrift Garamond zum Einsatz.

Beispiel
Für die Überschrift wurde hier die serifenlose Schrift Helvetica gewählt. Im Haupttext kommt die Serifen-Schrift Times New Roman zum Einsatz.

Page Layout & Typography IV

- **Use Variations Carefully**
  - CAPITALS require 12% more reading time!
  - Italics and Bold is more suitable
  - Do not use underlines – ugly!

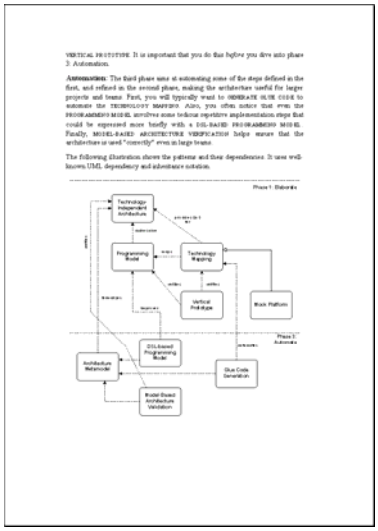
- **Max 3 levels of structure**
  - Chapters, Sections, Subsections
  - Things like 4.1.2.3.4.5 are not useful
- **Use graphical gimmicks** (lines, symbols), but use them sparsely

Page Layout & Typography V

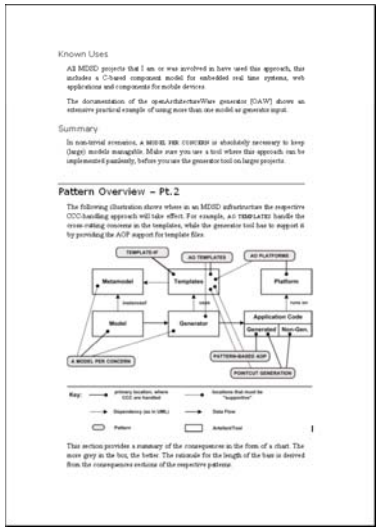
- **Enough Whitespace around illustrations**
  - Make sure illustrations are not jammed in between text
  - Use a different (Sans Serif) font for captions
- **Line Width for Illustrations**
  - Make sure the line width of illustrations is compatible with the weight of the font in the running text
  - Otherwise the illustration will disrupt the layout of the page
- **Spelling is important!**
  - … correct grammar and readable wording is important, too!
  - Short, simple sentences are better.
  - Consider the document literature! Write a book!
- **Use Active Voice!**
  - Talk to the reader: it is easier and more engaging to read!

Page Layout & Typography VI (Line Width for Illustr.)

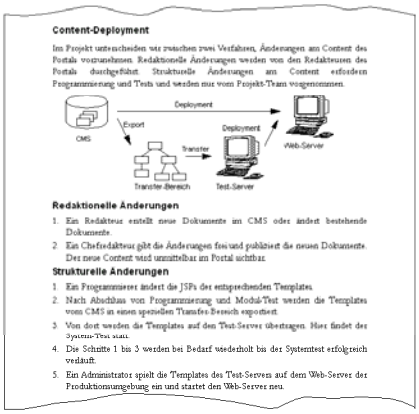
Bad:



Good:



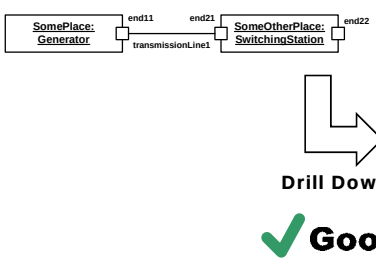
Examples

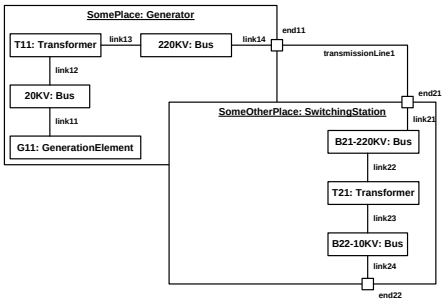


CONTENTS

Diagramming Guidelines

- **Limited Real Estate**
  - Diagram should be viewable on a screen
  - printable on a sheet of paper (Letter, DIN-A4)
  - 7 ± 2 boxes/entities
- **Hierarchical Decomposition** (with Drill-Down diagram)
  - Make sure all elements in a specific diagram are the same level in the hierarchy





Diagramming Guidelines II

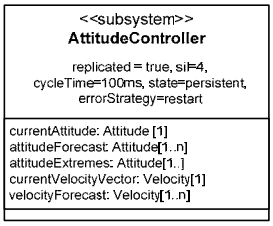
- **Always explain diagrams,** the picture itself is not enough
  - Give it a half-sentence **title**
  - **Explain** in prose **what the diagram shows** (or use the diagram to illustrate conceptes explained in the running text)
  - In the explanation **don't explain every detail** (parameters, eg.) shown in the diagram, but help people "find their way" around the diagram
- Provide a **diagram key** (generally: well-defined language)
  - A diagram is only useful if readers can know **what a graphical element means** (boxes and lines do need explanation!)
  - Hence, either provide a **key**, or use a **well-known language** for the diagram

Diagramming Guidelines III

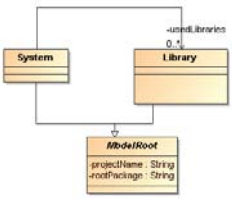
- **Clearly defined "message"**
  - A diagram should have a **well-defined purpose**,
  - Hence, it should typically only **illustrate one concern**, aspect, viewpoint, abstraction level or layer in a hierarchy, relationship kind, ...
  - ... unless it's purpose is to explicitly **illustrate the relationships** of some of these concerns, viewpoints or aspects
- **Readable Left-to-Right or Top-to-Bottom**
  - (most) People naturally scan a diagram from **left to right**, or from **top to bottom**
  - Layout your diagram so it can be read in these orders
  - Especially important if there's some kind of **signal flow**, **time progression** or **increasing level of detail**

Diagramming Guidelines IV

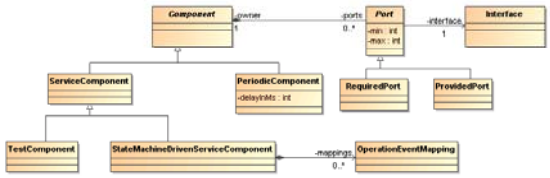
- **Don't add too much text to diagrams**
  - Rather, add these details to separate views, property lists, or render them as graphical elements
- If possible, run the **same kind of relationship** in the **same direction**
  - E.g. inheritance vertical, associations horizontal, dependencies diagonal



X Bad



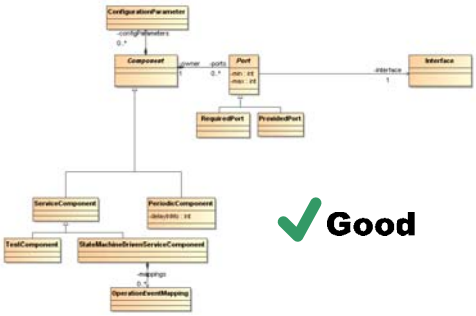
X Bad



Good

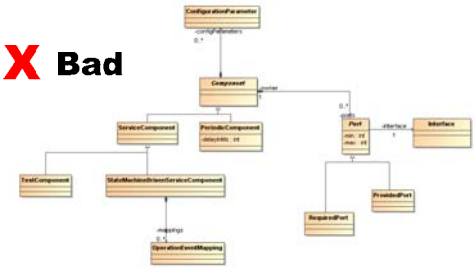
Diagramming Guidelines V

- **Graphical Proximity has meaning**
  - Cohesion
  - Grouping



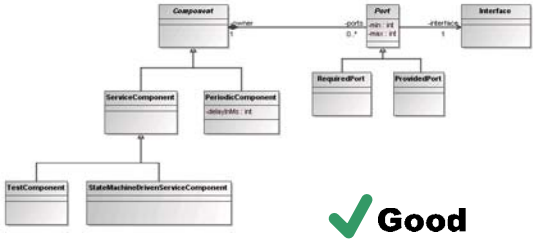
Good

X Bad

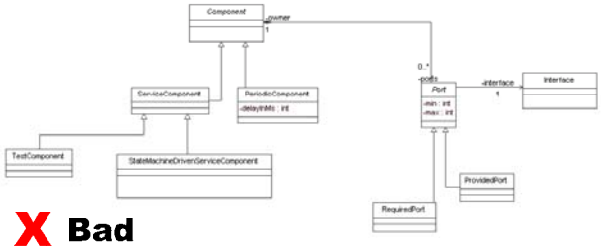


Diagramming Guidelines VI

- **Make it generally nice**
  - As few lines as possible (join/fork lines)
  - Join lines if possible
  - Line Width, Fill Color
  - Use a drawing tool, not a modeling tool!



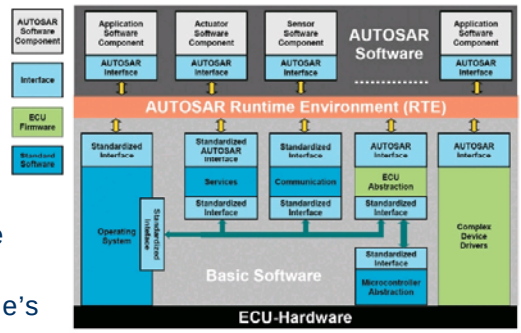
✓ Good



X Bad

Diagramming Guidelines VII

- **Don't imply stuff you don't mean to say**
  - Layers are a good candidate…
- **Use few colors**
  - Every color should have a defined meaning
  - It is part of the language's concrete syntax



✓ Good ? X Bad

Is this a layered architecture?

Diagramming Guidelines VIII

- And finally … don't force diagrams.
- Use diagrams for **what they are good for!**
  - Relationships between things
  - Processing steps (with in/out parameters)
  - Timelines
  - Signal Flow
  - Causality
- There are other ways of rendering things:
  - Tables/Matrices
  - Textual Notations

CONTENTS

Summary

- Software Architecture Documentation is **important** if you want to build a long-standing architecture.
- There are **more** aspects to this **than just a UML model** (which can play a role, but is not sufficient)
- You should use **other channels**, if applicable.
- Make sure that whatever channel you use, it is **executed well**, so that your audience likes to read/listen to/view it.
- In many ways, documenting software architectures can even be **fun**!

THANKS!