Software Architecture Documentation in the Real World

OOPSLA 2007 Tutorial

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

About me

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

Markus Völter

voelter@acm.org

www.voelter.de

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?
  - Specifics for Product Lines & Platforms
- 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?
  - Specifics for Product Lines & Platforms
- 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.

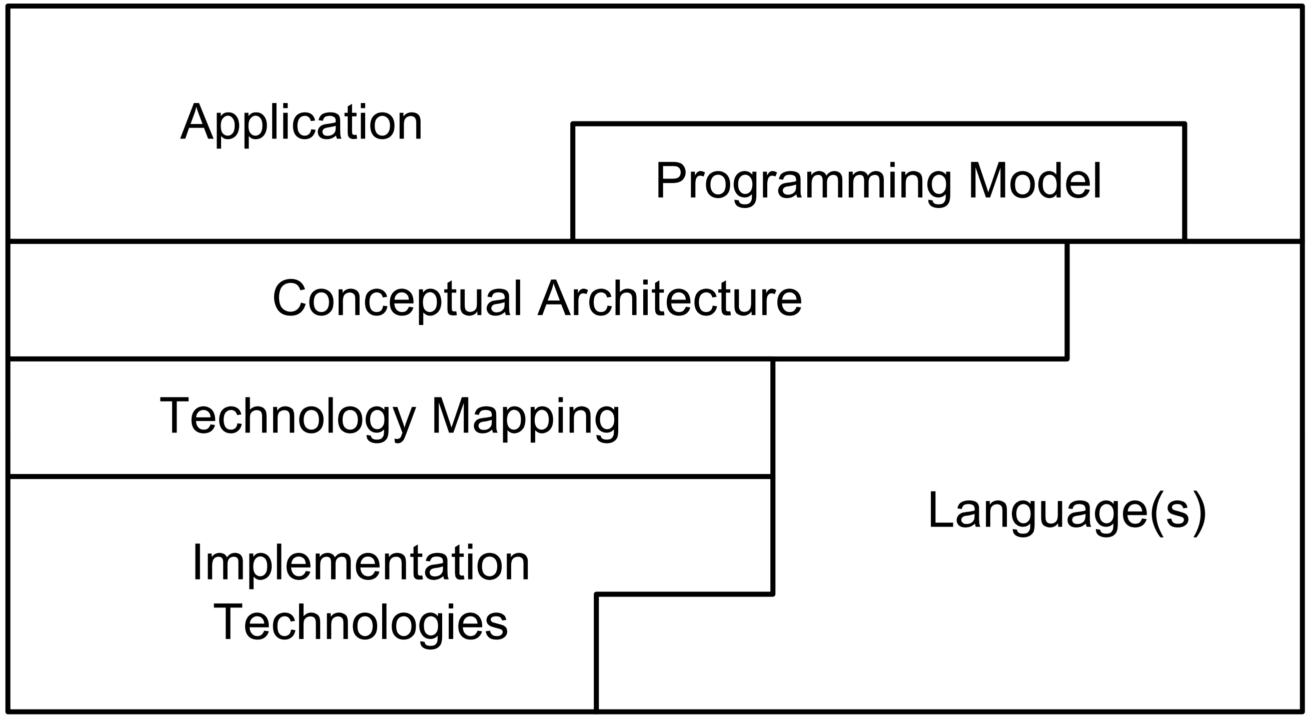
- **Mine:**
  Everything that needs to be consistent throughout a software system

Architecture / System Categories – Focus

- **Small, ad-hoc systems** typically developed by small teams
- **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.



Application vs. Conceptual Architecture

- Any non-trivial, well-architected system typically consists of many instances of a **limited set of concepts.**
  - Components & Connectors, Pipes & Filters, Layers, etc.
  - Architectural Patterns & Styles are good starting points
- We call these limited set of concepts and their relationships the **conceptual architecture**
- The concrete instantiation of these concepts used to build a specific application is called the **application architecture**
- A well-defined **conceptual architecture is essential** for large systems and product lines – to make sure the system(s) is/are
  - internally consistent
  - understandable
  - evolvable

Application vs. Conceptual Architecture II: Examples

- **Application Architecture:**

  We want to build an enterprise system that contains various subsystems such as **customer management**, **billing** and **catalogs**. In addition to managing the data using a database, forms and the like, we also have to manage the **associated long-running business processes**.

- **Conceptual Architecture:**

  Core building blocks are **components**, **interfaces**, data **types**, business **processes** and communication **channels**. Communication is **synchronous** and **local**. Communication to/from processes is **asynchronous** and **remote**. Components are **deployed / hosted** in some kind of **container** that takes care of the technical concerns.

Conceptual Architecture vs. Technology Decisions

- A conceptual architecture should be as **independent of specific technology decisions** as possible (POJOs)
  - Technologies include OS, DOC or Messaging Middleware, drivers, UI frameworks
  - We do not aim to abstract away languages or paradigms
- The mapping to a specific technology (or several technologies) should be **specified in a separate step**
- The mapping should be **guided by non-functional and operational requirements** that are specified as part of the conceptual architecture
- This approach is essential to make sure the technological aspects are **well isolated:**
  - to be able to **exchange** some of the technologies
  - to **simplify application development** by isolating it as far as possible from the details of the technologies

Conceptual Architecture vs. Technology Decisions: Ex.

- Components are implemented as **stateless session beans** with **local interfaces** only.
- Processes are implemented as **message driven beans**; messaging is implemented via **a JMS implementation**.
- Data structures and process state are persisted into a **relational database** using **JPA-based persistence.**
- We use JBoss as the **J2EE container** to host the application components.
- Oracle is used as the **database**.

Conceptual Architecture vs. Programming Model

- The conceptual architecture and its concrete technological realization **can be quite complex –** in order to satisfy all the (non-functional) requirements
- Application developers have to be given a **well-defined programming model** that makes application development based on the architecture as straight forward as possible
  - "Make typical cases simple, and exceptional cases possible"
- The programming model should **hide** as much of the **technology** as possible – and **make** the conceptual architecture **accessible**
  - It can be seen as the "architecture API"

Conceptual Architecture vs. Programming Model: Example

- How do I write a **component**?
- How do I specify a **process**?
- How do I **instantiate** a data object?
- How do I use **channels** for communication?
- How do I **send events** to a process?
  - How do I pass data along?
- What are the **services** the container will provide for me?
- Which **features** of the **Java** programming language can I not use?

Conceptual Architecture vs. Programming Model: Example II

- A component:

```
public @component class AddressManager
  implements IAddressStore {// provides AddressStore

  private IPersonDAO personDAO;

  public @resource void setPersonDAO( IPersonDAO d ) {
    this.personDAO = d;       // setter for dao
  }                           // interface

  public void addOrUpdateContact( Person p ) {
    ...                     // from IAddressStore
  }

  public void addAddress( Person p, Addr
    ...                     // from IAddre
  }

  public Address[] getAddresses( Person
    ...                     // from IAddre
  }
}
```

- A process comp't:

```
public @process class PaymentProcess
                implements IPaymentProcessTrigger {

  private ICustomerManager custMgt;

  public @resource void setCustomerManager(
                    ICustomerManager mgr ) {
    this.custMgr = mgr;
  }

  public @trigger void paymentMade( int procID ) {
    PaymentProcessInstance i = loadProcess( procID );
    if ( amountCorrect() ) {
      // advance to another state…
    }
  }

  public @trigger void paymentTimeout( int procID ) {
    PaymentProcessInstance i = loadProcess( procID );
    ... send reminder using the custMgr ...
  }
}
```

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

- 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?
  - Specifics for Product Lines & Platforms
- Layout and Typography
- Diagramming Guidelines
- Summary

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

- **Visualize!** ... see later.

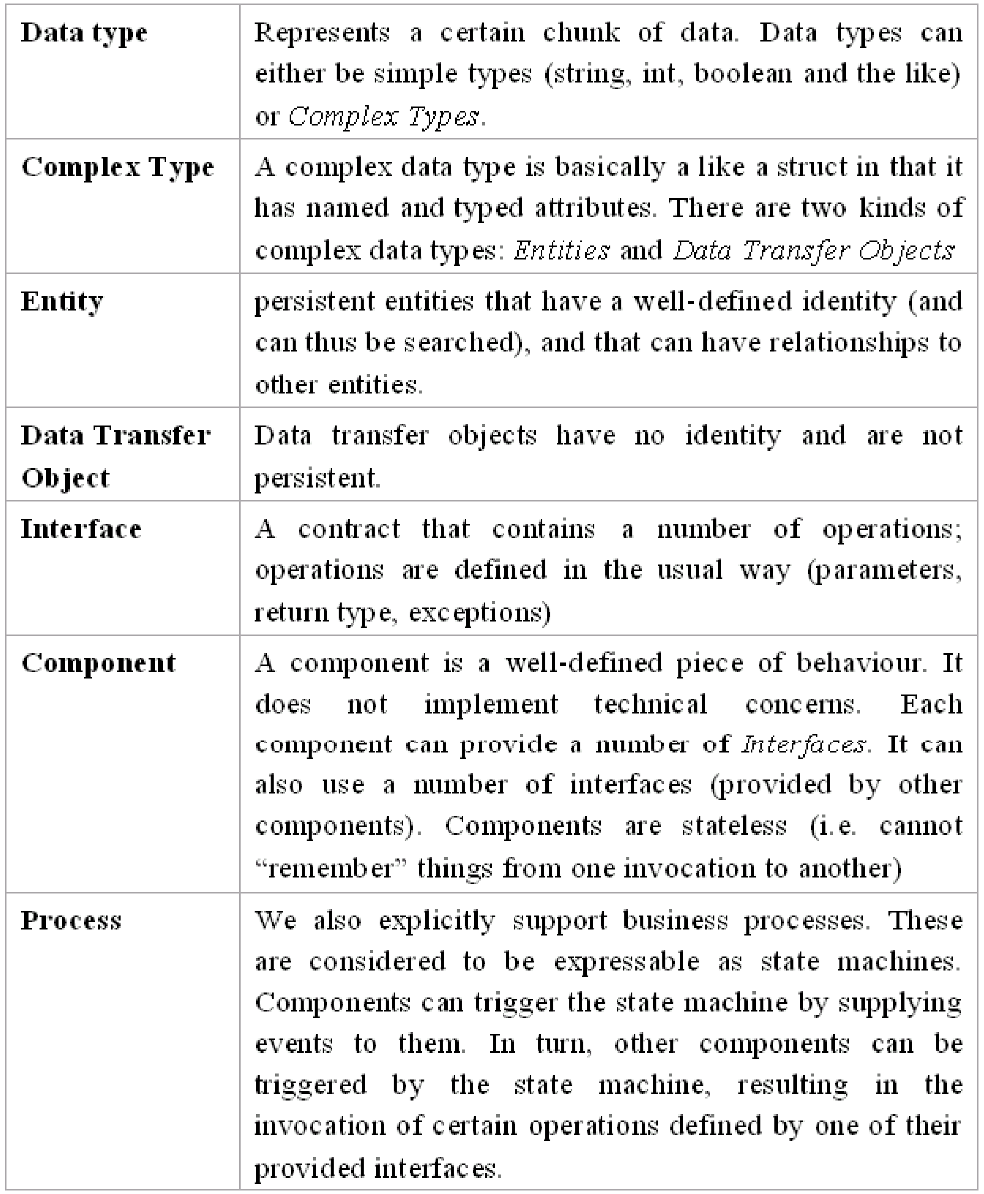
CONTENTS

- 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?
  - Specifics for Product Lines & Platforms
- Layout and Typography
- Diagramming Guidelines
- Summary

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

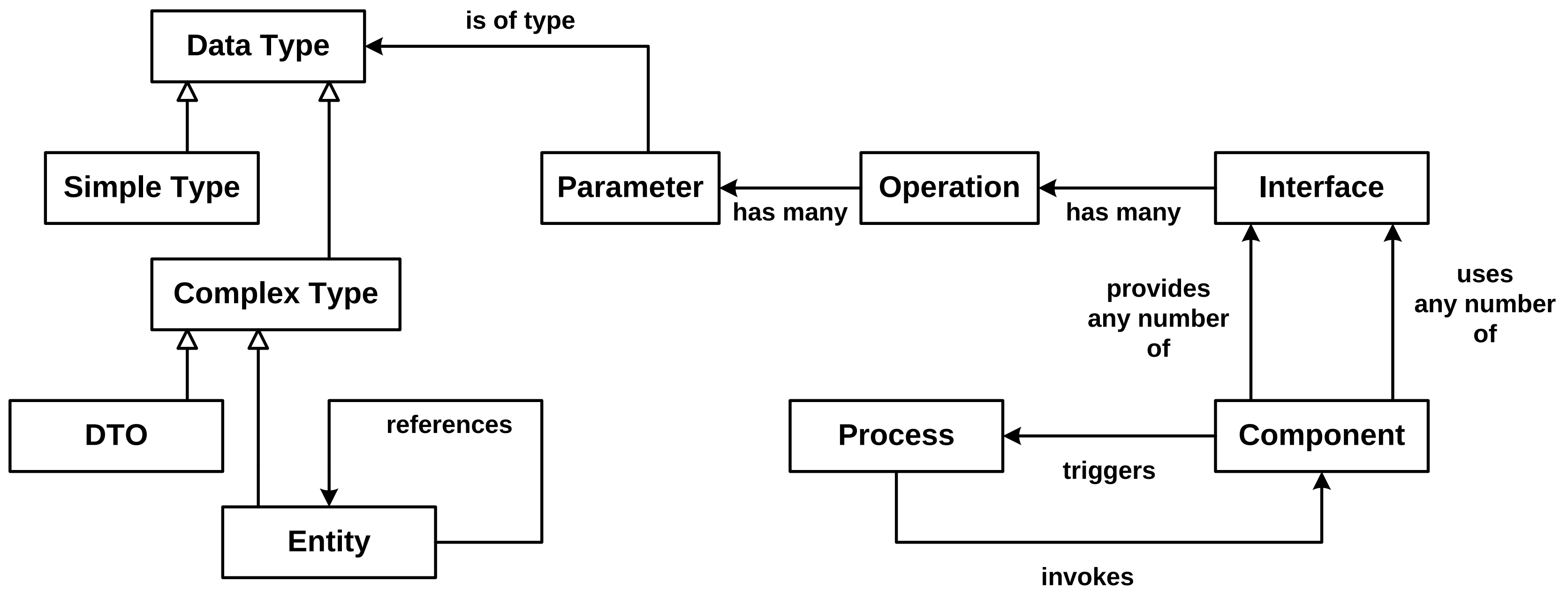
- Represents the core concepts as a diagram, **highlighting the relationships** between the concepts

- **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?
  - Specifics for Product Lines & Platforms
- Layout and Typography
- Diagramming Guidelines
- Summary

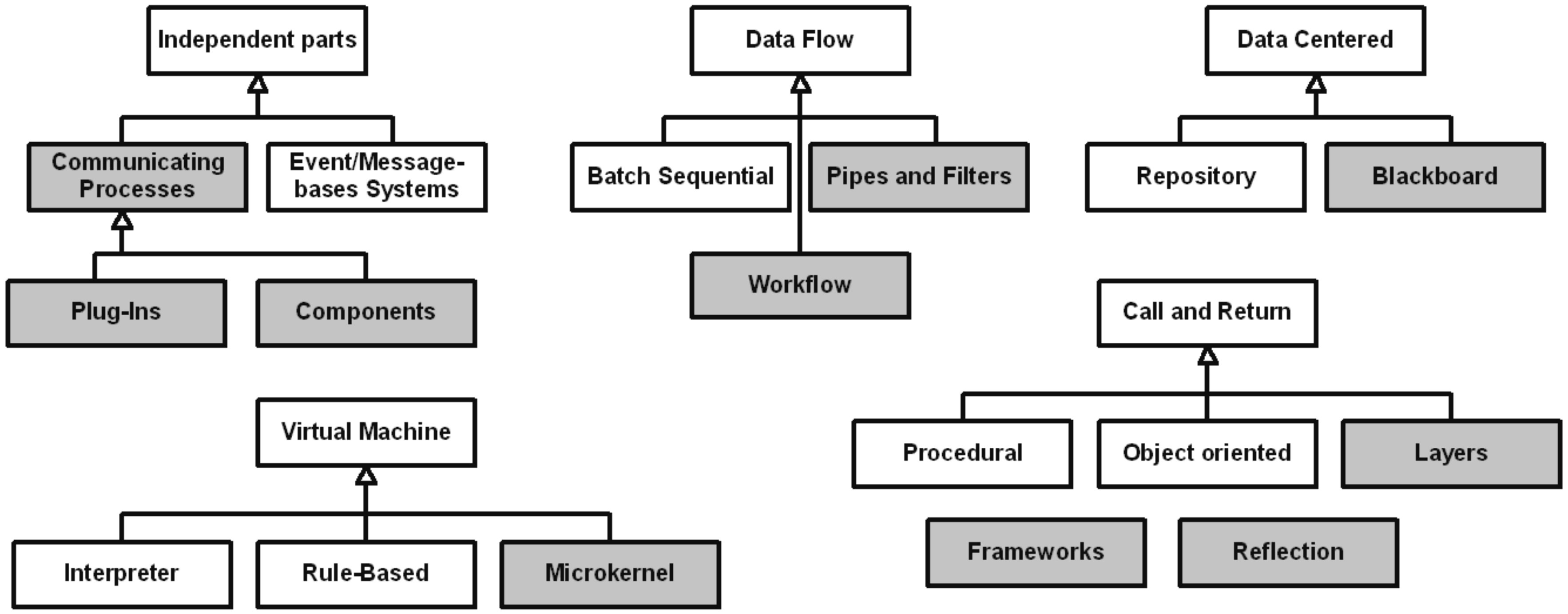
Referencing Patterns

- 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

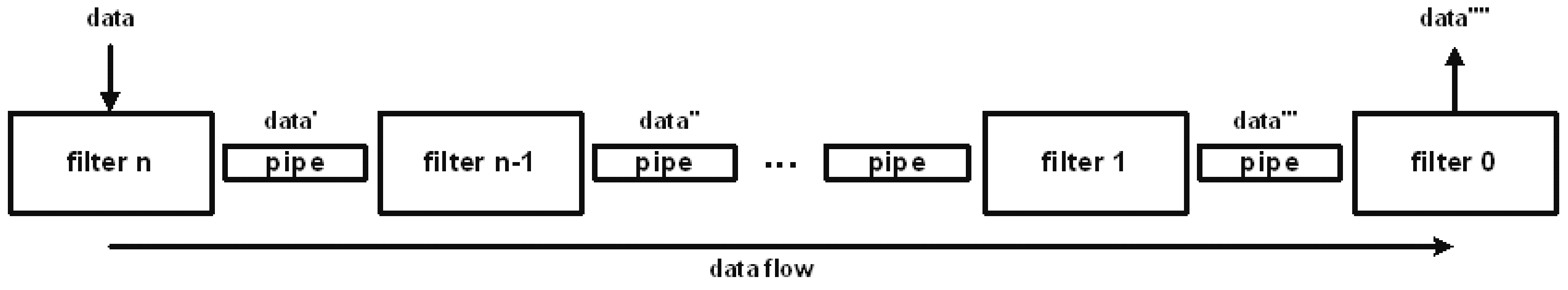
- Architectural patterns can be used to **describe well-working architectural styles** and blueprints.
- Many have been described in the POSA series books, for example, specifically in [POSA1].
- Examples include
  - Blackboard
  - Pipes and Filters
  - Microkernel
  - Components & Connectors
- Many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

Architectural Patterns and Styles; Overview



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

- 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?
  - Specifics for Product Lines & Platforms
- Layout and Typography
- Diagramming Guidelines
- Summary

The challenge of documenting complex architectures

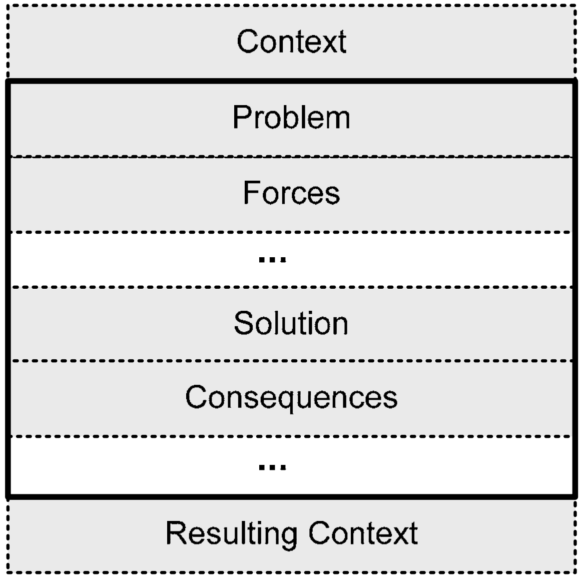
- It is **not enough to simply collect** descriptive data about an architecture
  - e.g. a big UML model or a collection of diagrams or APIs
- rather, communicating an architecture requires a **well-defined, didactic approach**, where
  - You start with a **motivation** of what the general problem is (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

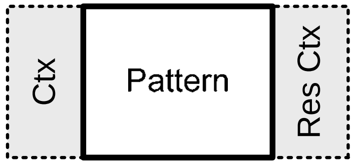
- **Pattern Languages** are collections/sequences of patterns that describe a “whole”,
  - The overall structure of the system is too complicated to be described in one step – thus the language.
  - Sometimes there are alternative sequences through the pattern language describing various alternatives of the “whole”
  - **Group** patterns into **chapters** to implement the layers/levels/rings mentioned before
- A pattern language thus describes **how to build** such 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

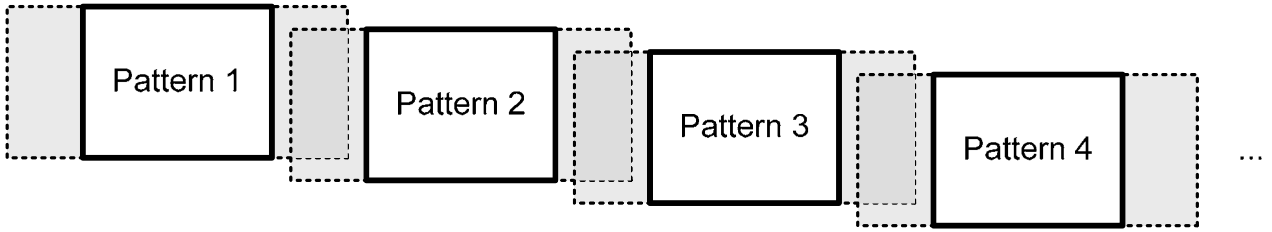
From Patterns to Pattern Languages

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



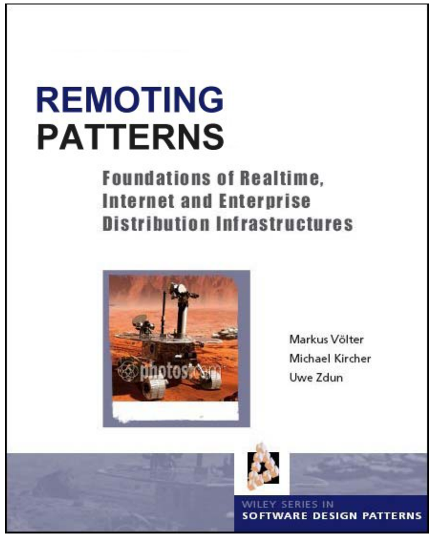
- Pattern Languages are built by having subsequent patterns **solve problems that arise from using a previous pattern.**

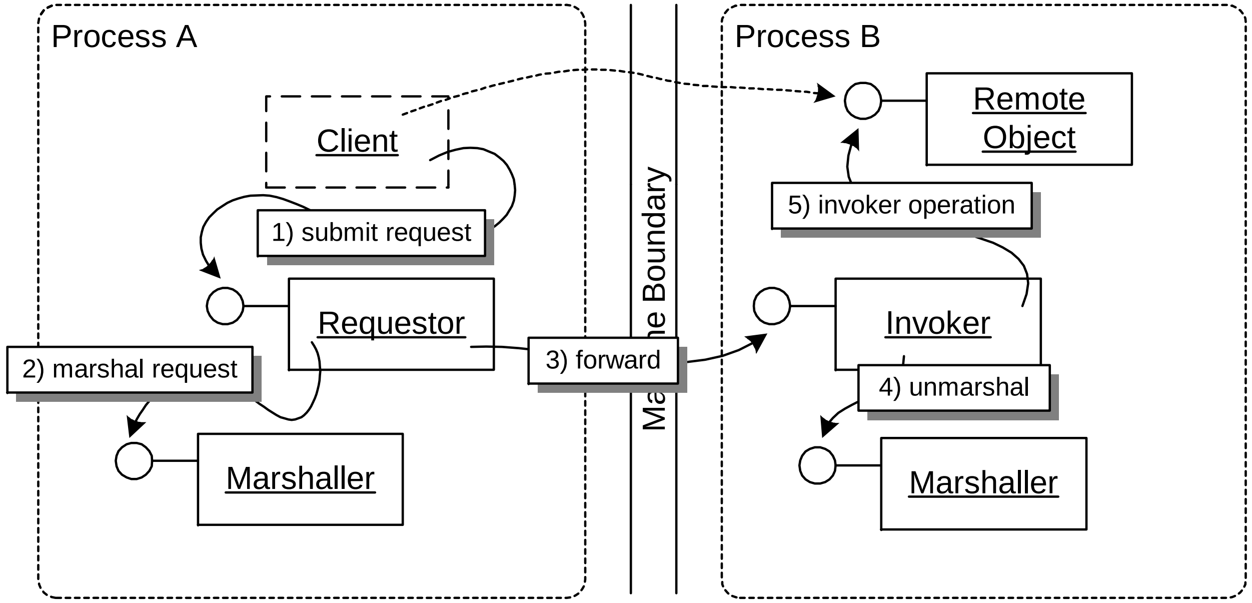




Example: Remoting

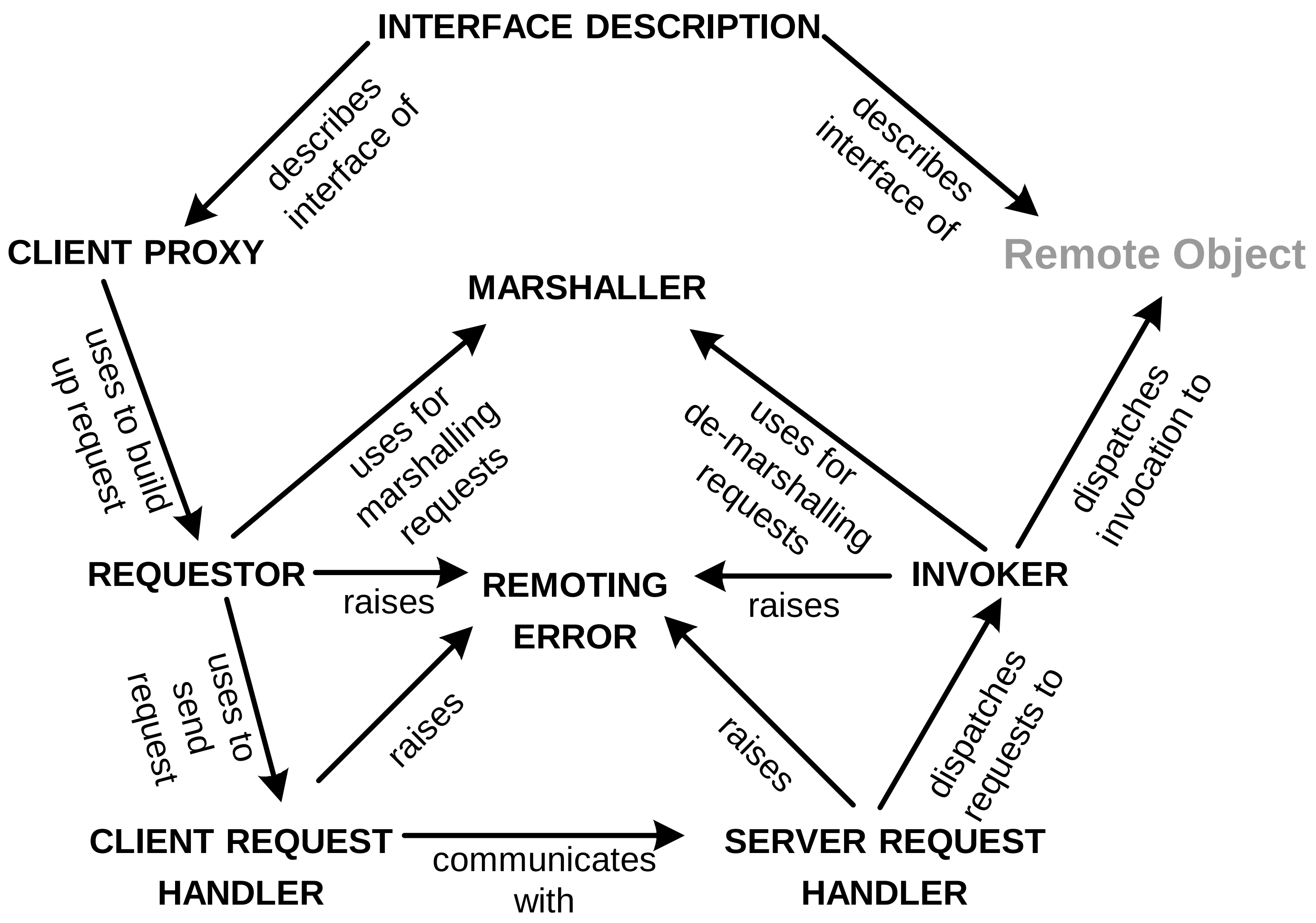
- Describes the internal architecture of **remoting middleware** such as CORBA, WebServices or .NET Remoting
- It can be seen as a pattern language that describes the **internal details of Broker architectures** in industrial practice.


REMOTING PATTERNS
Foundations of Realtime, Internet and Enterprise Distribution Infrastructures
Markus Völter
Michael Kircher
Uwe Zdun
WILEY SERIES IN SOFTWARE DESIGN PATTERNS



Example: Remoting II

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

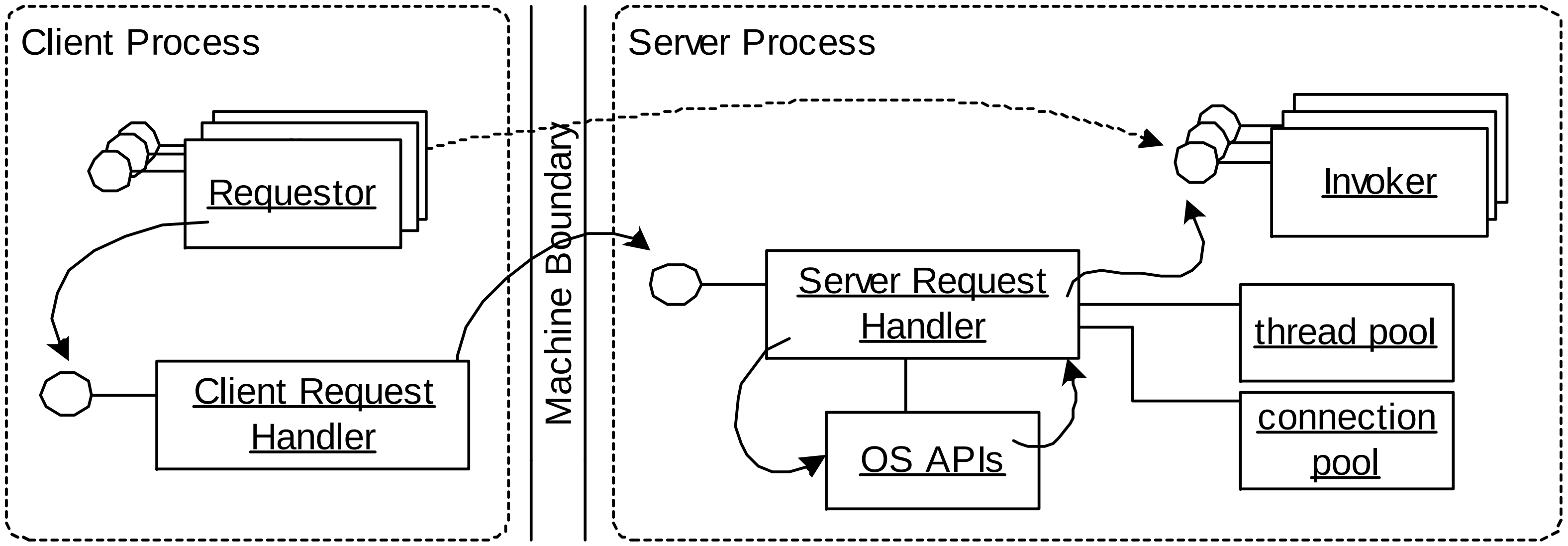


Example: Remoting III: Server Request Handler

- **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, ...)

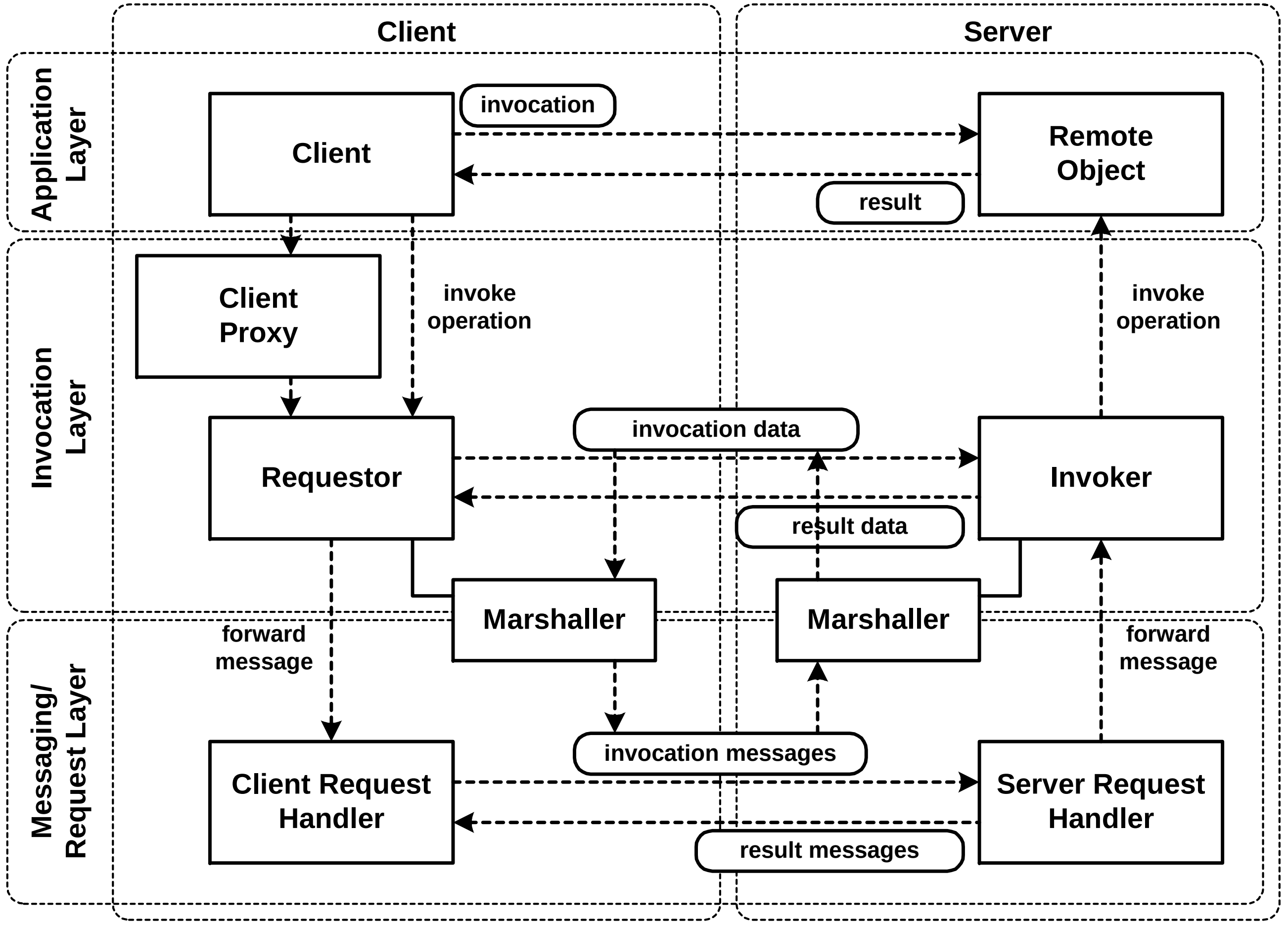
Example: Remoting IV: Server Request Handler 2

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



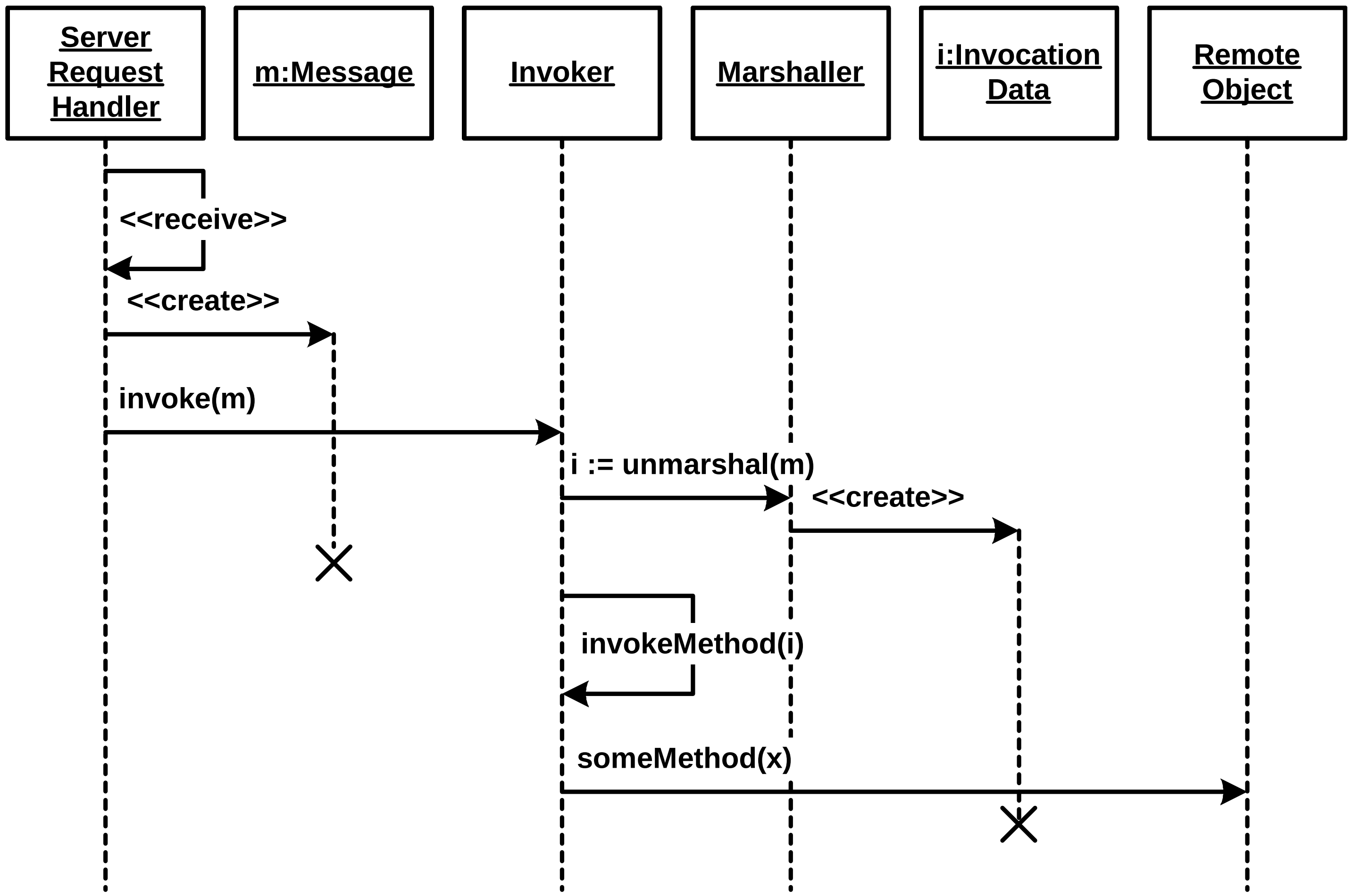
Example: Remoting V

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



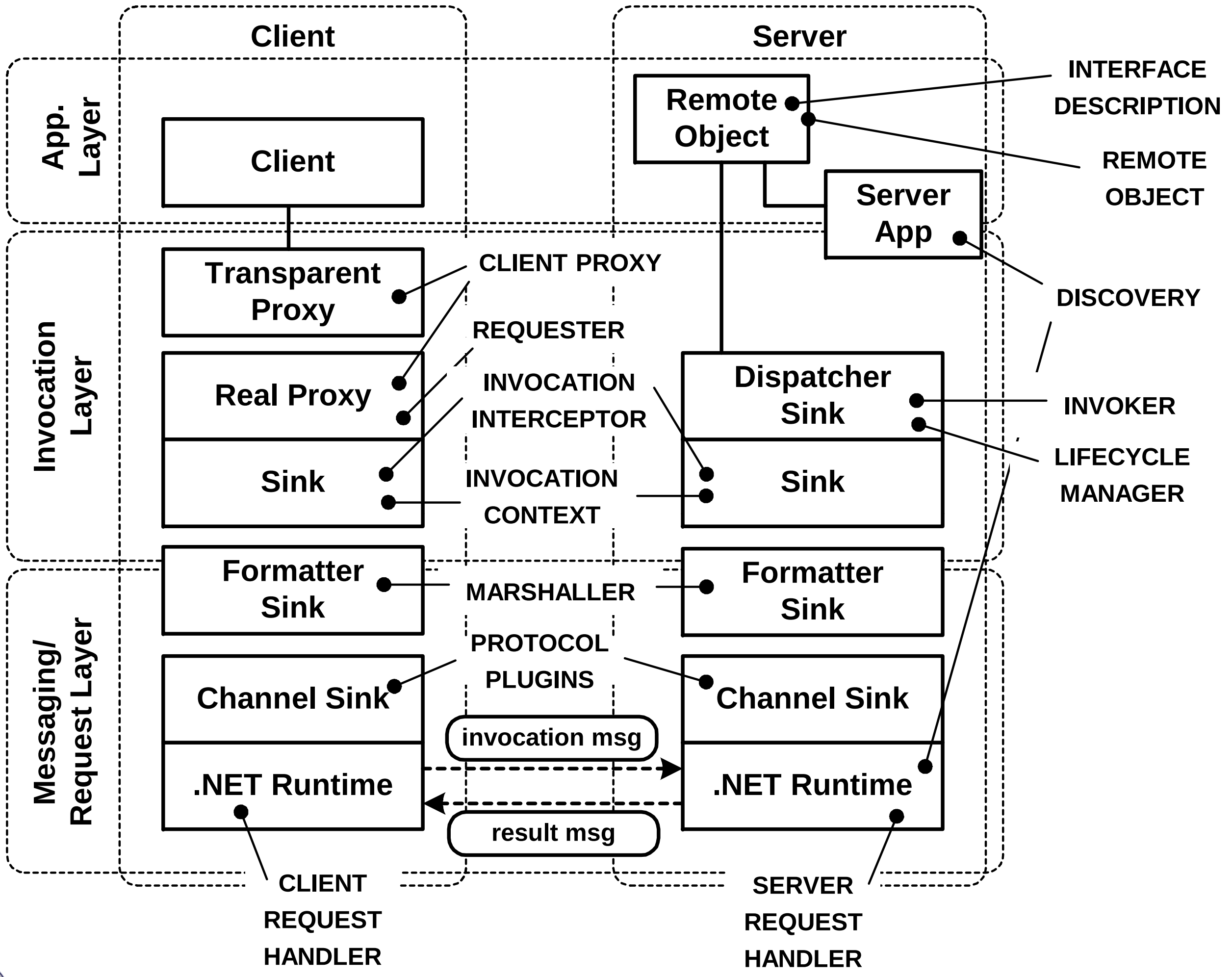
Example: Remoting VI

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



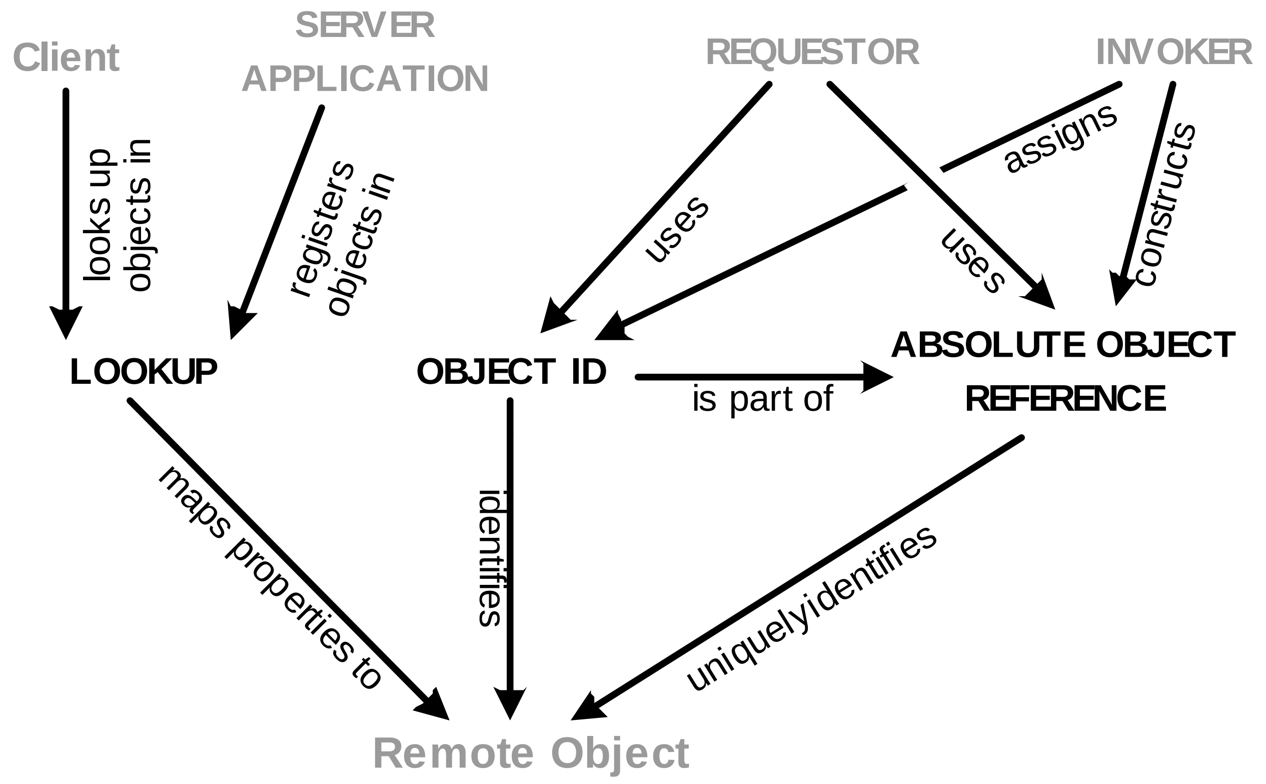
Example: Remoting – Technology Projection; .NET Example

- This view **maps the patterns** (general concepts) to a **specific example** (in this case, .NET remoting)



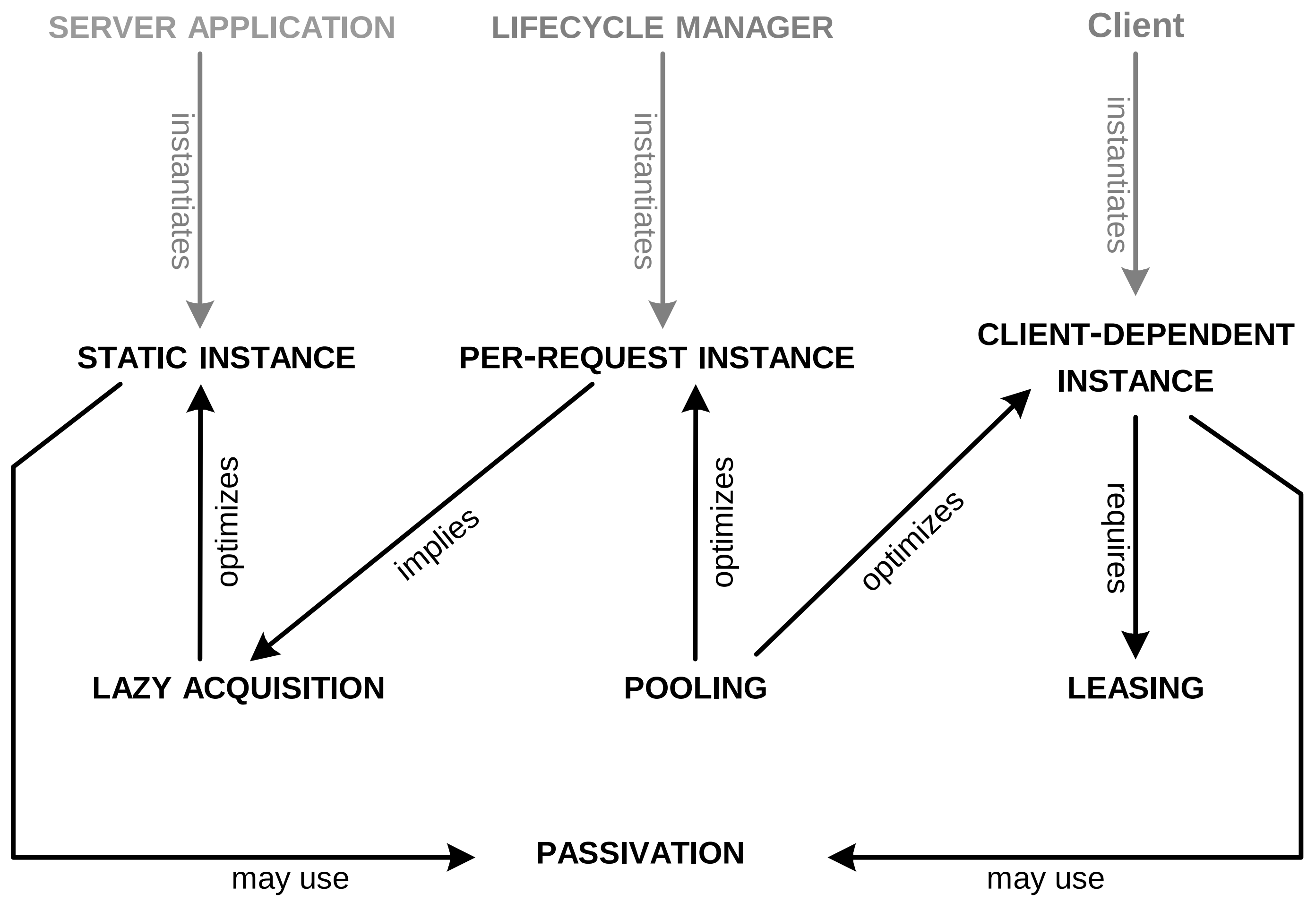
Example: Remoting - Identification

- This additional layer/level/ring explains how **remote objects are identified** – note how we refer to the patterns from the lower layers.



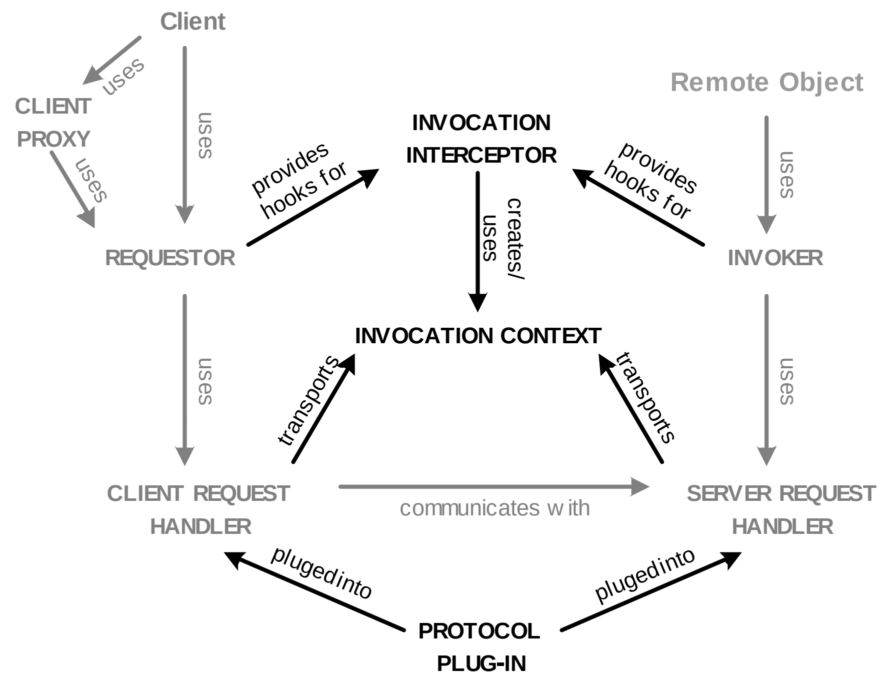
Example: Remoting - Lifecycle

- This layer explains the different **lifecycle patterns** and the associated (de-)activation strategies

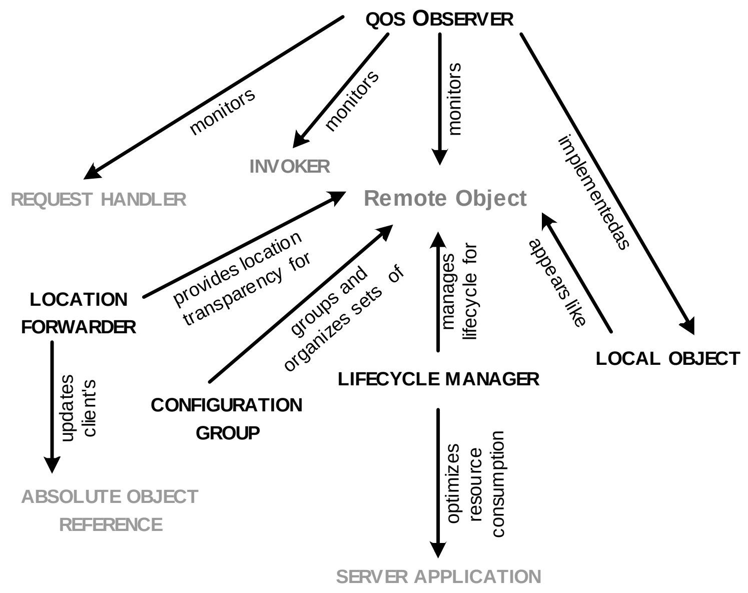


Example: Extension Layers

- Extending the **communication framework** with out-of-band data or cross-cutting functionality



- Extending the **internal infrastructure**



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?
  - Specifics for Product Lines & Platforms
- 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

- 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?
  - Specifics for Product Lines & Platforms
- Layout and Typography
- Diagramming Guidelines
- Summary

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.

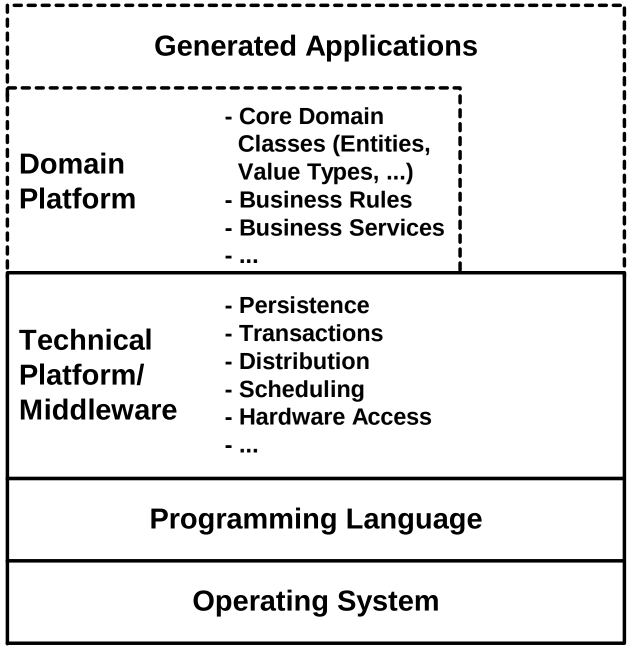
- **Definition III:** (www.evgschool.org/Columbus%20vocabulary.htm)
  A diagram is a drawing, sketch, plan, or chart that helps to make something easier to understand

Models vs. Diagrams

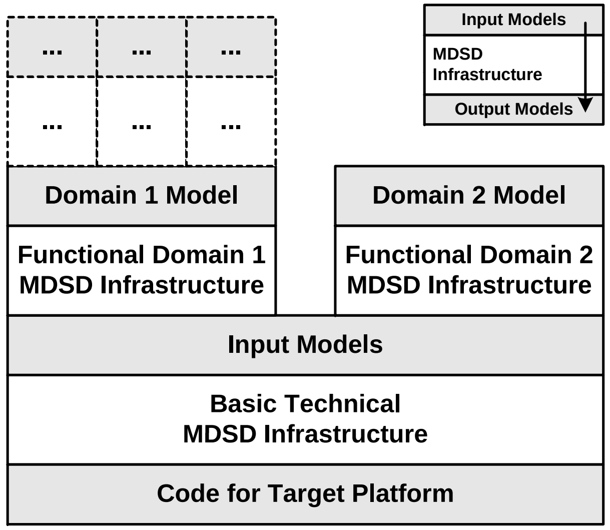
- Diagrams are mainly used to **“intuitively communicate”** something to **humans**
- Models are mainly used to **“formally specify”** something to **tools**
- Hence, models need to be **correct** and **complete** wrt. to the aspect, viewpoint or concern they describe.
  - They need to be based on a well-defined language
- **Diagrams** can be used to **represent models**.
- Models, however, can also be represented in other 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

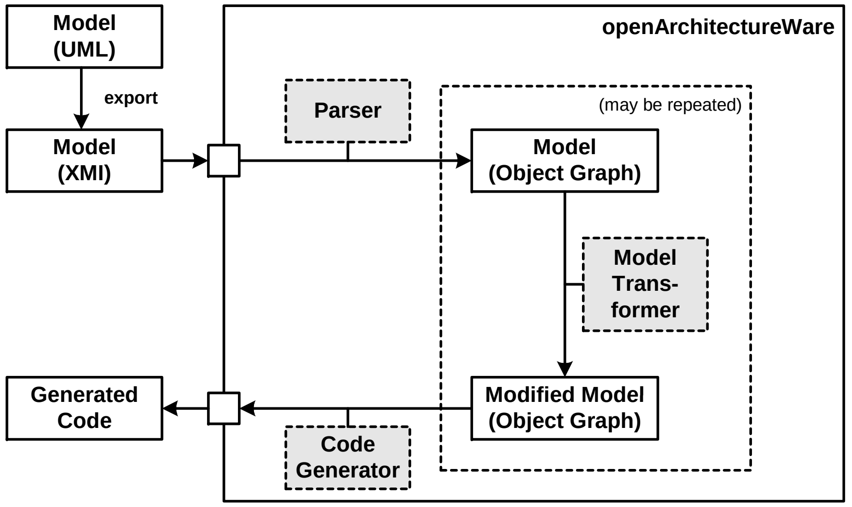


- **Transformation architecture** of a cascaded MDSD application
  - It is built by recursively applying the atomic building block shown in the top right corner

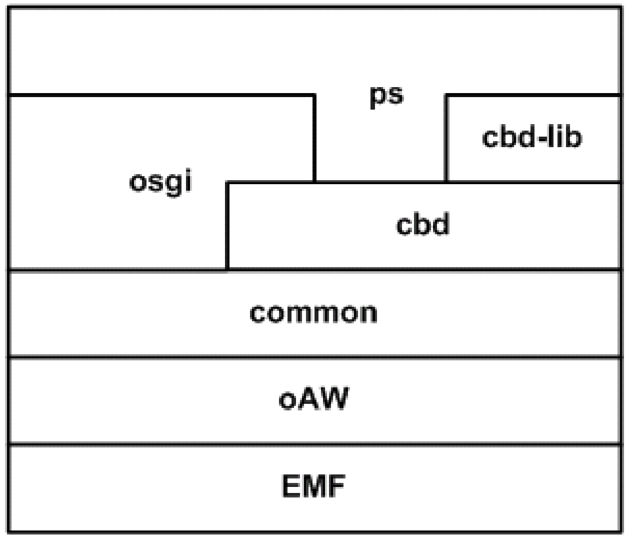

Input Models
MDSD Infrastructure
Output Models
...
Domain 1 Model
Functional Domain 1 MDSD Infrastructure
Domain 2 Model
Functional Domain 2 MDSD Infrastructure
Input Models
Basic Technical MDSD Infrastructure
Code for Target Platform

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

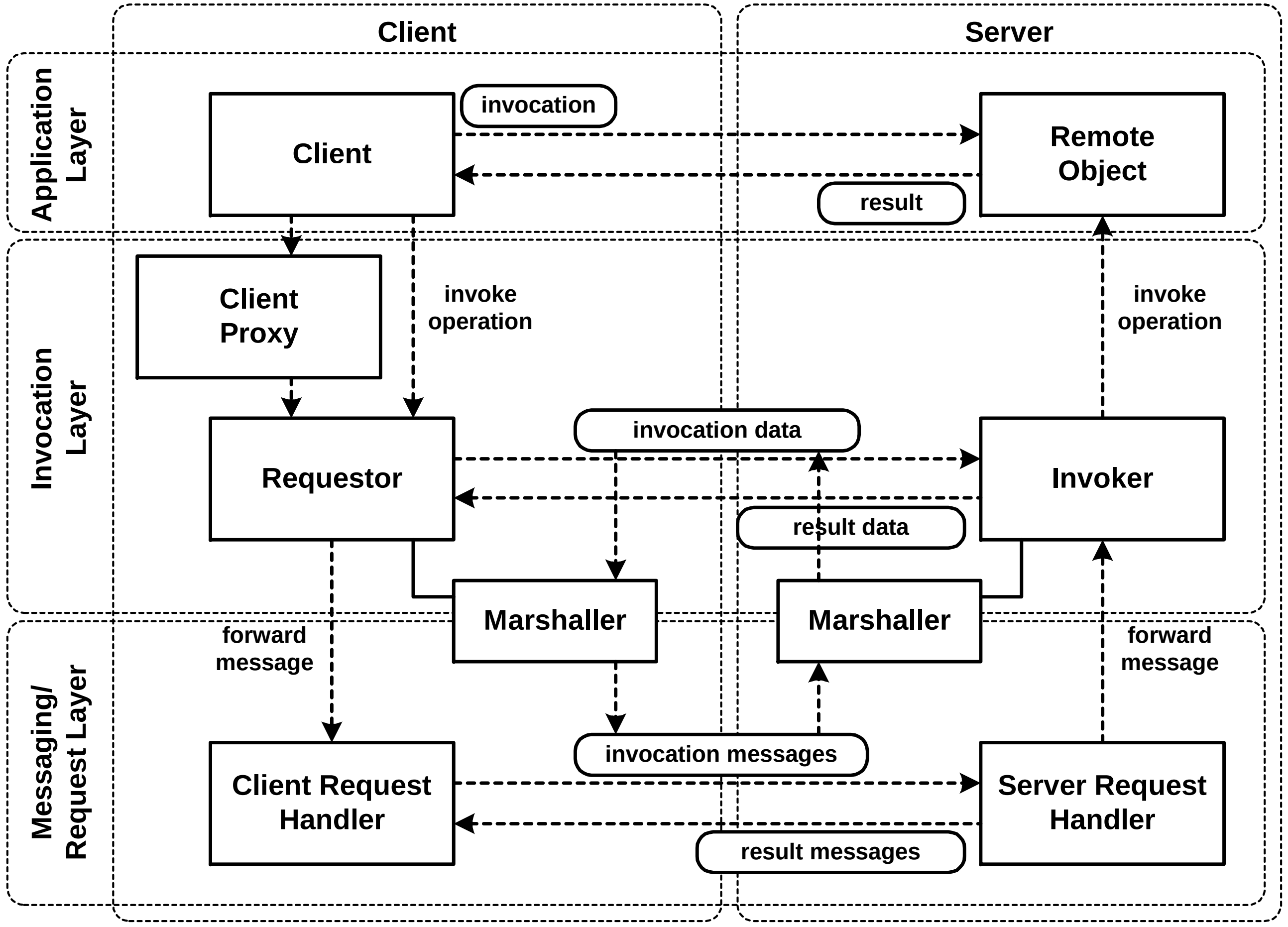


- **Layers** of a product-line architecture
  - If you visually draw layers, make sure this is actually what you want to communicate (i.e. there really is a layering in the system you describe)



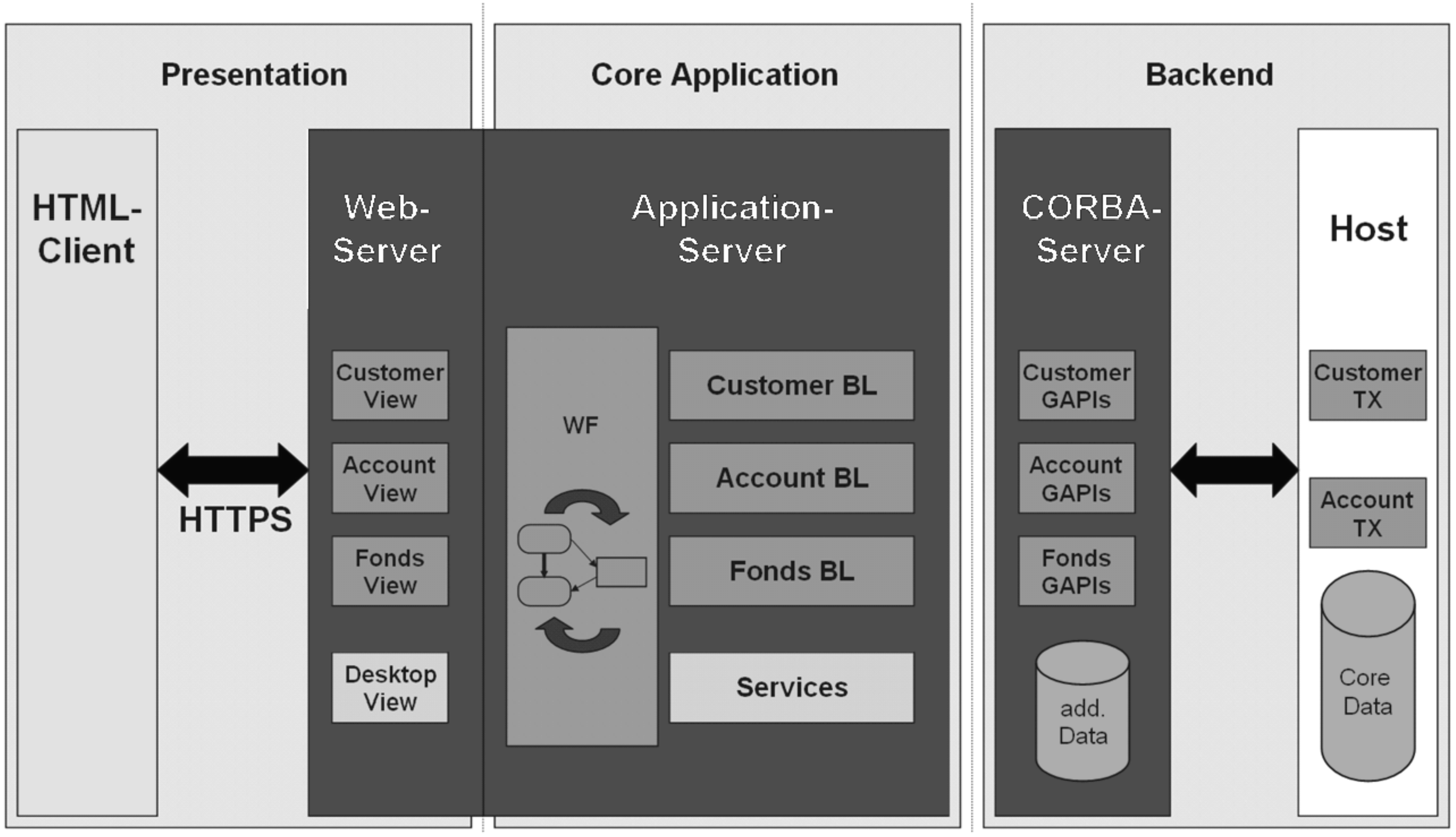
Examples of Architectural Diagrams VII

- This one shows **several aspects**: components, layers, client/server, dependencies, invocations, exchanged data



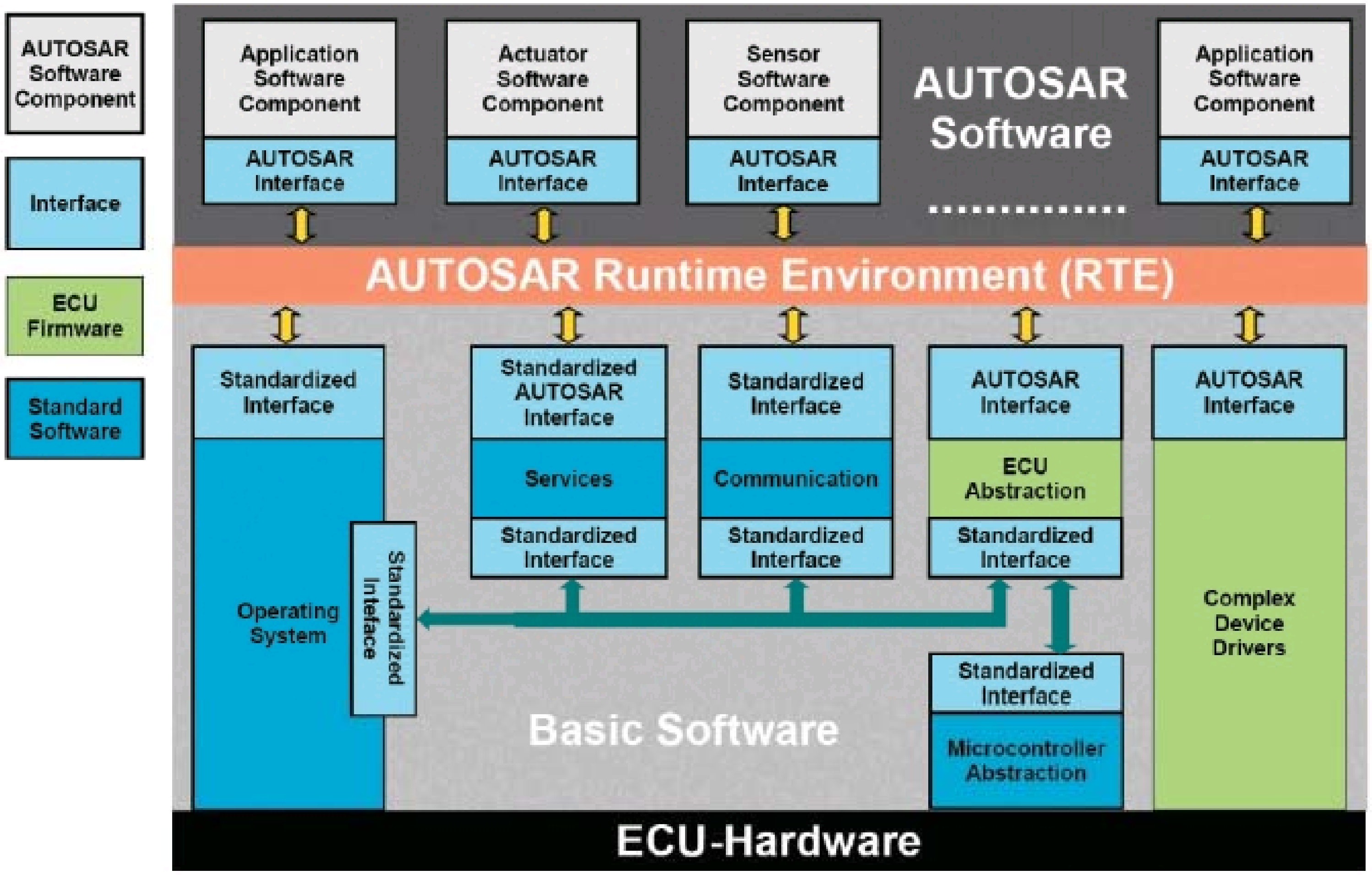
Examples of Architectural Diagrams III

- A three-tier enterprise system. Useful diagram?



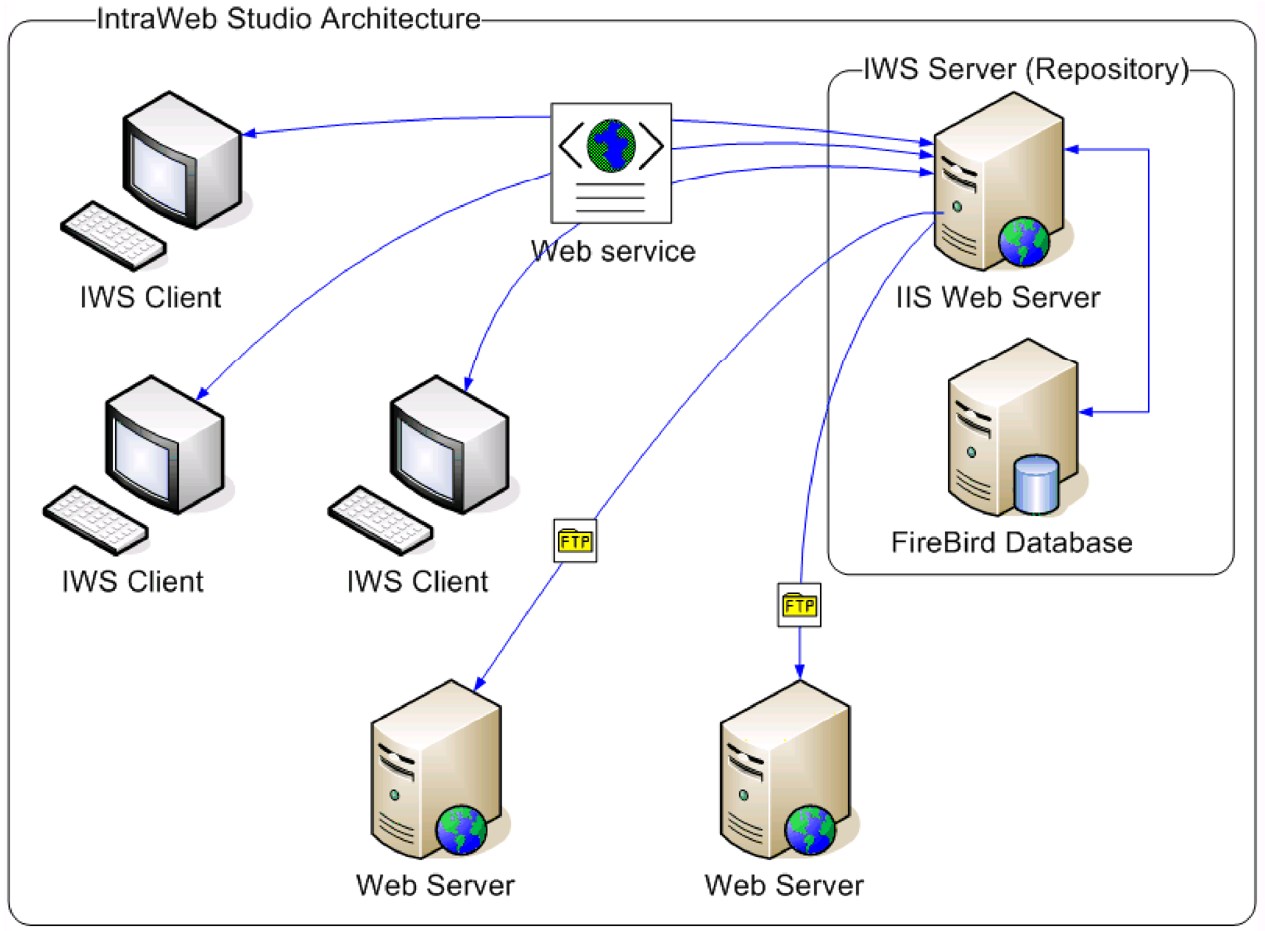
Examples of Architectural Diagrams IV

- The AUTOSAR Architecture. Are the layers really there?



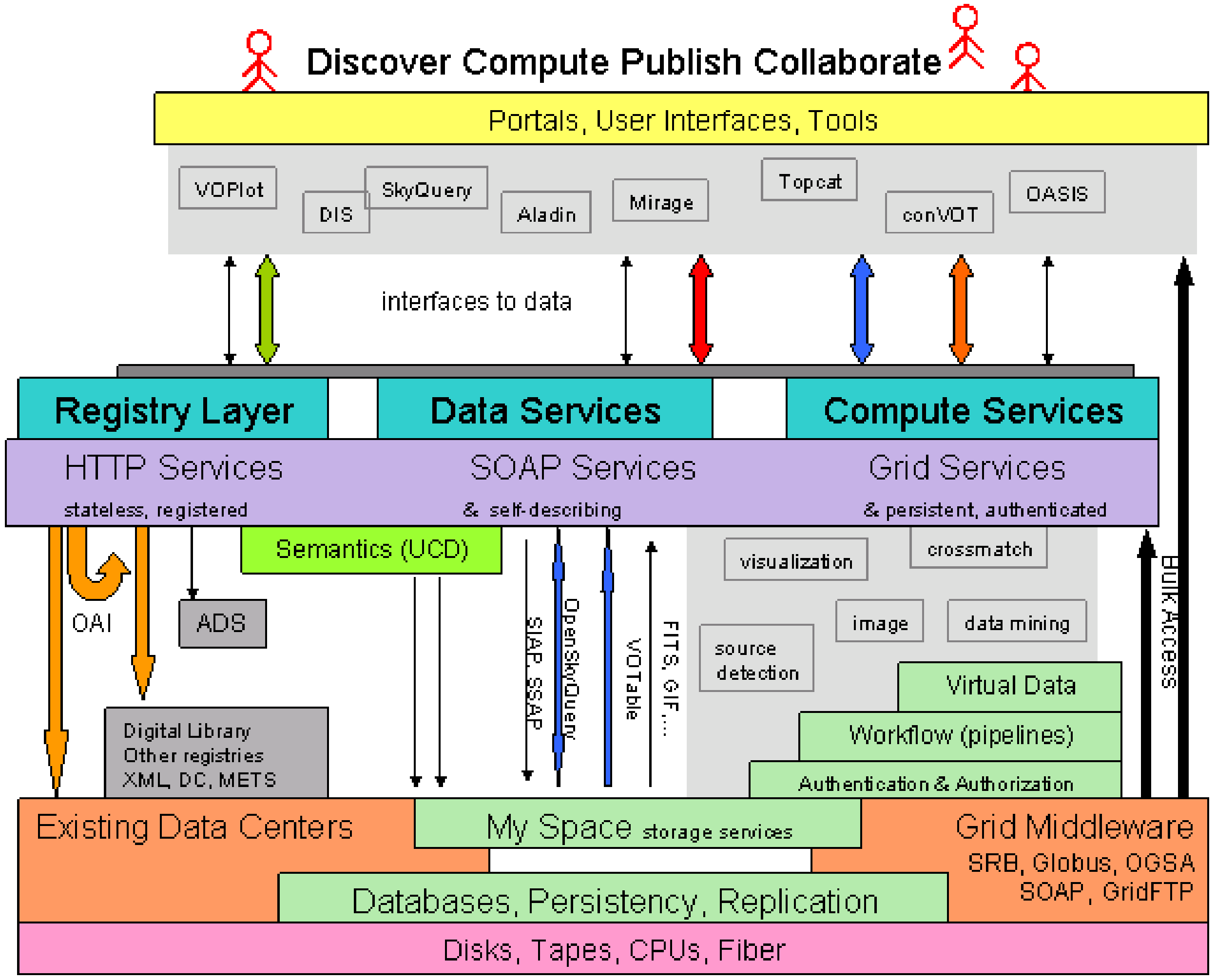
Examples of Architectural Diagrams V

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



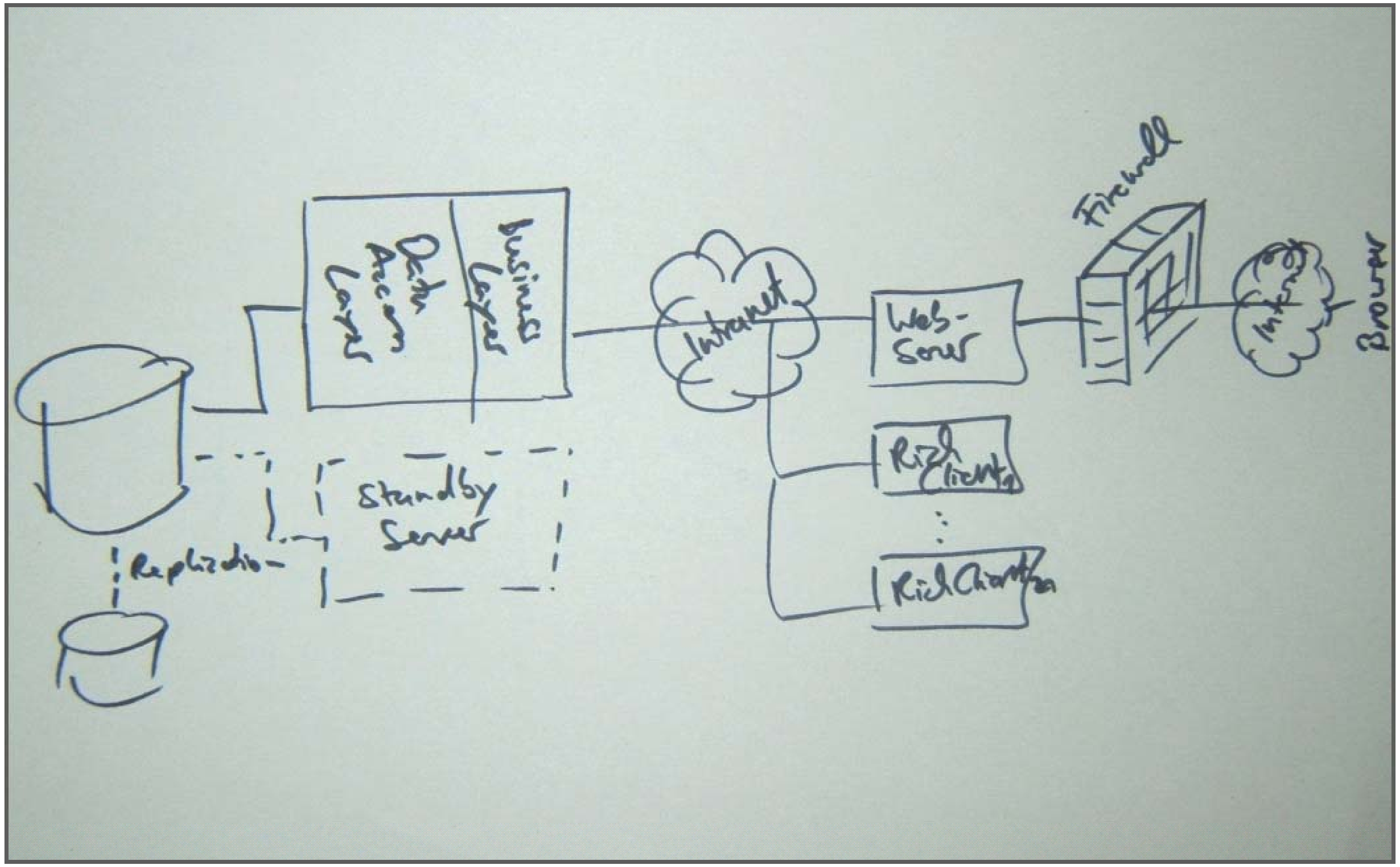
Examples of Architectural Diagrams VI

- 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

A bit more formal: FMC

- What is FMC? (http://www.fmc-modeling.org)

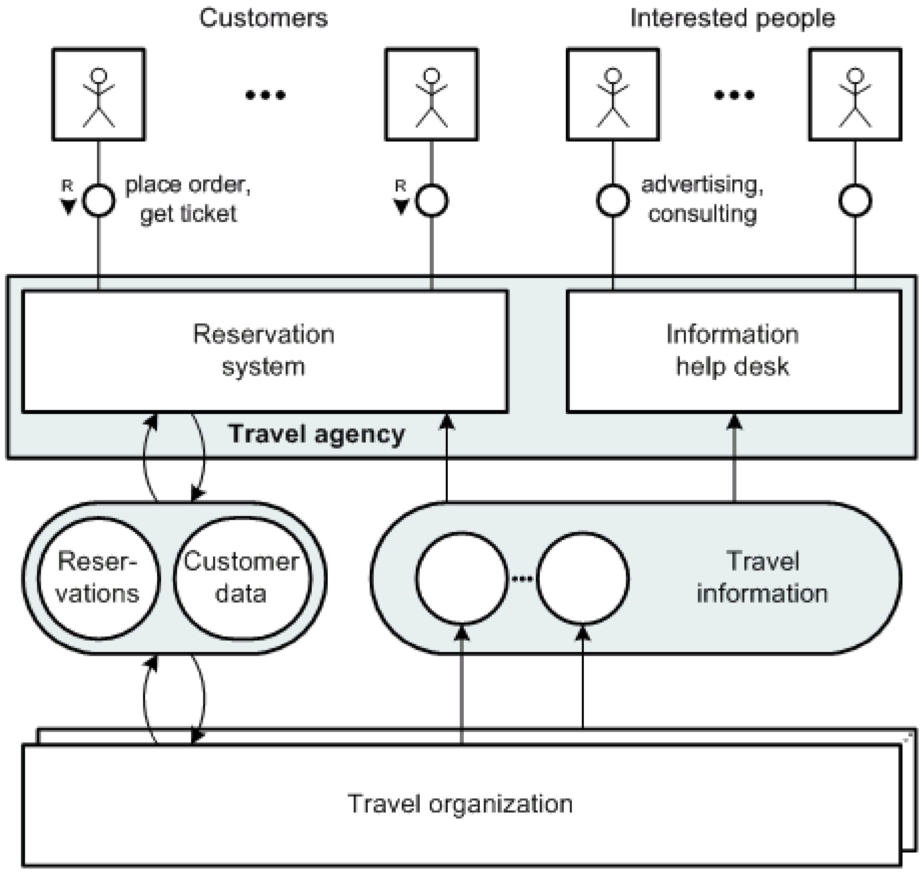
  FMC is the acronym for **Fundamental Modeling Concepts**, a consistent and coherent way to **think and talk** about dynamic systems.

  It enables **people to communicate** the concepts and structures of complex informational systems in an efficient way among the different types of **stakeholders**.

- Developed by the **FMC Consortium** (SAP, Hasso Plattner Institut)

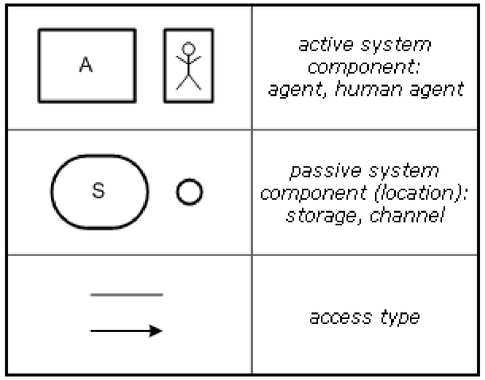
Example FMC Model

- The **Travel Organization** reads and writes various data.
- The travel agency’s **Reservation** system does the same.
- The **Reservation** system and the **Info** help desk only read travel information.
- **Customers** use a request/response scheme to place orders and get tickets
- ...



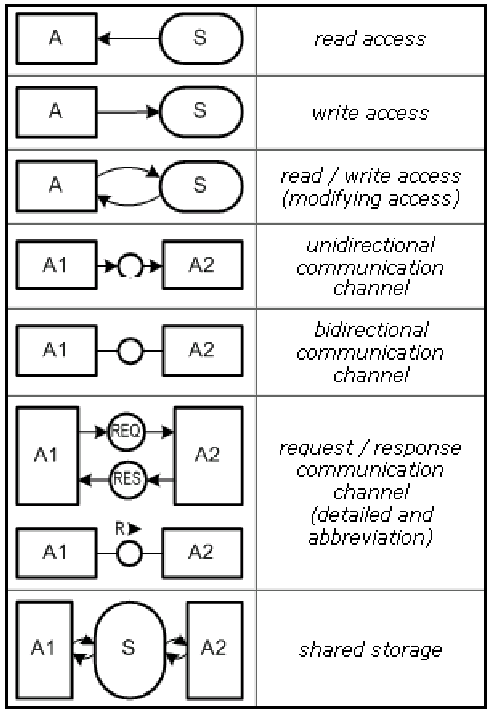
FMC Notation Overview

- Basic Elements



A	active system component: agent, human agent
S	passive system component (location): storage, channel
	access type

Common Structures



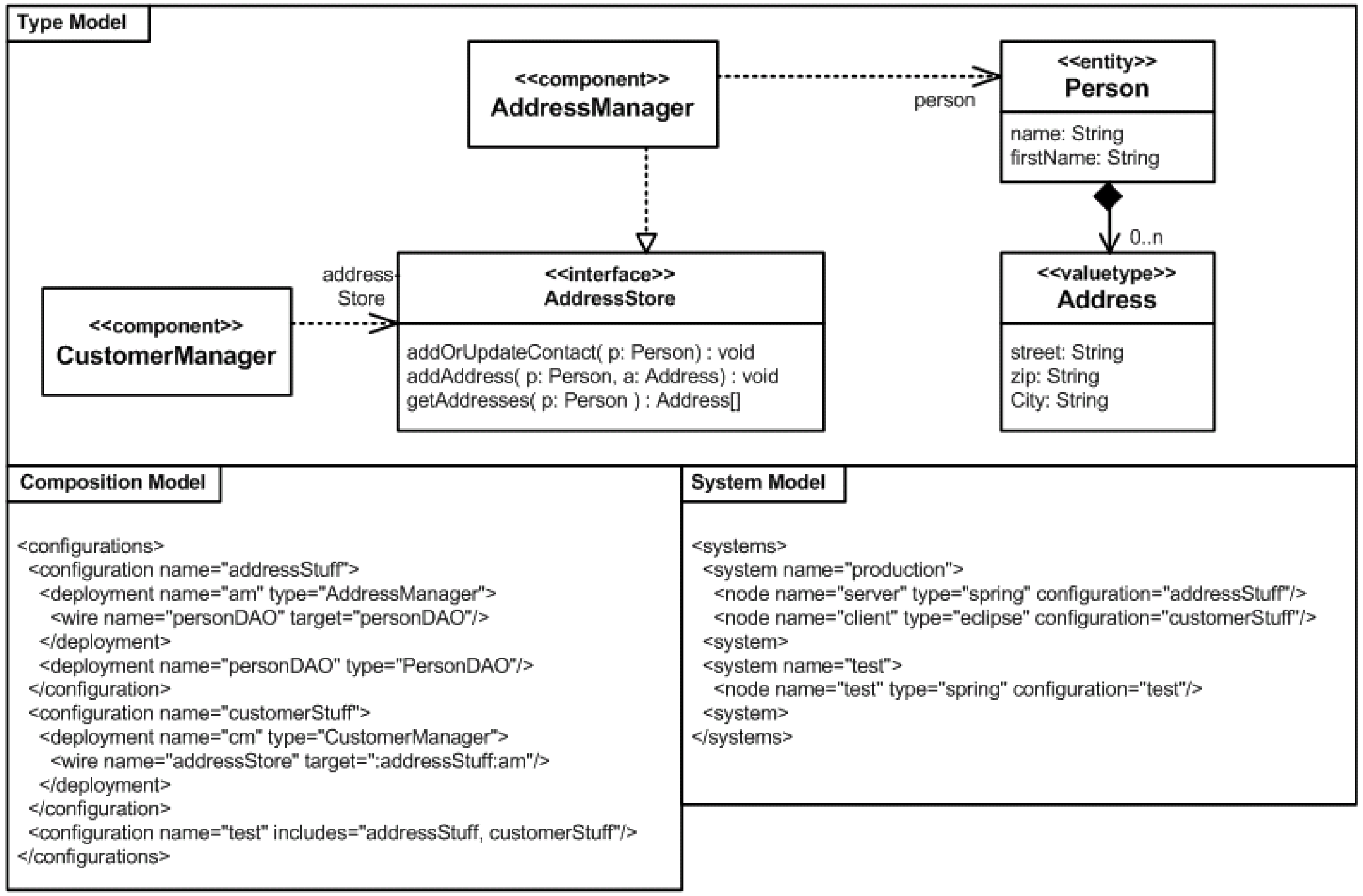
A S	read access
A S	write access
A S	read / write access (modifying access)
A1 A2	unidirectional communication channel
A1 A2	bidirectional communication channel
A1 REQ RES A2 A1 R A2	request / response communication channel (detailed and abbreviation)
A1 S A2	shared storage

- They also have **Petri Nets** for dynamic structures, and **ER Diagrams** for structured data

- They have **Visio Stencils** (which look really good)!

Example of an Architectural Models

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

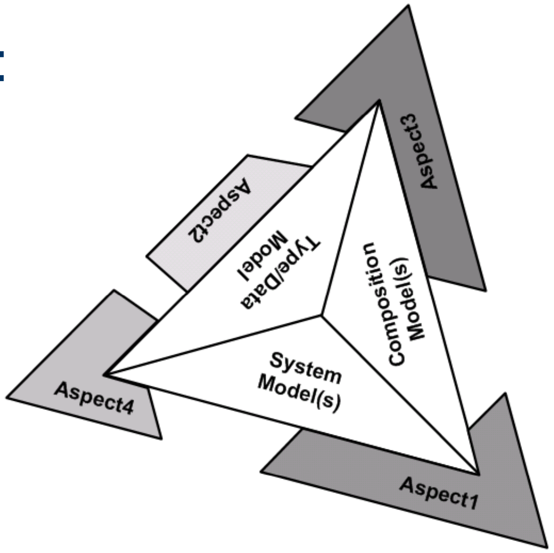


```
<configurations>
  <configuration name="addressStuff">
    <deployment name="am" type="AddressManager">
      <wire name="personDAO" target="personDAO"/>
    </deployment>
    <deployment name="personDAO" type="PersonDAO"/>
  </configuration>
  <configuration name="customerStuff">
    <deployment name="cm" type="CustomerManager">
      <wire name="addressStore" target=":addressStuff:am"/>
    </deployment>
  </configuration>
  <configuration name="test" includes="addressStuff, customerStuff"/>
</configurations>
```

```
<systems>
  <system name="production">
    <node name="server" type="spring" configuration="addressStuff"/>
    <node name="client" type="eclipse" configuration="customerStuff"/>
  <system>
  <system name="test">
    <node name="test" type="spring" configuration="test"/>
  <system>
</systems>
```

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**

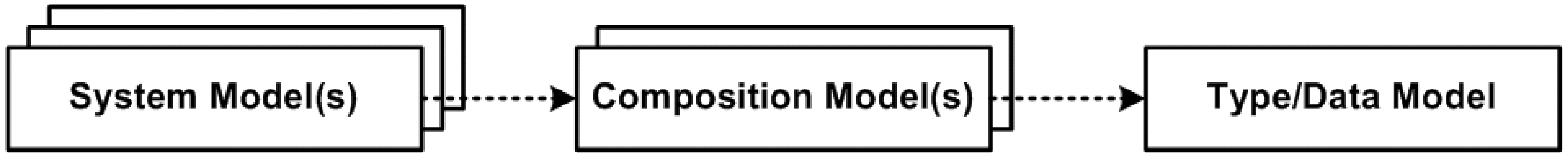
- **Core Views** used to describe the architecture
  - **Logical View:** Functional requirements (e.g. UML diagrams, structural and behavioral)
  - **Process View:** Non-Functional (concurrency, performance, scalability)
  - **Development View:** file layout, project structure, versioning, packaging
  - **Physical View:** topology, communication, deployment

- **+1:** Scenarios (Use Cases)

- Not too much used in practice...

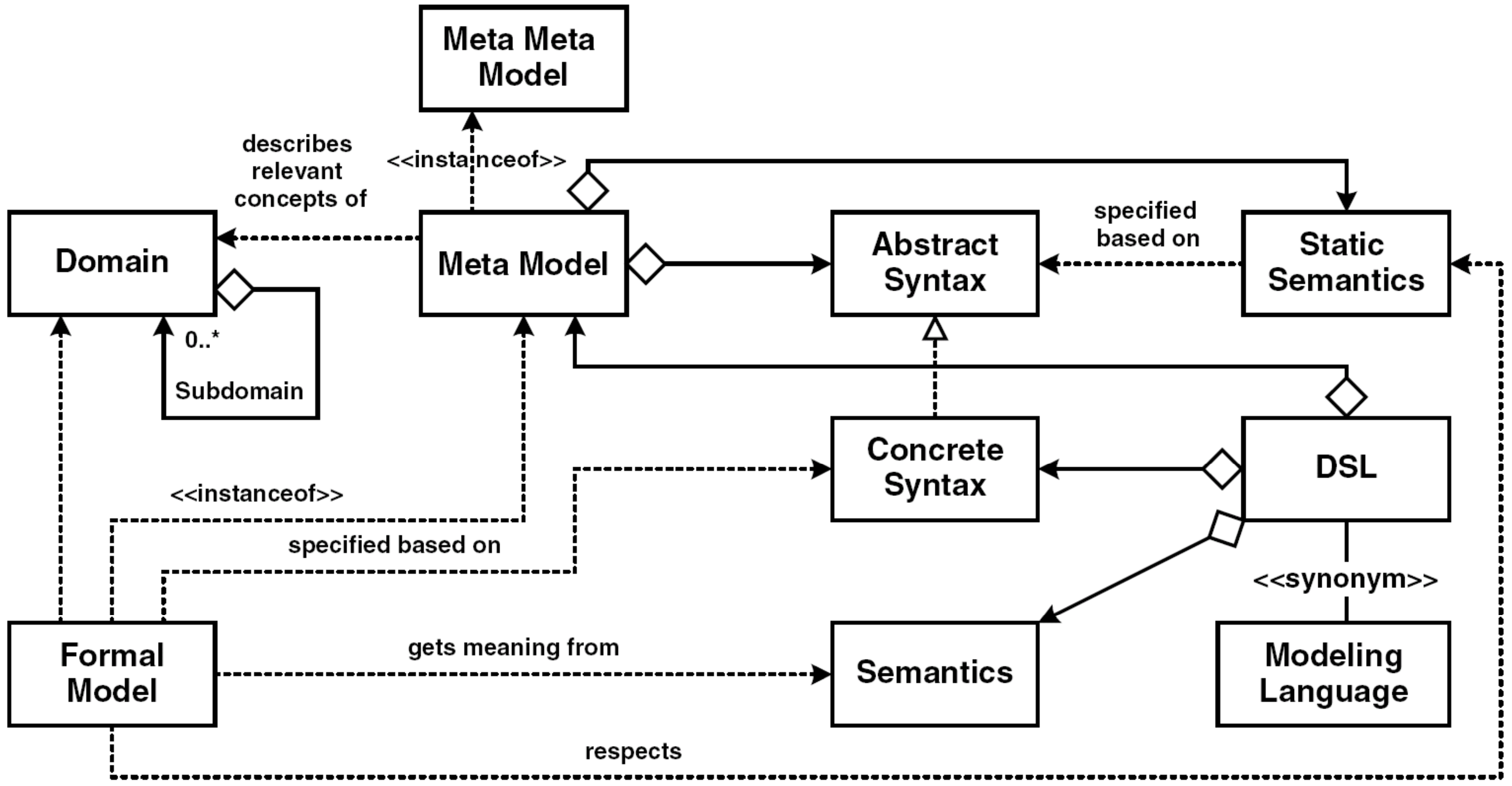
Viewpoints III – connection to modeling

- 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.



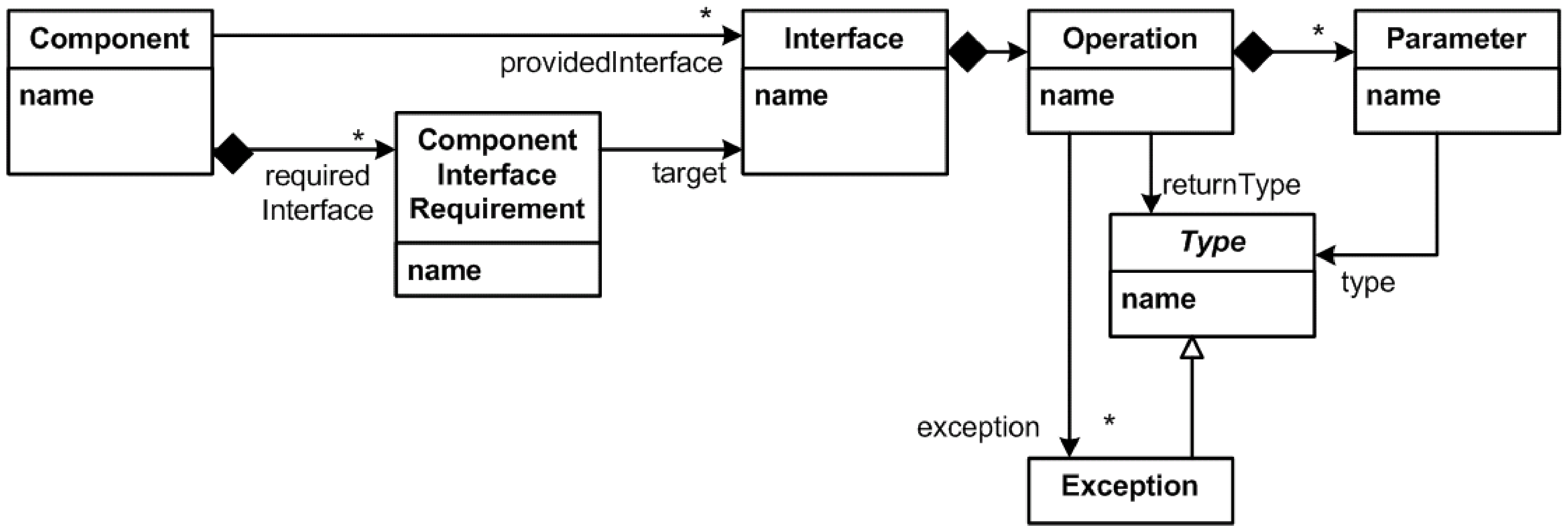
Modeling Languages (DSLs)

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



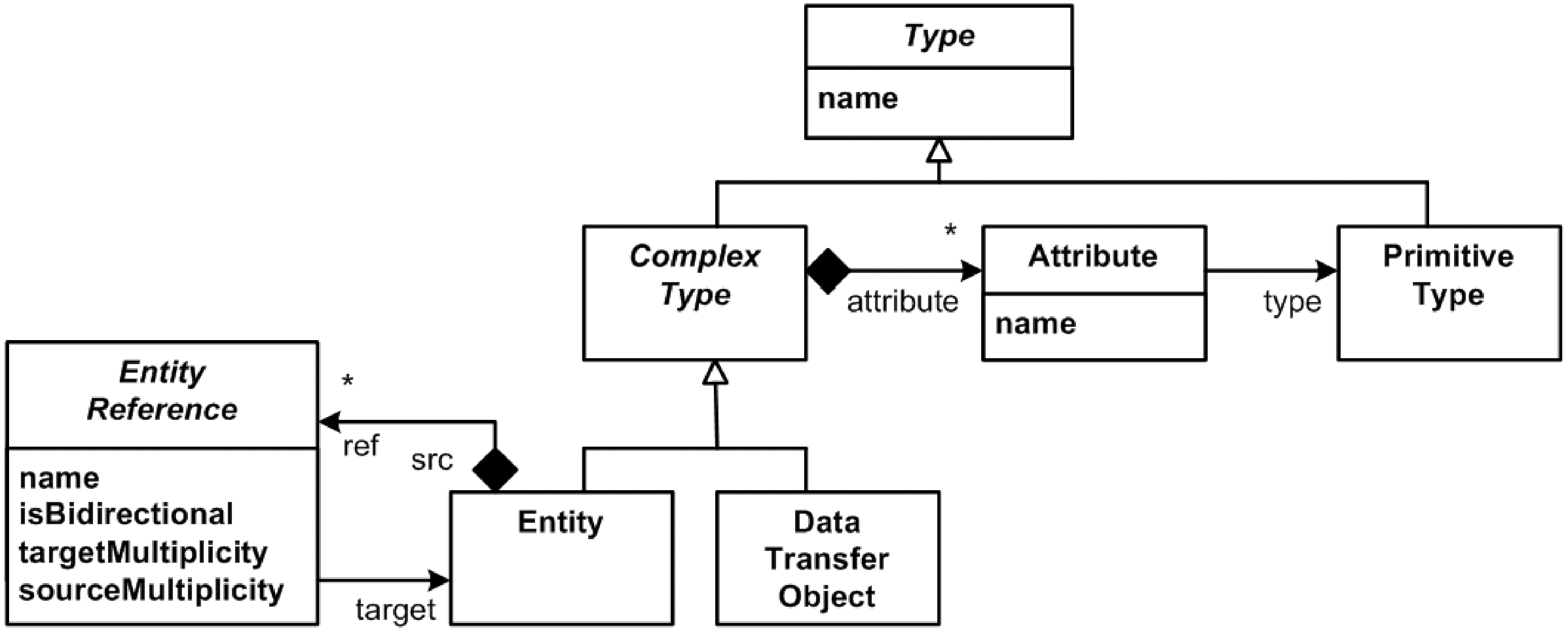
Architectural Metamodels: Type Viewpoint

- **Components** provide **interfaces**
- And components use interfaces (provided by others)
- An interface has a number of **operations** (these are defined as you'd expect)



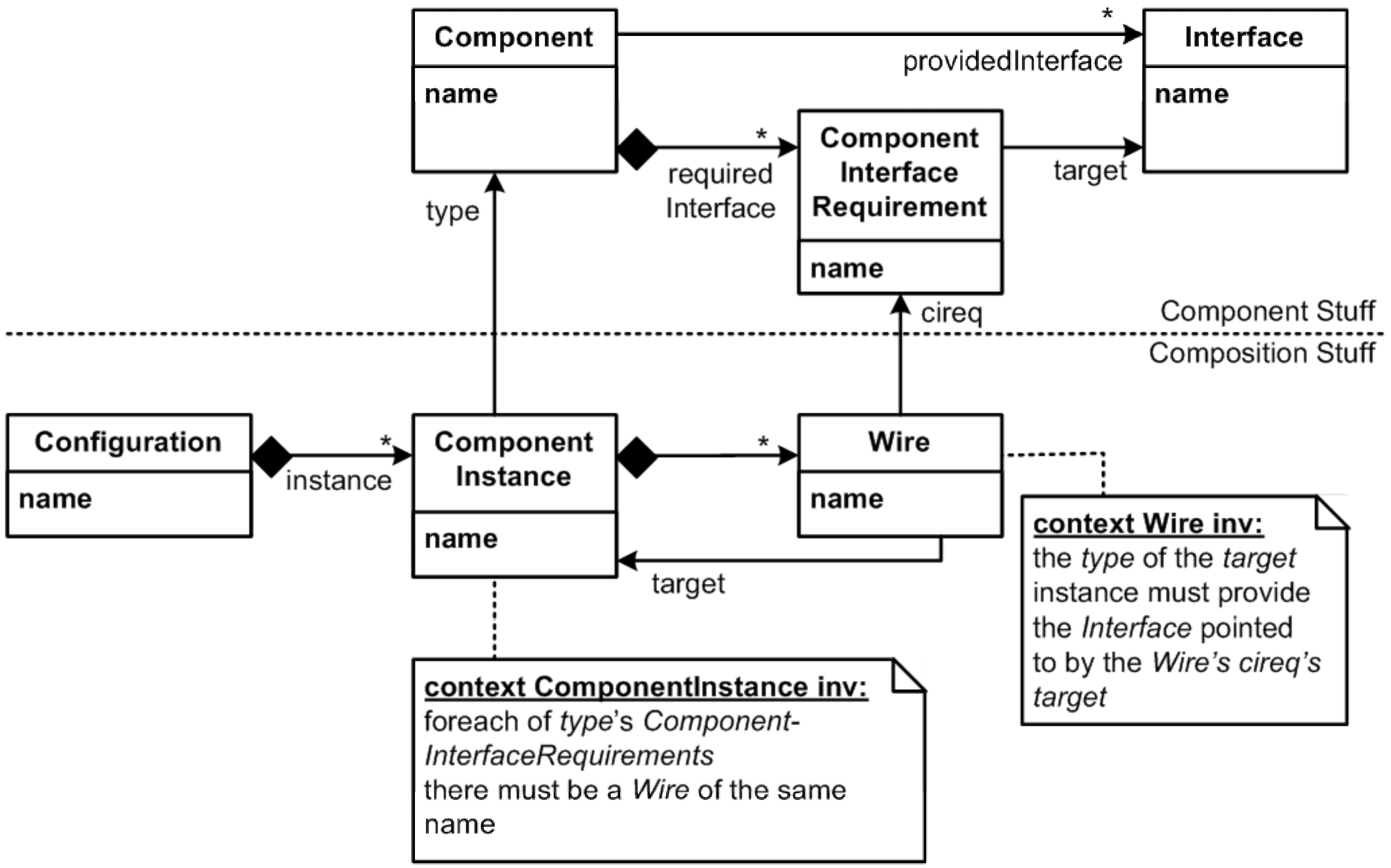
Architectural Metamodels: Type Viewpoint II (Data)

- **Types** are either **complex** or **primitive**
- **Complex Types** have attributes typed to be primitive
- A complex type is either an **Entity** or a **DTO**
- Entities can have **References** to other entities



Architectural Metamodels: Composition Viewpoint

- A **Configuration** consists of a number of **Component Instances** connected by **Wires**

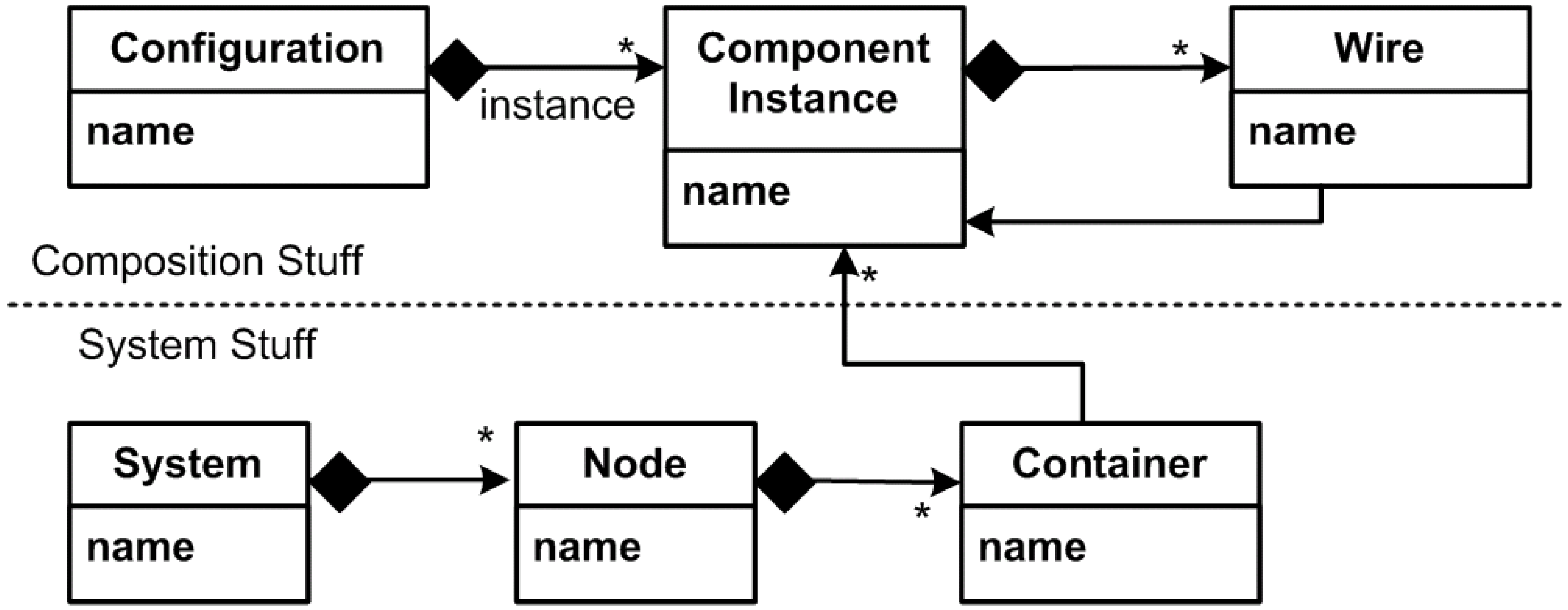


context Wire inv:
the *type* of the *target* instance must provide the *Interface* pointed to by the *Wire's cireq's target*

context ComponentInstance inv:
foreach of *type*'s *Component-InterfaceRequirements* there must be a *Wire* of the same name

Architectural Metamodels: System Viewpoint

- A **System** consists of a number of **Nodes**, each hosting **Containers**
- A Container is a runtime environment for **Component Instances**



Why modeling (as opposed to diagramming)?

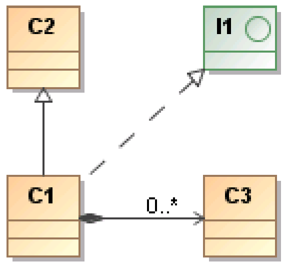
- If I actually formally specify my architecture, I want to **benefit** from that additional “overhead”

- Hence, you want to generate as much of the architecture-related code, for example
  - **Implementation skeletons** to fill in business logic
  - **Build** Files (e.g. ant based)
  - **Adapters** to all kinds of technical infrastructure (remember: the programming model shall be free of such stuff)
  - Infrastructure **configuration files**
  - **Deployment** skripts

- This leads us to **model-driven software development**, which is another topic...

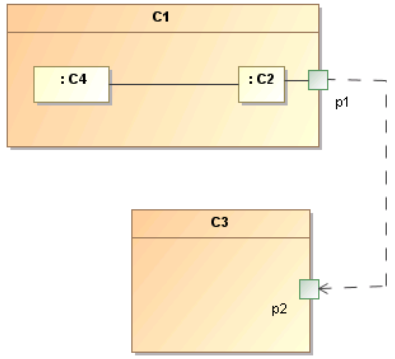
The role of UML

- 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 might need a specific 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.

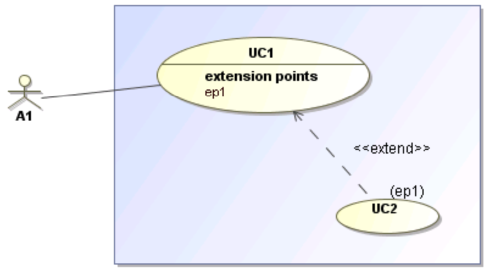


The role of UML II

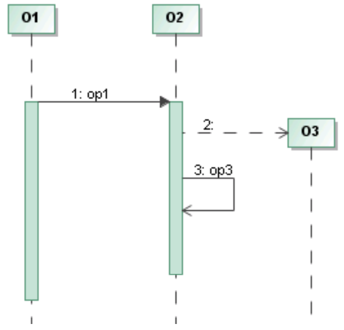
- **Composite Structure Diagrams**
  - Extremely useful for modeling hierarchical structures of components, instances, as well as component connections
  - My favourite kind of diagram in UML ☺



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

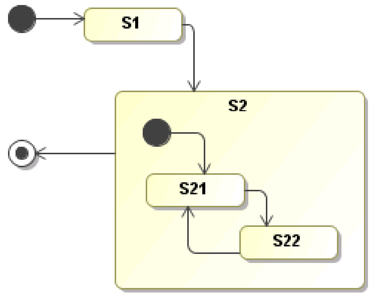


- **Sequence Diagram**
  - Very useful for illustrating the interactions among architectural components
  - Note the sequence diagrams are good for scenarios, not for closed, complete behavioral specification

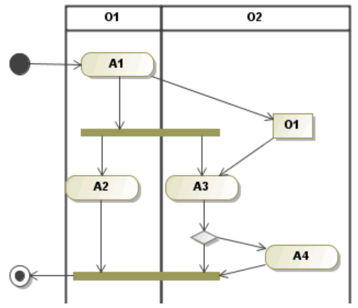


The role of UML III

- **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

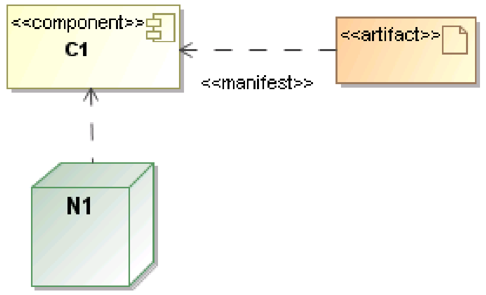


- **Activity Diagrams**
  - Useful for describing activities, their allocation to components and data flow
  - They can be used to formally specify behaviour, but I don't do this very often



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

Architecture Description Languages (ADLs)

- ADLs are **predefined** and **formal** modeling languages specifically **designed to describe architectures** (as opposed to software in general as in UML).
- Typically, an ADL is defined by either a **university**, a **research department** or an **industry consortium** for a specific domain
  - Their practical use is limited
  - http://www.sei.cmu.edu/architecture/adl.html
- ADLs are mostly used in the following domains:
  - Embedded systems
  - Realtime systems
  - Safety critical systems
- Since ADL models are formal, various aspects of a system can be **simulated** or **proven** using them.

Architecture Description Languages (ADLs) II

- Considering the MDSD and DSL stuff we discussed before, an ADL can be seen as a **DSL** for **describing** (certain aspects of) (certain kinds of) **architectures**.

- Since architecture is a wide field, there's no (useful) general purpose ADL – all usable ones are **restricted to a specific technical domain** (embedded realtime systems, automotive systems, ...)

- Often, ADLs describe **components**, **connectors**, **data types, threads** as well as characteristics of the protocols between those artifacts to enable analyses.

- These days many ADLs provide a **UML profile** so it can be integrated with the UML.

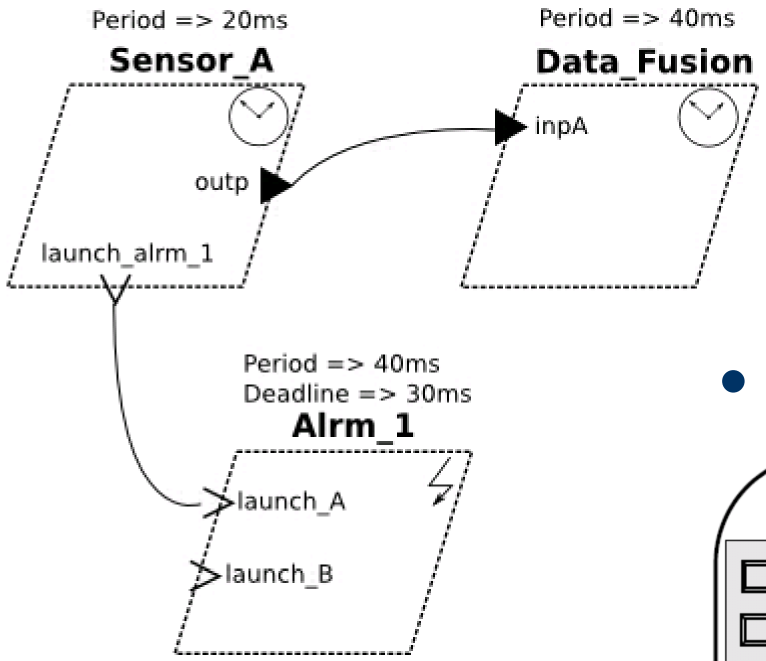
- In most environments they **don't play an important role** (although they maybe should...)

Example ADL: AADL

- AADL stands for *Architecture Analysis & Design Language* (historically: *Avionics Architecture Description Language*)
  - Domain-specific to Embedded Realtime Systems
- It consists of component types and component implementations. The following **component types** exist:
  - Memory
  - Device
  - Processor
  - Bus
  - Data
  - Subprogram
  - Thread
  - thread group
  - Process
  - System
- Components have different **ports**: data ports, event ports
- **Connectors** connect ports from different components
- **Notations:**
  - Textual
  - Graphical
  - UML Profile

Example ADL: AADL II, Examples

- Communicating threads

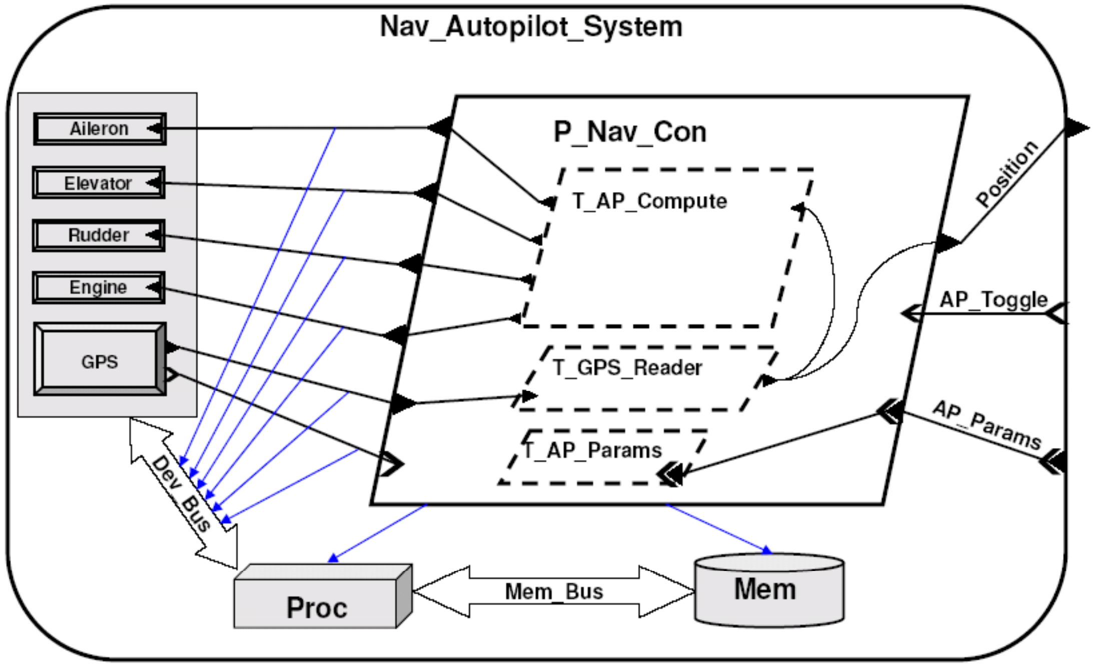


AADL Examples taken from *http://aadl.enst.fr/* with permission from Irfan Hamid. Thanks!

- Data Types

```
data implementation Lat_Long.Generic
subcomponents
        Degs : data Integer;
        Mins : data Integer;
        Secs : data Integer;
end Lat_Long.Generic;
```

- Autopilot System



Do-it-yourself vs. Standard

Comparison Criterion	DIY (DSL)	Standard (UML,ADL)
Tool Support	0	+
Task-Specificness (Modeling Efficiency)	+	-
Adaptability (your architecture changes – what do you do?)	+	0
Suitable for Generation (meta model complexity and comrehensibility)	+	0
Learn-your-domain (defining a meta model helps you understand your own domain)	+	-
Learning overhead (learn the language in order to use it)	-	0
Communicate with outsiders (… who might not want to learn your language)	-	+

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?
  - Specifics for Product Lines & Platforms
- Layout and Typography
- Diagramming Guidelines
- Summary

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 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?**
  - Specifics for Product Lines & Platforms
- Layout and Typography
- Diagramming Guidelines
- Summary

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 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.

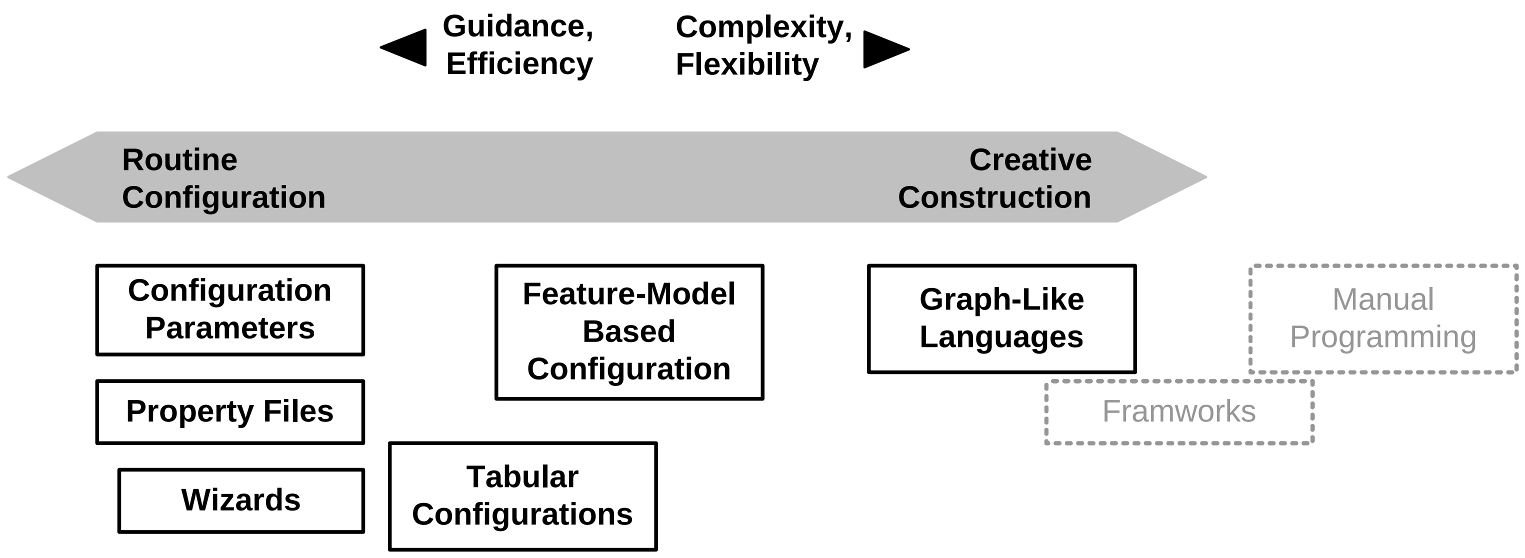
- Do not document things elsewhere that are **obvious** and **understandable** from the code.

CONTENTS

- 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?
  - **Specifics for Product Lines & Platforms**
- Layout and Typography
- Diagramming Guidelines
- Summary

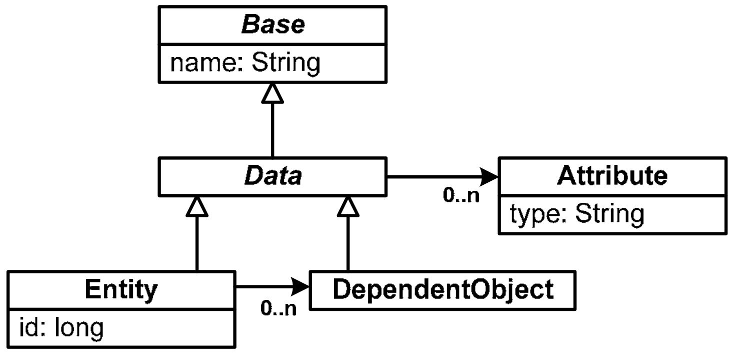
Product Lines & Platforms

- In addition to the best practices already introduced, you must document the **variation points** in the product line.
- A variation point is a location in the product line where **product specifics** can be "plugged in".
- A variation point can support **customization** (build) or **configuration** (selecting):

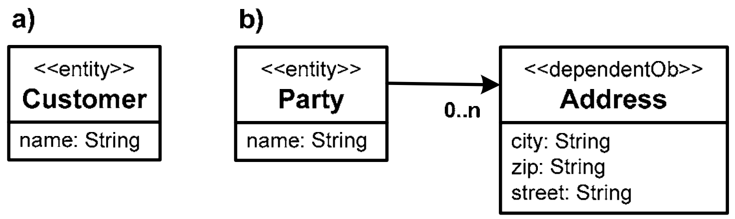


Customization vs. Configuration

- **Customization**
  Example Metamodel



- Based on this sample metamodel, you can build a **wide variety of models:**

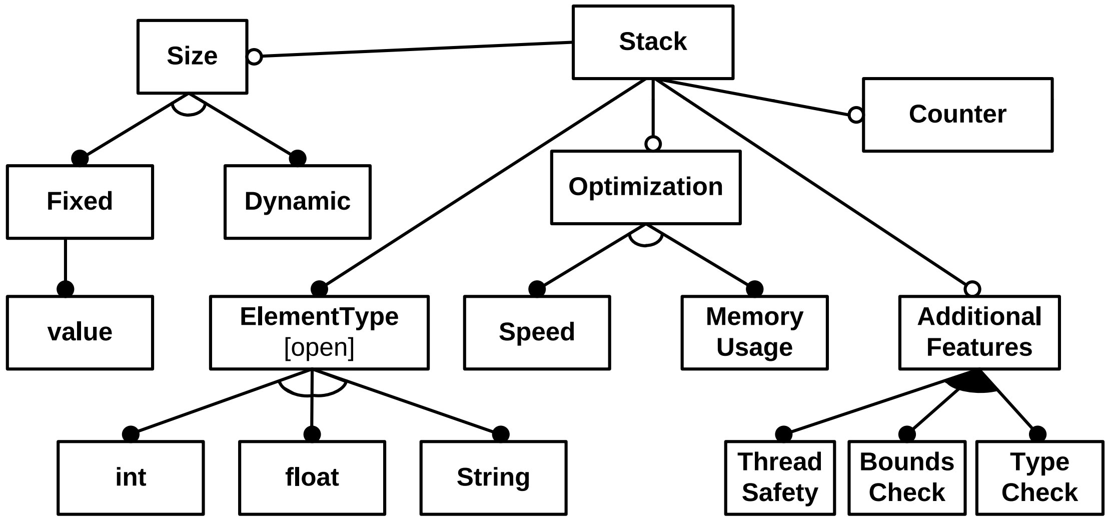


- **Configuration**
  Example Feature Models

  Dynamic Size, ElementType: int, Counter, Threadsafe

  Static Size (20), ElementType: String

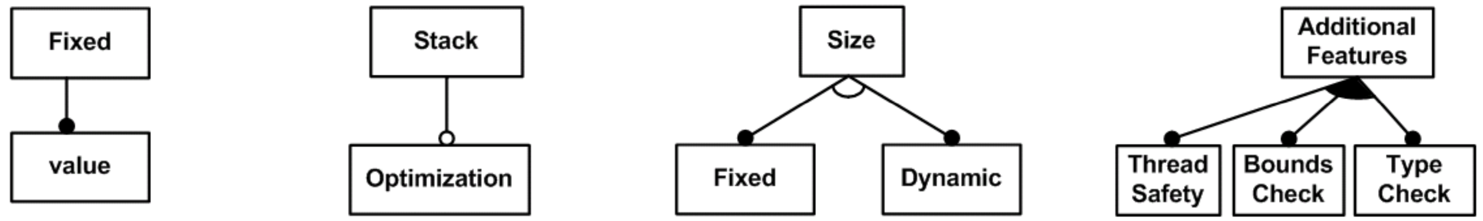
  Dynamic Size, Speed-Optimized, Bounds Check



Documenting Variability using Feature Modeling

- You have to document which variation points exist and how they **relate / constrain** each other
- A **feature model** describes the **variability of a product** line without considering the implementation of the variation point (or feature)
- Subfeatures can have different **relationships,** including

Mandatory Optional Alternative N of M



- A feature can represent some kind of **component** or an **aspect**.

Product Lines & Platforms: What to document

- For each variation point, you need to document
  - Does the variation point support **configuration** or **customization** (frameworks)
  - What is the **mechanism** for selecting/building a variant, incl. the binding time (compile-time, runtime, …)
  - A **rationale for the variation points** – tracing back to the requirements
  - An **example** of customizing/configuring the variation point (basically a kind of mini-tutorial or FAQ)
- **Feature models** (together with explaining text) are a good way of providing an overview over the variability in a product line.

CONTENTS

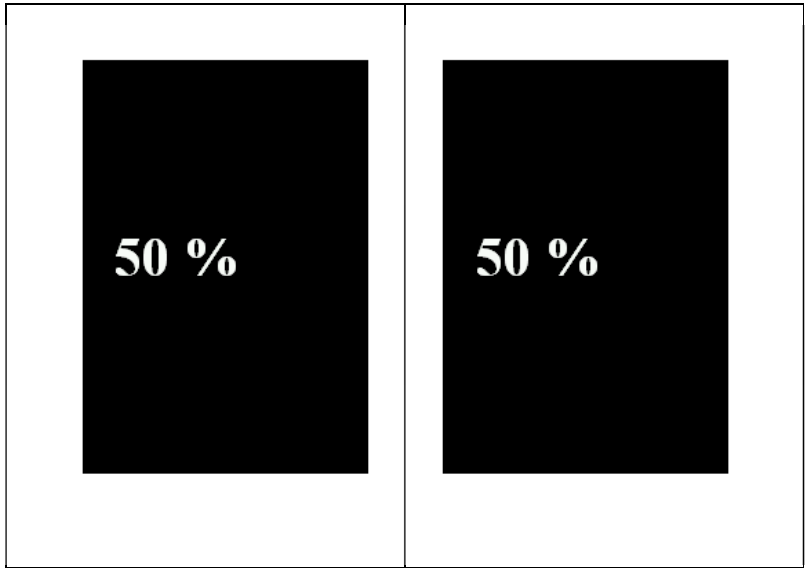
- What is Software Architecture
- Documenting Software Architectures
  - (Structured) Glossaries
  - Patterns and the Pattern Form
  - Pattern Languages
  - Tutorials and FAQs
  - Diagramming and Modeling
  - Channels
  - What about Code?
  - Specifics for Product Lines & Platforms
- **Layout and Typography**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

Page Layout & Typography

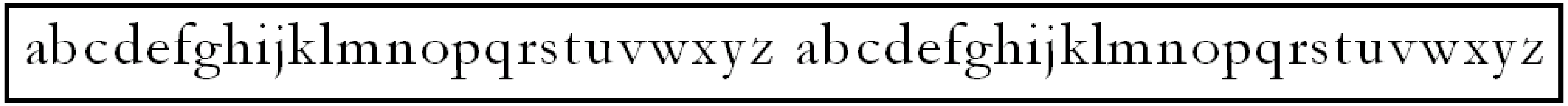
- Typography influences the reader when reading the document
- You'll read faster if the **page geometry** is suitable and you've chosen **suitable fonts**
- You should use **document templates**
  - that contain only stylistic aspects, not 25 sections to fill in
  - They are prepared by a small number 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

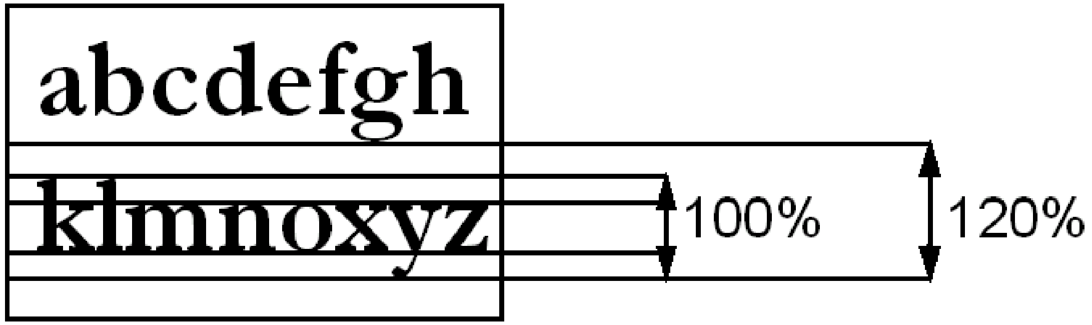


- **2 – 2.5 Alphabets per Line**
  - Long lines are **hard to follow**
  - Short lines require **too many "cariage returns"**
  - Might result in **several columns** in a document

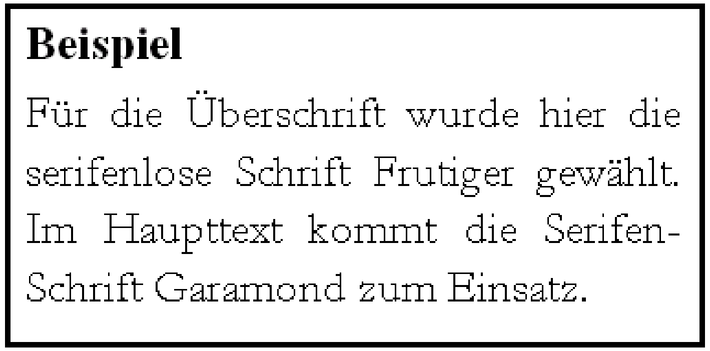


Page Layout & Typography III

- **120% Line Spacing**

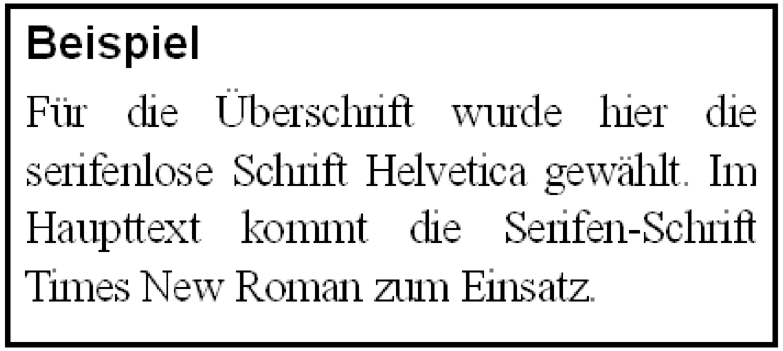


- **2 Fonts**
  - Use Serif Font for the text (guides the eye)
  - Use Sans Serif for Headlines
  - … and maybe Monospaced for Code



Beispiel

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

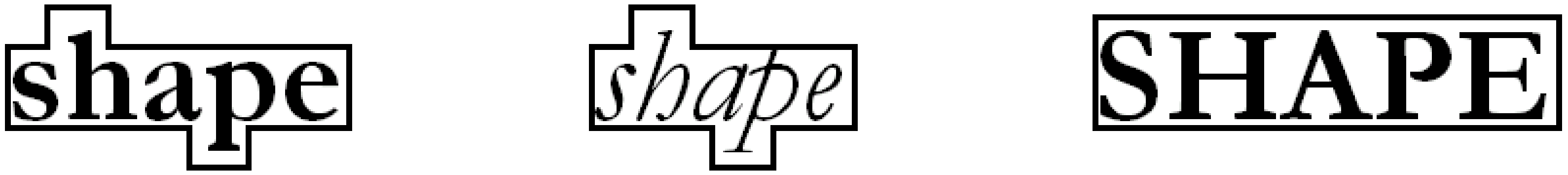


Beispiel

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

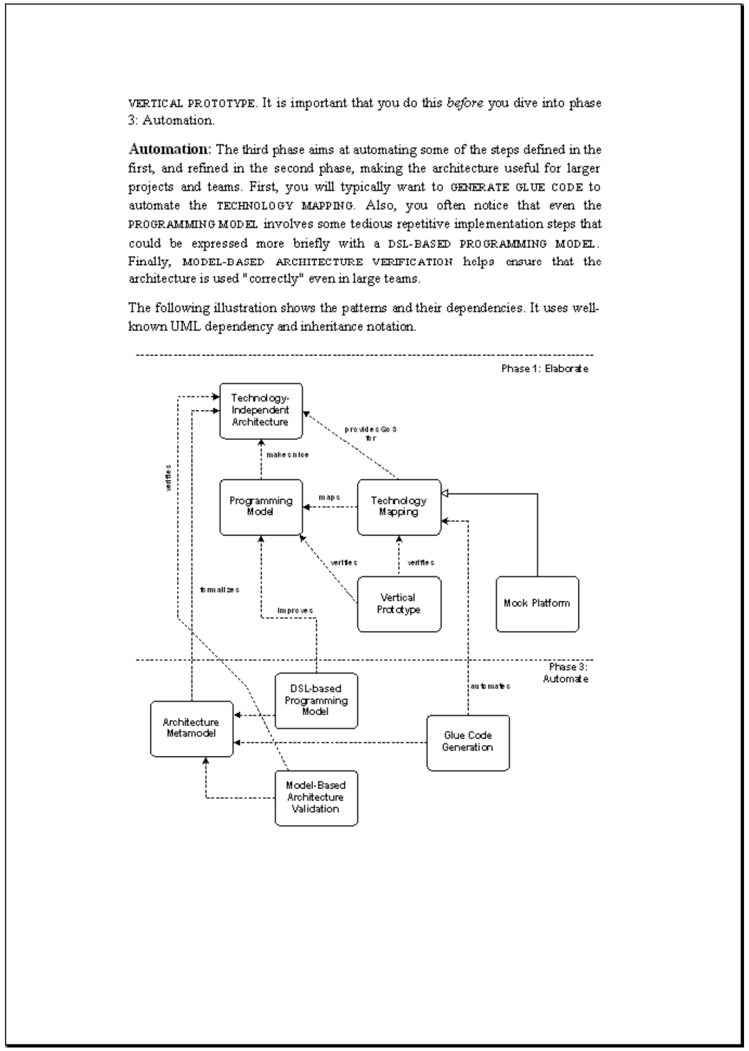
- **Use graphical gimmicks** (lines, symbols), but use them sparsely

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
- **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!
- **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

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

Bad:

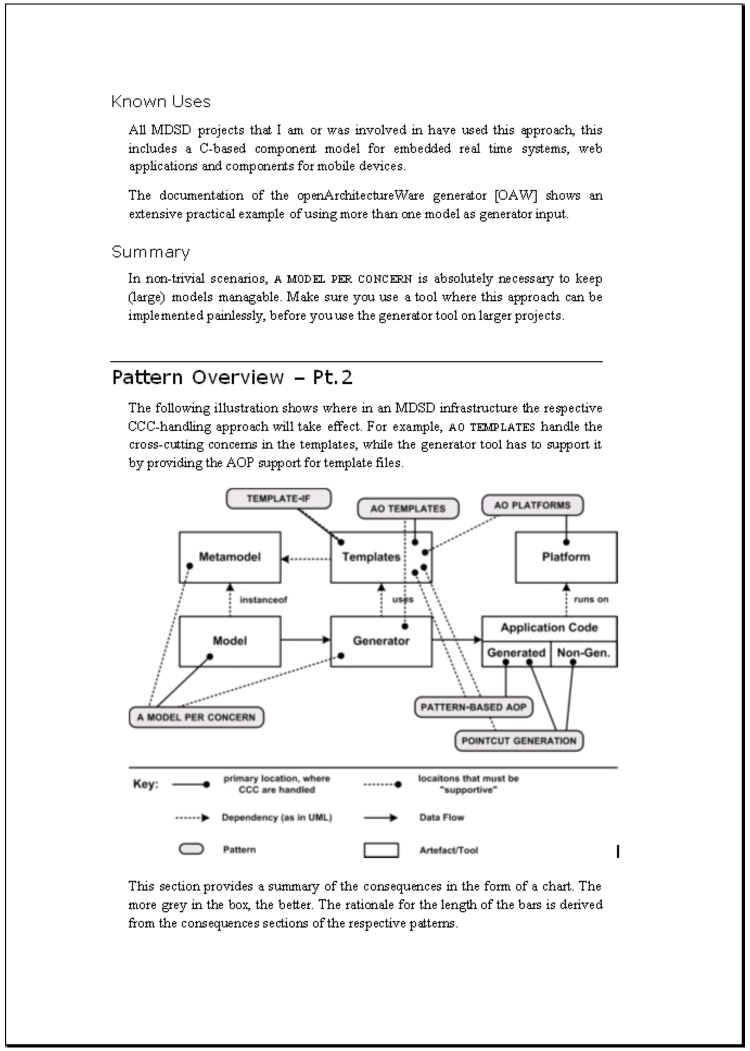


VERTICAL PROTOTYPE. It is important that you do this *before* you dive into phase 3: Automation.

Automation: The third phase aims at automating some of the steps defined in the first, and refined in the second phase, making the architecture useful for larger projects and teams. First, you will typically want to GENERATE GLUE CODE to automate the TECHNOLOGY MAPPING. Also, you often notice that even the PROGRAMMING MODEL involves some tedious repetitive implementation steps that could be expressed more briefly with a DSL-BASED PROGRAMMING MODEL. Finally, MODEL-BASED ARCHITECTURE VERIFICATION helps ensure that the architecture is used "correctly" even in large teams.

The following illustration shows the patterns and their dependencies. It uses well-known UML dependency and inheritance notation.

Good:



Known Uses

All MDSD projects that I am or was involved in have used this approach, this includes a C-based component model for embedded real time systems, web applications and components for mobile devices.

The documentation of the openArchitectureWare generator [OAW] shows an extensive practical example of using more than one model as generator input.

Summary

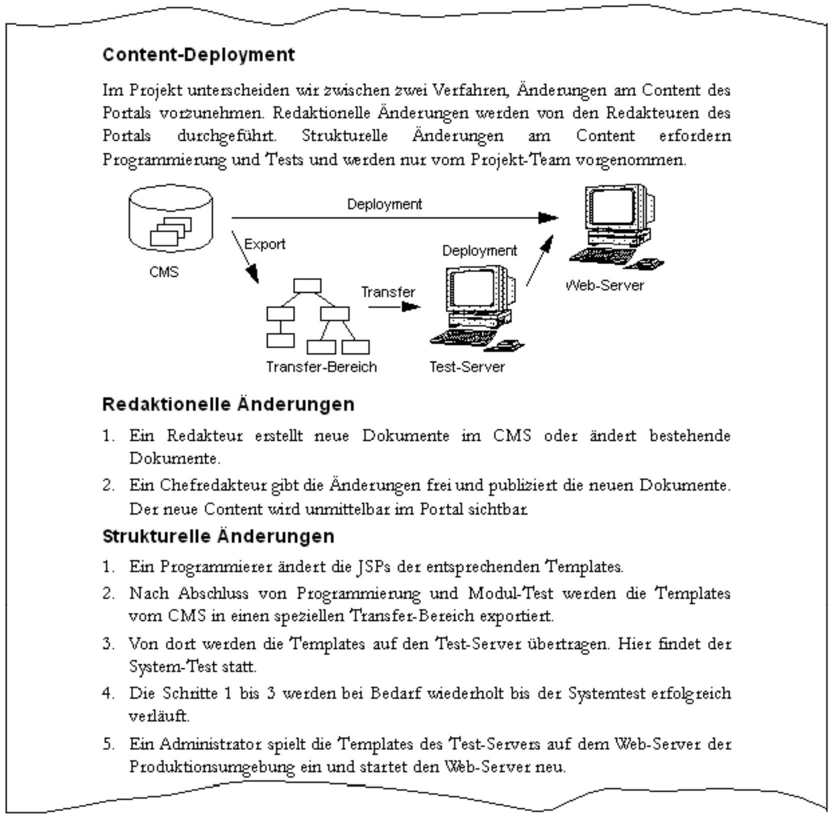
In non-trivial scenarios, A MODEL PER CONCERN is absolutely necessary to keep (large) models managable. Make sure you use a tool where this approach can be implemented painlessly, before you use the generator tool on larger projects.

Pattern Overview – Pt.2

The following illustration shows where in an MDSD infrastructure the respective CCC-handling approach will take effect. For example, AO TEMPLATES handle the cross-cutting concerns in the templates, while the generator tool has to support it by providing the AOP support for template files.

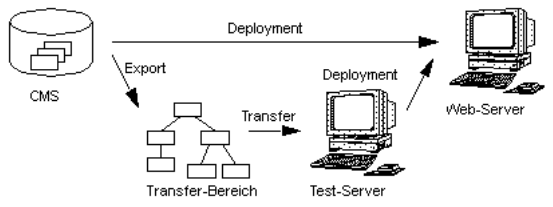
This section provides a summary of the consequences in the form of a chart. The more grey in the box, the better. The rationale for the length of the bars is derived from the consequences sections of the respective patterns.

Examples



Content-Deployment

Im Projekt unterscheiden wir zwischen zwei Verfahren, Änderungen am Content des Portals vorzunehmen. Redaktionelle Änderungen werden von den Redakteuren des Portals durchgeführt. Strukturelle Änderungen am Content erfordern Programmierung und Tests und werden nur vom Projekt-Team vorgenommen.

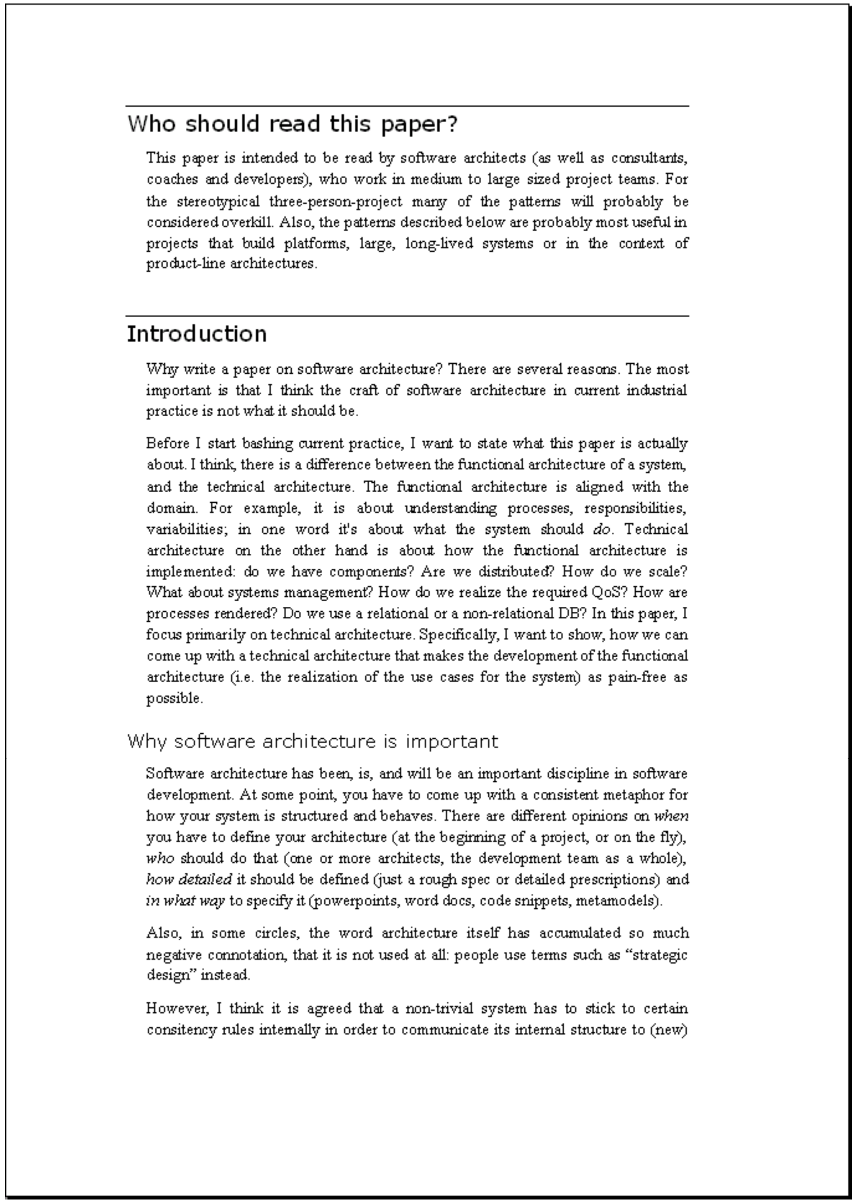


Redaktionelle Änderungen

1. Ein Redakteur erstellt neue Dokumente im CMS oder ändert bestehende Dokumente.
2. Ein Chefredakteur gibt die Änderungen frei und publiziert die neuen Dokumente. Der neue Content wird unmittelbar im Portal sichtbar.

Strukturelle Änderungen

1. Ein Programmierer ändert die JSPs der entsprechenden Templates.
2. Nach Abschluss von Programmierung und Modul-Test werden die Templates vom CMS in einen speziellen Transfer-Bereich exportiert.
3. Von dort werden die Templates auf den Test-Server übertragen. Hier findet der System-Test statt.
4. Die Schritte 1 bis 3 werden bei Bedarf wiederholt bis der Systemtest erfolgreich verläuft.
5. Ein Administrator spielt die Templates des Test-Servers auf dem Web-Server der Produktionsumgebung ein und startet den Web-Server neu.



Who should read this paper?

This paper is intended to be read by software architects (as well as consultants, coaches and developers), who work in medium to large sized project teams. For the stereotypical three-person-project many of the patterns will probably be considered overkill. Also, the patterns described below are probably most useful in projects that build platforms, large, long-lived systems or in the context of product-line architectures.

Introduction

Why write a paper on software architecture? There are several reasons. The most important is that I think the craft of software architecture in current industrial practice is not what it should be.

Before I start bashing current practice, I want to state what this paper is actually about. I think, there is a difference between the functional architecture of a system, and the technical architecture. The functional architecture is aligned with the domain. For example, it is about understanding processes, responsibilities, variabilities; in one word it's about what the system should *do*. Technical architecture on the other hand is about how the functional architecture is implemented: do we have components? Are we distributed? How do we scale? What about systems management? How do we realize the required QoS? How are processes rendered? Do we use a relational or a non-relational DB? In this paper, I focus primarily on technical architecture. Specifically, I want to show, how we can come up with a technical architecture that makes the development of the functional architecture (i.e. the realization of the use cases for the system) as pain-free as possible.

Why software architecture is important

Software architecture has been, is, and will be an important discipline in software development. At some point, you have to come up with a consistent metaphor for how your system is structured and behaves. There are different opinions on *when* you have to define your architecture (at the beginning of a project, or on the fly), *who* should do that (one or more architects, the development team as a whole), *how detailed* it should be defined (just a rough spec or detailed prescriptions) and *in what way* to specify it (powerpoints, word docs, code snippets, metamodels).

Also, in some circles, the word architecture itself has accumulated so much negative connotation, that it is not used at all: people use terms such as "strategic design" instead.

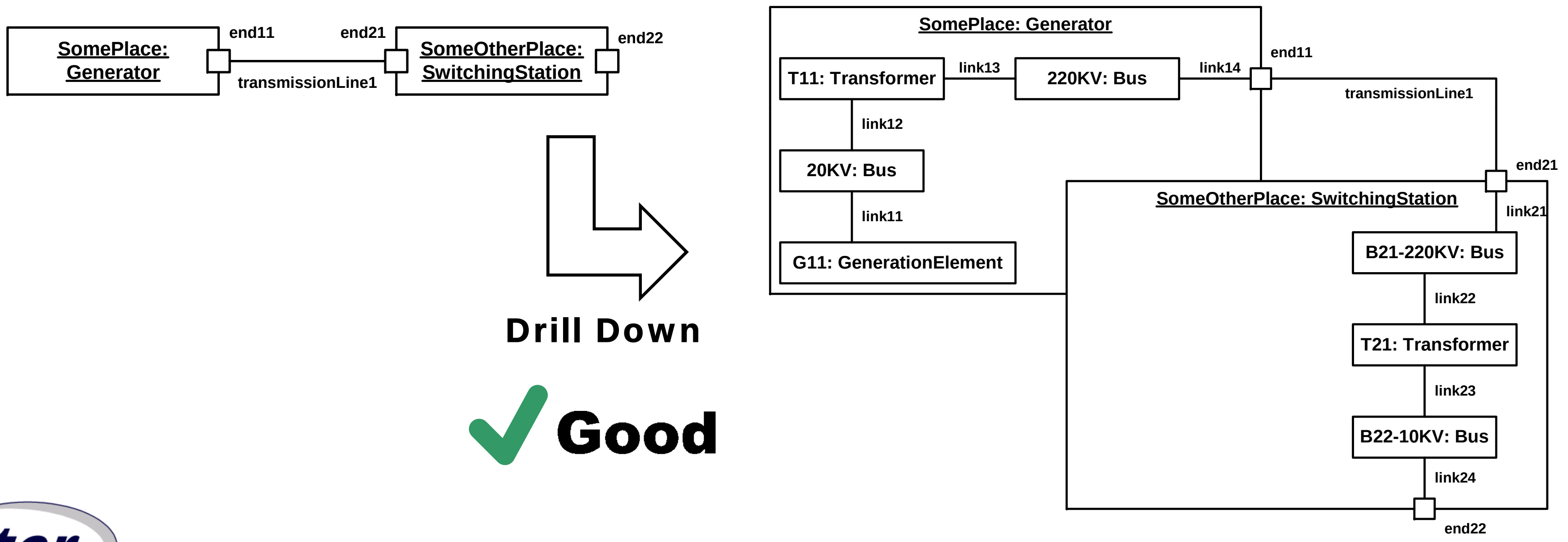
However, I think it is agreed that a non-trivial system has to stick to certain consitency rules internally in order to communicate its internal structure to (new)

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?
  - Specifics for Product Lines & Platforms
- Layout and Typography
- **Diagramming Guidelines**
- Summary

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

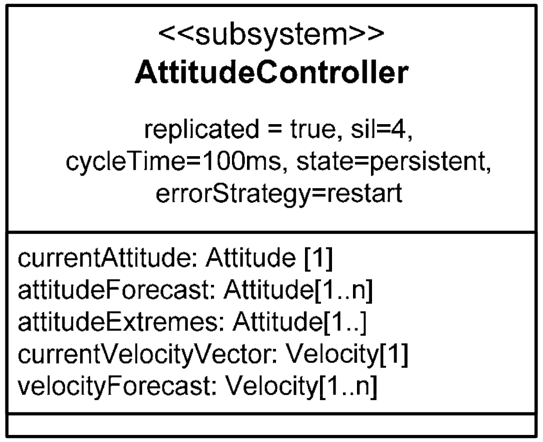
- **Always explain diagrams,** the picture itself is not enough
  - Give it a half-sentence **title**
  - **Explain** in prose **what the diagram shows** (or use the diagram to illustrate conceptes explained in the running text)
  - In the explanation **don’t explain every detail** shown in the diagram, 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**

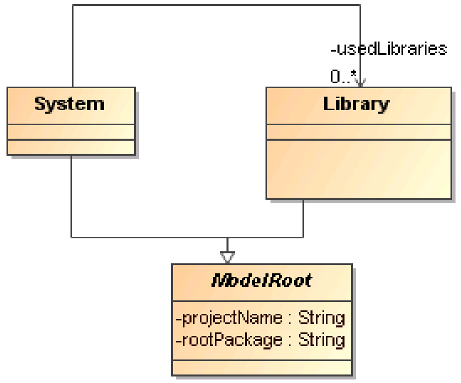
Diagramming Guidelines IV

- **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

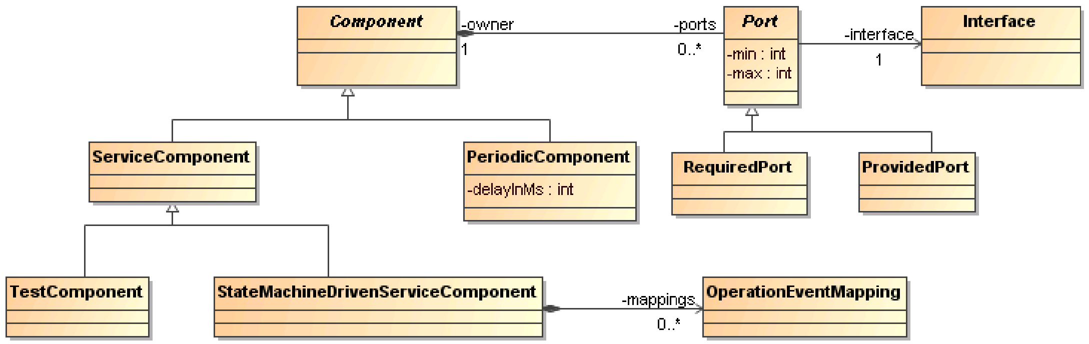


<<subsystem>>
AttitudeController
replicated = true, sil=4, cycleTime=100ms, state=persistent, errorStrategy=restart
currentAttitude: Attitude [1]
attitudeForecast: Attitude[1..n]
attitudeExtremes: Attitude[1..]
currentVelocityVector: Velocity[1]
velocityForecast: Velocity[1..n]

X **Bad**



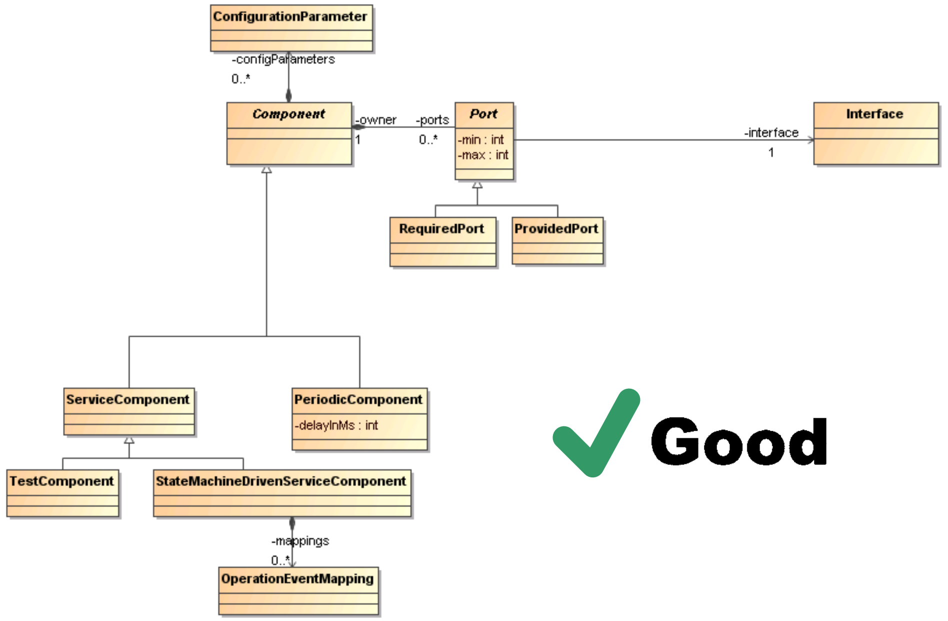
X Bad



✓ **Good**

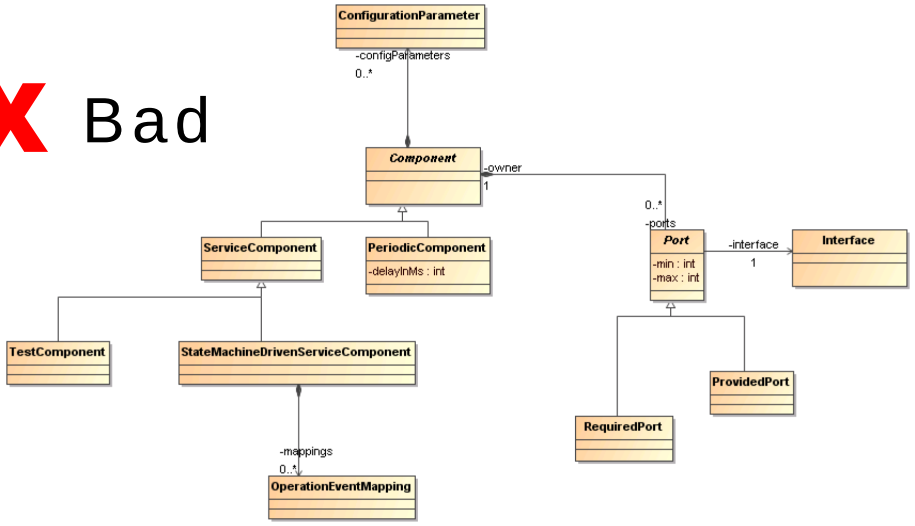
Diagramming Guidelines V

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



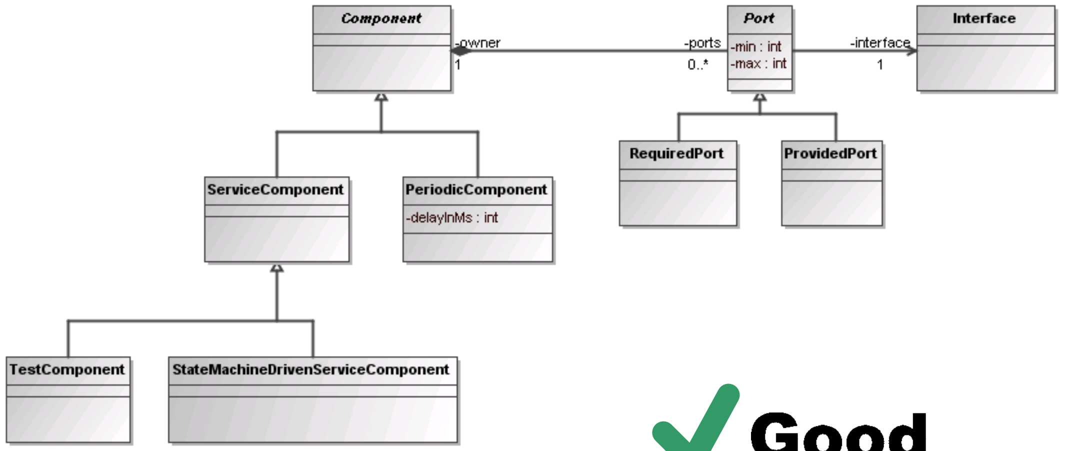
✓ **Good**

✗ Bad

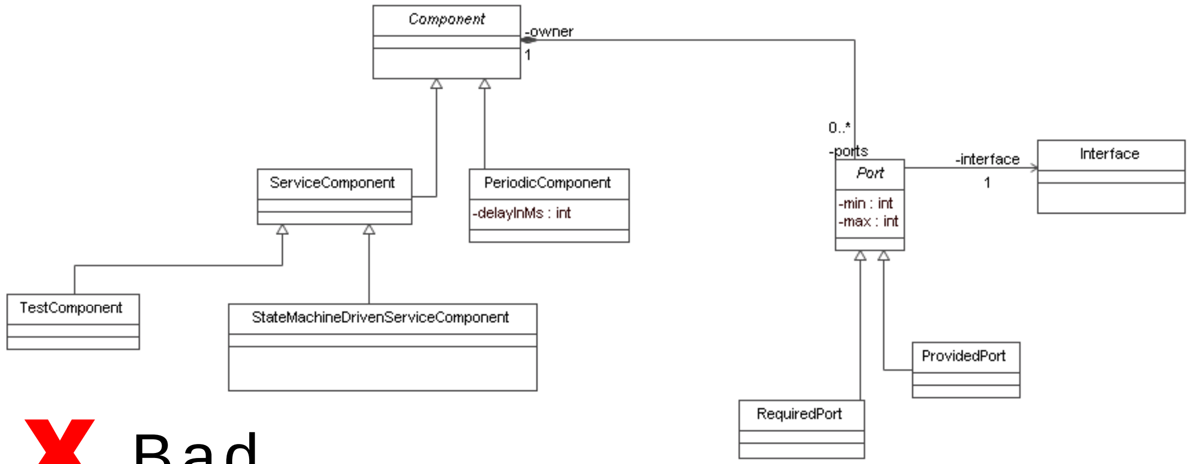


Diagramming Guidelines VI

- **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!



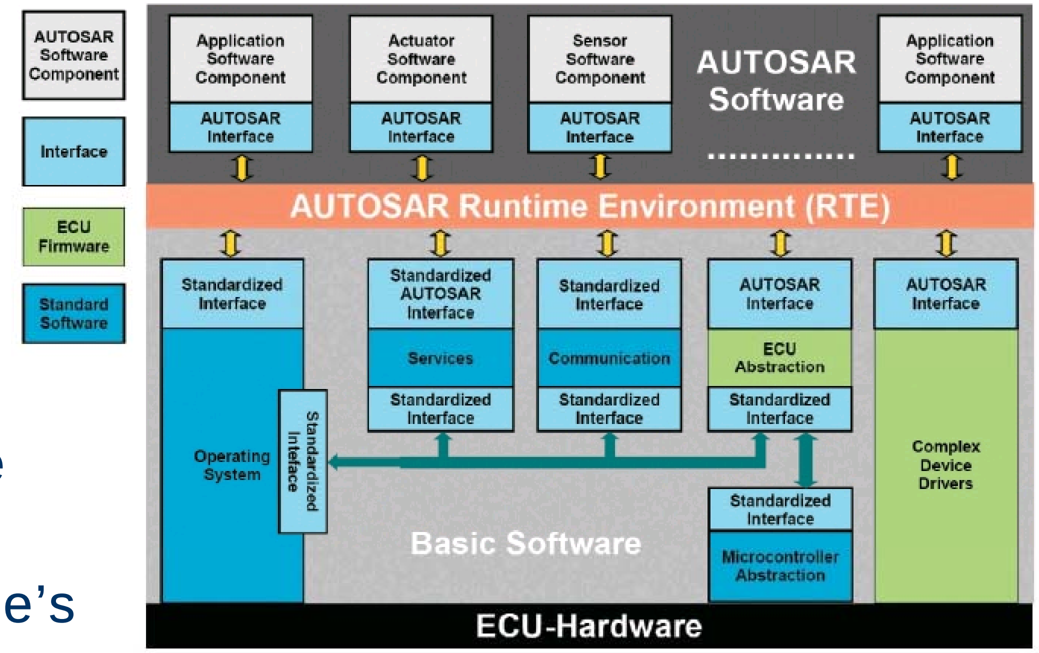
✓ **Good**



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** **?** ✗ 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

- 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?
  - Specifics for Product Lines & Platforms
- Layout and Typography
- Diagramming Guidelines
- **Summary**

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!