Architecture As Language

Markus Voelter

www.voelter.de
voelter@acm.org

About

Markus Voelter
Independent Consultant

About

Markus Voelter
Independent Consultant

Software Architecture
DSLs & MDSD
Product Lines

About

Markus Voelter
Independent Consultant

http://www.voelter.de
voelter@acm.org
skype: schogglad

Architecture

Core System Abstractions
Stuff that needs to be consistent

Architecture

Core System Abstractions
Stuff that needs to be consistent

DSLs/ MDSD

Define „custom" languages
Automate construction of artifacts

Architecture

Core System Abstractions
Stuff that needs to be consistent

PLE

Variant Management
Feature Models
Aspect Orientation

DSLs/ MDSD

Define „custom" languages
Automate construction of artifacts

Architecture

Core System Abstractions
Stuff that needs to be consistent

DSLs/ MDSD

Define „custom" languages
Automate construction of artifacts

PLE

Variant Management
Feature Models
Aspect Orientation

This Talk

Based on actual practical experience

Currently in use with four of my customers

Benchmarked by suitability for use in today's projects

Part 1
A real-word Story

1.1 System Background
1.2 The Starting Point
1.3 What is a language?
1.4 Developing the Language

...

Part 2
Theory and Concepts

2.1 What we did in a Nutshell
2.2 Domain Specific Languages
2.3 The Importance of Viewpoints
2.4 Benefits
2.5 Why Textual?
2.6 Model Validation
2.7 Generating Code
2.8 Architecture Analysis Tools
2.9 Standards, UML and ADLs ...

Part 2
Theory and Concepts

2.10 Why not use a 3GL?
2.11 My Notion of Components
2.12 Component Implementation
2.13 The Role of Patterns
2.14 Documentation
2.15 Expressing Variability
2.16 PLE – Big Picture
2.17 Process
2.18 Underlying Concepts

More Reading:

This presentation is not suitable for reading without me talking in parallel. You can find documents describing the content at

http://www.voelter.de/data/articles/ArchitectureAsLanguage-PDF.pdf
http://www.voelter.de/data/articles/ArchitectureAsLanguage-Part2-PDF.pdf

1

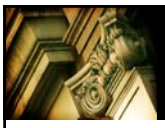
A Real-World Story

Customer

Consultant

Requirements
Domain Expertise

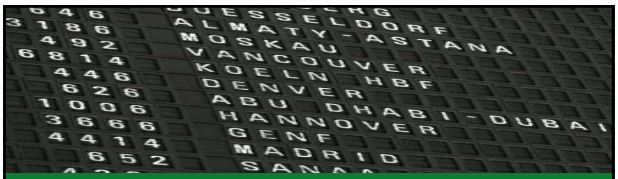
Solution Approach



1.1

A Real-World Story

System Background



Flight Management System

Monitors
Website
Aircraft-Module

Data Center

Distributed System

A lot of experience with systems like these

Evolve over 15 – 20 years

Multiple Languages

Java
C++
C#

Technology Abstraction

Cannot update everything at once

Loose coupling Versioning

Different Installations have different features

Variants Product Lines

Backbone: Messaging

[Good]

Technology Evaluated

Business Object Model

[Not so good]

Problem:

Consistence?
Big Picture?
Technology Abstraction?

Huge & Typical Problem

Language!

Formal Language!

1.3

A Real-World Story

What is a language?

INFORMAL

Set of well-defined terms

INFORMAL

Stakeholders agree on meaning

FORMAL

Metamodel

FORMAL

Metamodel Grammar

FORMAL

Metamodel Grammar Notation

1.4

A Real-World Story

Developing the Language

Let‘s define a formal language to describe the system‘s architecture

Components

Component:

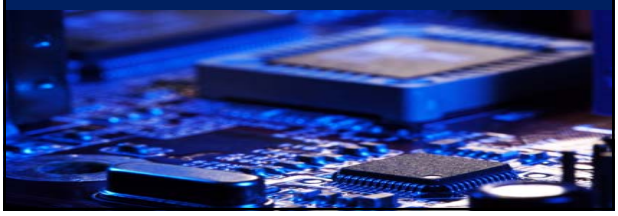
Smallest Architecturally Relevant Building Block

Component:

Piece of functionality

Component:

Can be instantiated



Initial Components:

Delay Calculator
Info Screen
Aircraft Module

```
component DelayCalculator {}
component InfoScreen {}
component AircraftModule {}
```

Language
==
Conceptual Architecture

„Program“
==
Application Architecture

Conceptual:
Component

Application:
DelayCalculator, InfoScreen, Aircraft Module

DelayCalculator receives data from Aircraft Module

DelayCalculator forwards data to InfoScreens

Interfaces

```
component DelayCalculator
        implements IDelayCalculator {}
component InfoScreen
        implements IInfoScreen {}
component AircraftModule
        implements IAircraftModule {}

interface IDelayCalculator {}
interface IInfoScreen {}
interface IAircraftModule {}
```

But we don‘t express interface *requirements!*

Interface *provision* + Interface *requirement*

➔ Dependency graph

Essential for understanding and tool-based Analysis

```
component DelayCalculator {
  provides IDelayCalculator
  requires IInfoScreen
}
component InfoScreen {
  provides IInfoScreen
}
component AircraftModule {
  provides IAircraftModule
  requires IDelayCalculator
}

interface IDelayCalculator {}
interface IInfoScreen {}
interface IAircraftModule {}
```

There are **many** InfoScreens and AircraftModules

Components need to be instantiated

Instantiation:
very important concept when defining languages

```
component InfoScreen {
  provides IInfoScreen
}


instance screen1: InfoScreen
instance screen2: InfoScreen
…
```

Componets (types) are already **compatible** via their **interfaces.**

One DelayCalculator typically talks to **many** InfoScreens

Ports

Port:
Communication Endpoint

Port:
Owned by Component

Port:
Instantiated along with Component

Port:
Can have cardinality

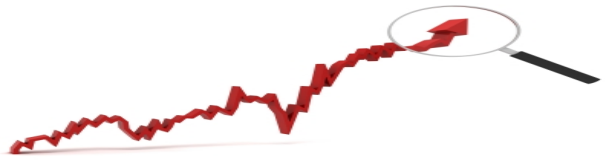
```
component DelayCalculator {
  provides default: IDelayCalculator
  requires screens[0..n]: IInfoScreen
}

component InfoScreen {
  provides default: IInfoScreen
}

component AircraftModule {
  provides default: IAircraftModule
  requires calculator[1]: IDelayCalculator
}
```

We should separate ports and interfaces by **roles**!

```
component DelayCalculator {
  provides aircraft: IAircraftStatus
  provides managementConsole:
                IManagementConsole
  requires screens[0..n]: IInfoScreen
}

component Manager {
  requires backend[1]: IManagementConsole
}

component InfoScreen {
  provides default: IInfoScreen
}

component AircraftModule {
  requires calculator[1]: IAircraftStatus
}
```

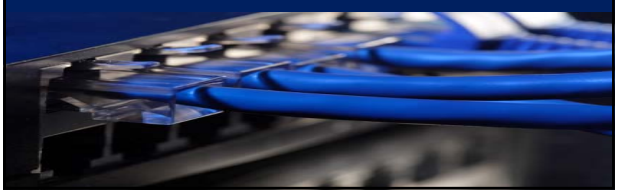
Enrichment of Language

➔ **Better Application Architecture**



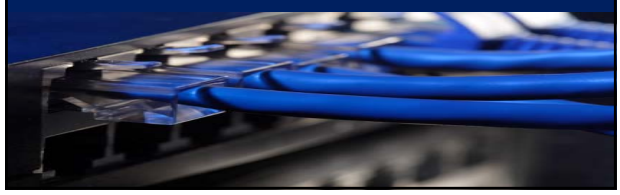
How to actually **connect** instances and their ports?

Connectors



Connector:

Connects two or more port instances



```
component DelayCalculator {
  requires screens[0..n]: IInfoScreen
  …
}

component InfoScreen {
  provides default: IInfoScreen
}

instance dc: DelayCalculator
instance screen1: InfoScreen
instance screen2: InfoScreen

connect dc.screens
    to (screen1.default, screen2.default)
```

How do we **structure/organize** the overall system?

Namespaces

```
namespace com.mycompany {
  namespace datacenter {
    component DelayCalculator {
      provides aircraft: IAircraftStatus
      provides managementConsole: IManagementConsole
      requires screens[0..n]: IInfoScreen
    }
    component Manager {
      requires backend[1]: IManagementConsole
    }
  }
  namespace mobile {
    component InfoScreen {
      provides default: IInfoScreen
    }
    component AircraftModule {
      requires calculator[1]: IAircraftStatus
    }
  }
}
```

Systems are a separate viewpoint

```
namespace com.mycompany.test {
  system testSystem {
    instance dc: DelayCalculator
    instance screen1: InfoScreen
    instance screen2: InfoScreen
    connect dc.screens
        to (screen1.default, screen2.default)
  }
}
```

We don‘t know the InfoScreens statically when the system is designed!

Dynamic Connectors

Dynamic Connector:

Discovers target(s) at runtime using a naming/trader/registry service

```
namespace com.mycompany.production {
  instance dc: DelayCalculator

  // InfoScreen instances are created and
  // started in other configurations
  dynamic connect dc.screens every 60 query {
    type = IInfoScreen
    status = active
  }
}
```

Dynamic Wiring does not invalidate the overall approach

Similar Approach can be used for redundant connectors

Registration

Properties for instances used in lookup

```
namespace com.mycompany.datacenter {
  registered instance dc1: DelayCalculator {
      parameters {role = primary}
  }
  registered instance dc2: DelayCalculator {
      parameters {role = backup}
  }
}
```

Interfaces aren‘t really defined yet

Interface:

set of related messages

```
interface IInfoScreen {
  message expectedAircraftArrivalUpdate
          (id: ID, time: Time)
  message flightCancelled(flightID: ID)
  …
}
```

Data Structures

```
typedef long ID
struct Time {
  hour: int
  min: int
  seconds: int
}
```

Different kinds of messages: oneway, req/resp

Message Interaction Patterns

```
interface IAircraftStatus {

  oneway message reportPosition
                (aircraft: ID, pos: Position )

  request-reply message reportProblem {
    request (aircraft: ID, problem: Problem,
             comment: String)
    reply (repairProcedure: ID)
  }

}
```

Is it really Messages?

?

Status Updates!
Flight Plans…!

Wrong Abstraction!

Messages are the implementation, not the architectural intent.

Data Replication

Data Replication
1) Define Data
2) Updated by few, read by many
3) Full update, incremental update, invaldation…

New architectural abstraction is added to the language

```
struct FlightInfo {
  from: Airport
  to: Airport
  scheduled: Time
  expected: Time
  …
}

replicated singleton flights {
  flights: FlightInfo[]
}

component DelayCalculator {
  publishes flights
}

component InfoScreen {
  consumes flights
}
```

More Concise

System **derives** Messages for (full & partial) update, invalidation, etc.

More specifics

```
component DelayCalculator {
  publishes flights { publication = onchange }
}

component InfoScreen {
  consumes flights { init = all update = every(60) }
}
```

We still need messages. **Semantics?**

Pre- and Post-Conditions

```
interface IAircraftStatus {

  oneway message reportPosition(aircraft: ID,
                                pos: Position ) {
    pre aircraft != null: "aircraft not specified"
    pre pos != null: "position not specified"
  }

  request-reply message reportProblem {
    request (aircraft: ID, problem: Problem,
                           comment: String)
    reply (repairProcedure: ID)
    pre aircraft != null: "aircraft not specified"
    pre problem != null: "problem not specified"
    post repairProcedure != null
  }

}
```

Common Constraints: Shorter Syntax

```
interface IAircraftStatus {

  oneway message reportPosition(aircraft: ID!,
                pos: Position! )

  request-reply message reportProblem {
    request (aircraft: ID!, problem: Problem!,
             comment: String!)
    reply (repairProcedure: !ID)
  }

}
```

Message Sequences: Protocol State Machines

```
interface IAircraftStatus {
  oneway message registerAircraft(aircraft: ID! )
  oneway message unregisterAircraft(aircraft: ID! )
  oneway message reportPosition(aircraft: ID!,
                pos: Position! )
  request-reply message reportProblem {
    request (aircraft: ID!, problem: Problem!,
             comment: String!)
    reply (repairProcedure: !ID)
  }
  protocol initial = new {
    state new {
      registerAircraft => registered
    }
    state registered {
      unregisterAircraft => new
      reportPosition
      reportProblem
    }
  }
}
```

Invariants for Data Structures

```
struct Time {
  hour: int
  min: int
  seconds: int
  inv hour >= 0 && hour <= 23:
      "hour must be 0-23"
  inv min >= 0 && min <= 59:
      "min must be 0-59"
  inv seconds >= 0 && seconds <= 59:
      "seconds must be 0-59"
}
```

Widely distributed! Versioning?

Mark up versions of components (and other stuff)

```
component DelayCalculator {
  publishes flights { publication = onchange }
}

newImplOf component DelayCalculator: DelayCalculatorV2
```

```
component DelayCalculator {
  publishes flights { publication = onchange }
}

newVersionOf component DelayCalculator:
              DelayCalculatorV3 {
  publishes flights { publication = onchange }
  provides somethingElse: ISomething
}
```

None of those things is new or unique.

But everything is expressed explicitly and consistently, all in one place.

Theory and Concepts

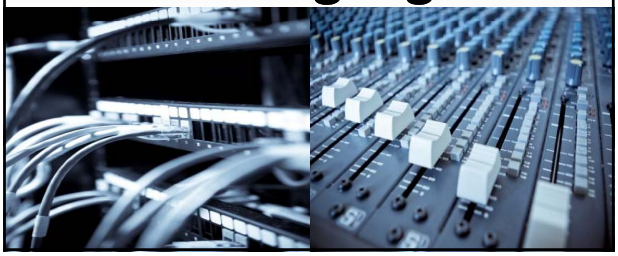
Theory and Concepts
What we did in a Nutshell

As you understand and develop your Architecture…

Develop a language to express it!

Language resembles architectural concepts

We express the application(s) with the language.



Architecture

DSL

2.2

Theory and Concepts

Domain Specific Languages

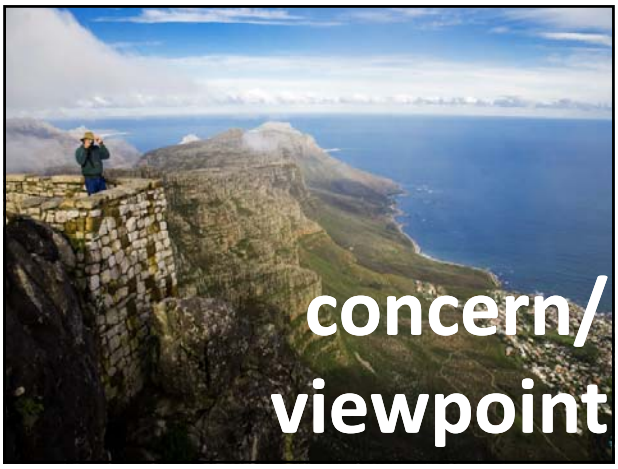
Definition

A DSL is a **focussed**, **processable language** for describing a specific **concern** when building a system in a specific **domain**. The **abstractions** and **notations** used are natural/suitable for the **stakeholders** who specify that particular concern.

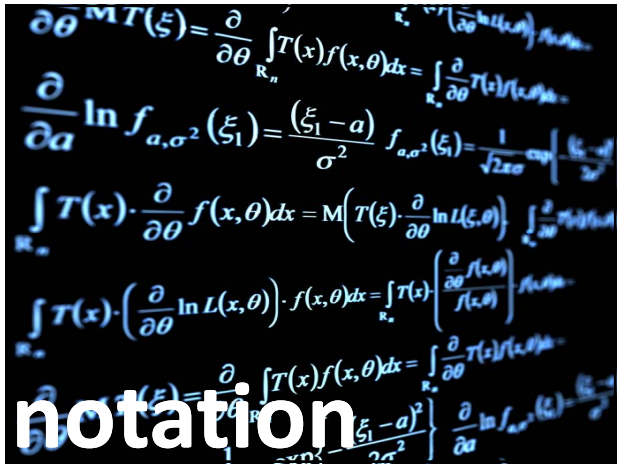
focussed

processable

language



abstractions



stakeholders

Dimensions

internal

vs.

external

compiled

vs.

interpreted

customization

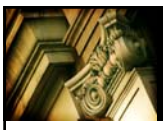
vs.

configuration

graphical

vs.

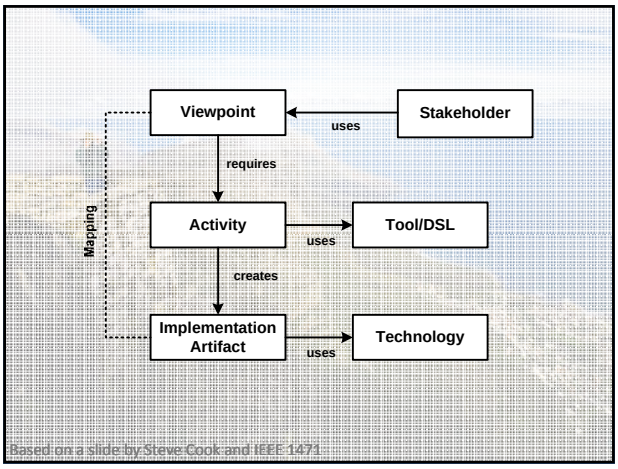
textual

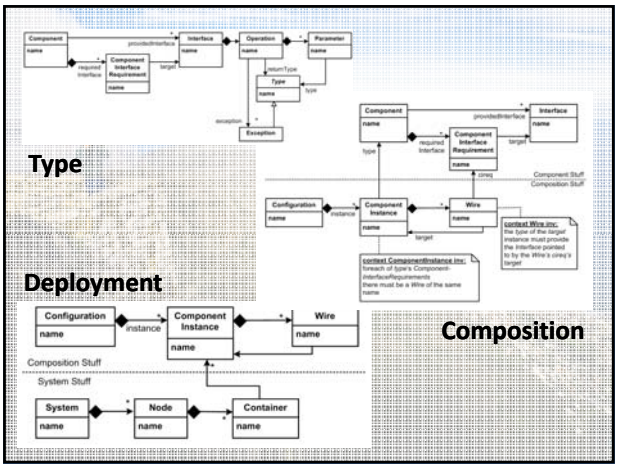


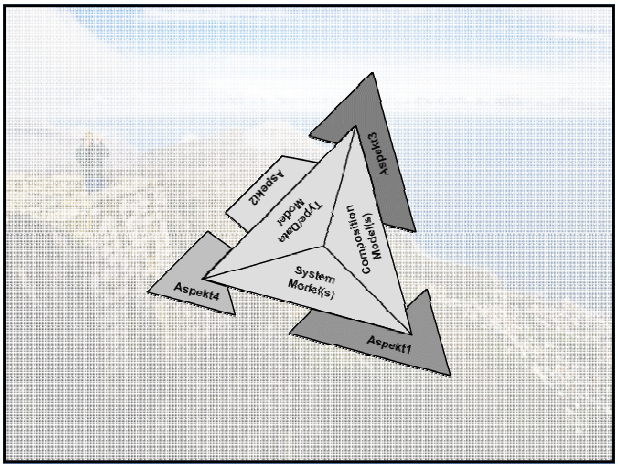
2.3

Theory and Concepts

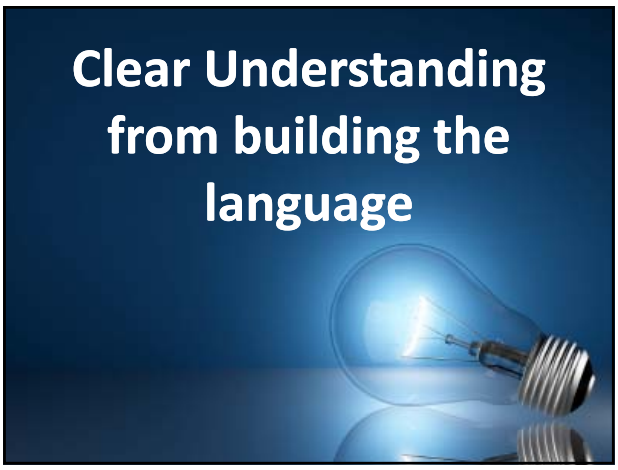
The Importance of Viewpoints


Viewpoint
uses
Stakeholder
requires
Activity
uses
Tool/DSL
creates
Mapping
Implementation Artifact
uses
Technology
Based on a slide by Steve Cook and IEEE 1471


Type
Deployment
Composition
Configuration
Component Instance
Wire
Composition Stuff
System Stuff
System
Node
Container


Aspekt3
Aspekt2
Typ/Dat. Modell
Composition Model(s)
System Model(s)
Aspekt4
Aspekt1

2.4
Theory and Concepts
Benefits


Clear Understanding from building the language

Unambigious Vocabulary

Concepts independent from Technology

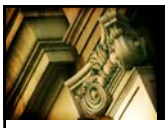
Programming Model can be defined based on Conceptual Arcitecture

Architecture „executable“
(i.e. more than rules and docs)

2.4

Theory and Concepts

Why Textual?



2.4

Theory and Concepts

… or: why not graphical?

Languages and Editors are easier to build

Languages and Editors are easier to build

Evolve Language and simple editor as you understand and discuss the architecture, in real time!

Integrates easily with current infrastructure: CVS/SVN diff/merge

adapting existing models as the DSL evolves

Model evolution is trivial, you can always use *grep*.

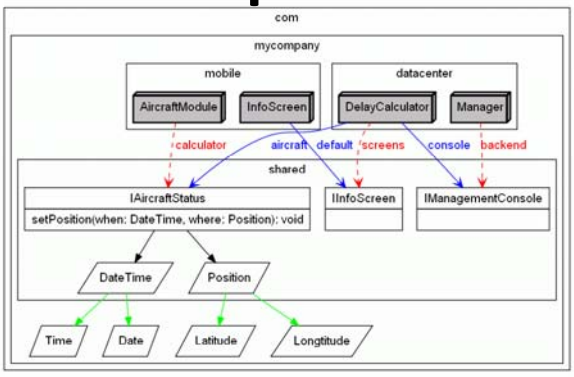
Many Developers prefer textual notations

When a graphical notation is better, you can **visualize**.

Via M2M
Read-Only
Auto-Layout
Drill-Down

Textual DSLs
vs.
Graphical
vs.
Visualization

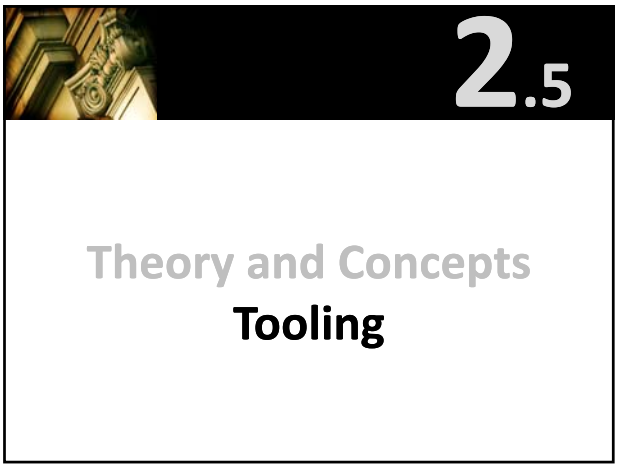
Graphviz



Prefuse

I am aware of the usefulness of graphical notations.
I am exaggerating here a little bit.

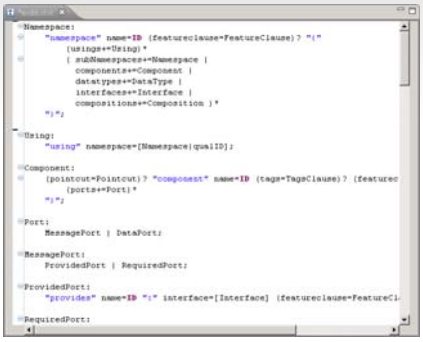
But: We've been building software with text only for a long time. We know it works.



Several tools available. Example: oAW Xtext

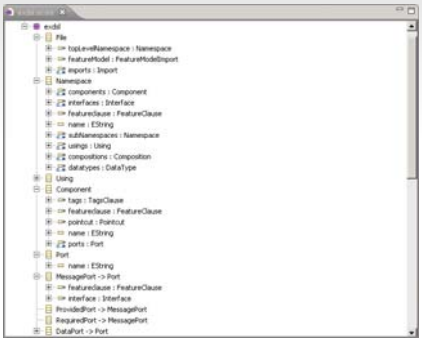
eclipse modeling PROJECT

Specify Grammar

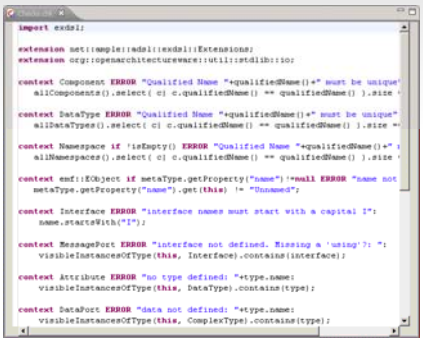


Antlr Grammar and Parser is generated from this specification

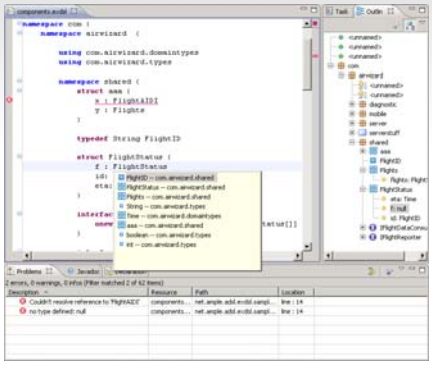
Generated Metamodel



Specify Constraints

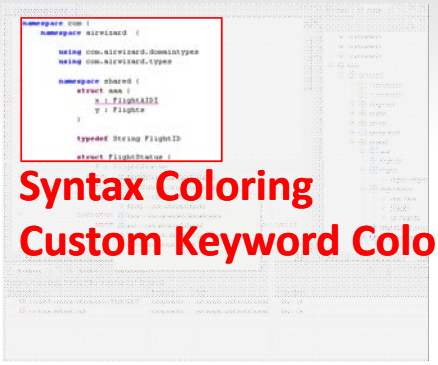


Generated Editor

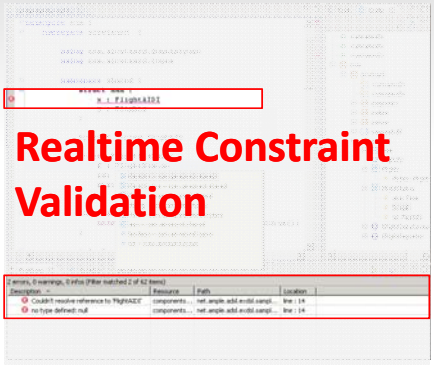


Generated Editor

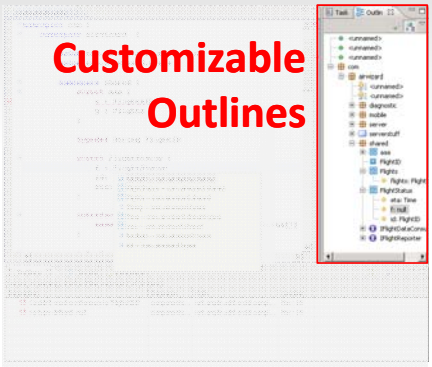
Generated Editor



Generated Editor

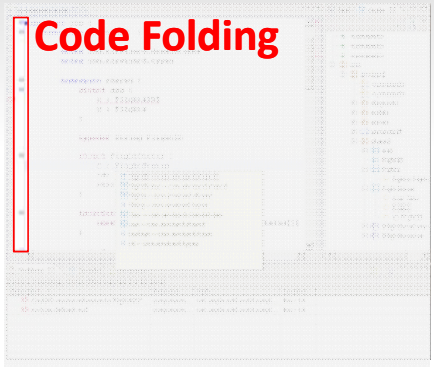


Generated Editor



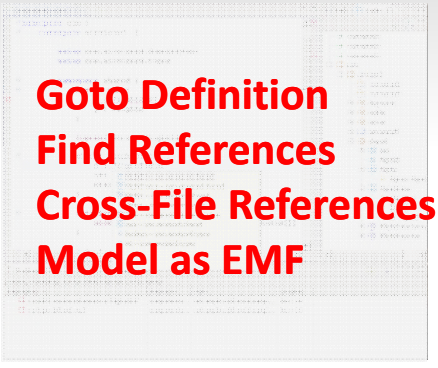
Generated Editor

Code Folding

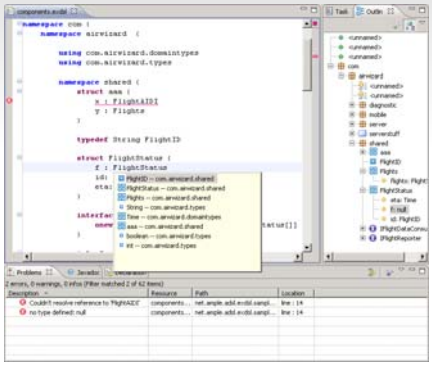


Generated Editor

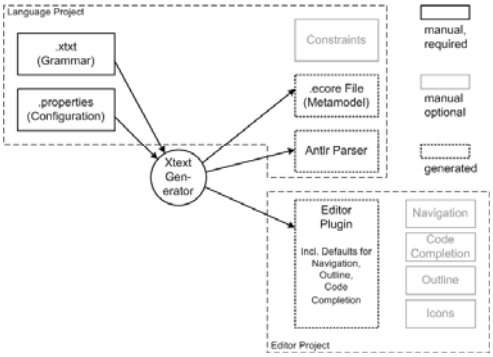
Goto Definition
Find References
Cross-File References
Model as EMF



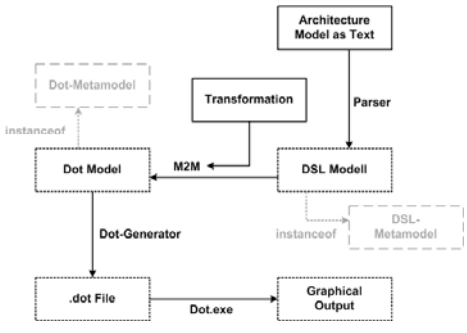
Generated Editor



Xtext Overview



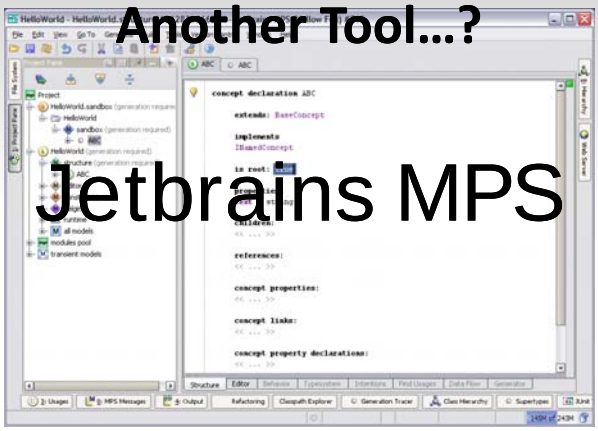
GrapViz Overview



GrapViz Trafo Code

Another Tool…?

Another Tool…?



Another Tool…?

INTENTIONAL™
SOFTWARE

2.6

Theory and Concepts

Model Validation

Grammar is not expressive enough

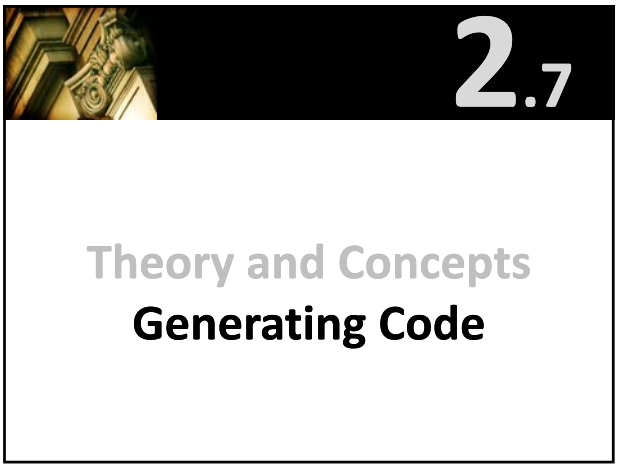
Simple Examples:
Name-Uniqueness
Type Checks
Non-Nullness

More Complex Examples:
Version Compatibility
Overall completeness
Quality of Service

```
context Interface ERROR "interface names must start "+
                        "with a capital I":
  name.startsWith("I");


context Component ERROR "Qualified Name "+
                        qualifiedName()+" must be unique" :
   allComponents().
     select( c | c.qualifiedName() == qualifiedName() )
     .size == 1;


context Attribute ERROR "no type defined: "+
                        type.name:
  visibleInstancesOfType(this, DataType)
    .contains(type);


context Connector ERROR "target must be provided port":
  ProvidedPort.isInstance( u);
```

Precondition I:
Algorithm for Constraint

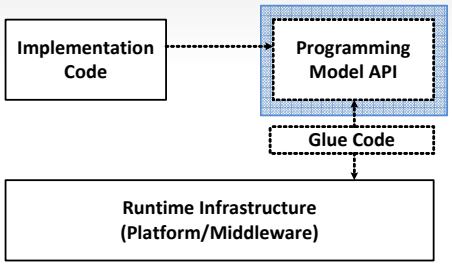
Precondition II:
All Data Available



Since we already have a formal model….

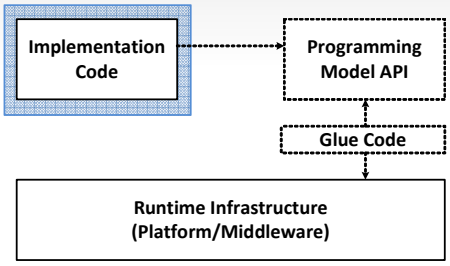
Generate API

Maps Architectural Concepts to Implementation language (non-trivial!)



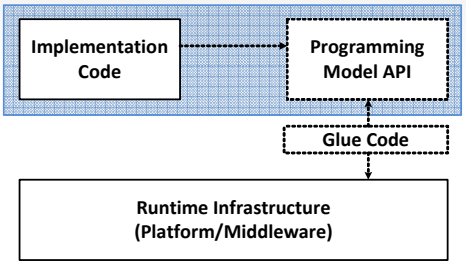
Implementation

Implementation only depends on the generated programming model API



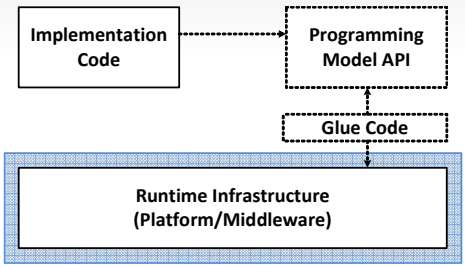
Programming Model

Generated API + Usage Idioms Completely Technology-Independent



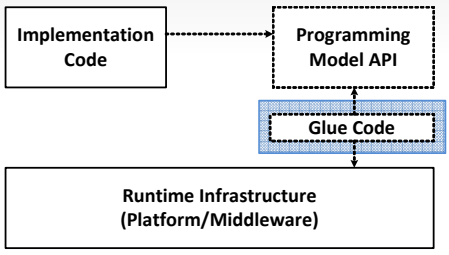
Runtime Infrastructure

Select based on fit wrt. to architectural concepts and non-functional requirements



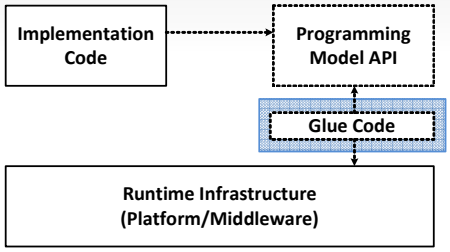
Glue Code

Aka Technology Mapping Code
Maps API to selected platform



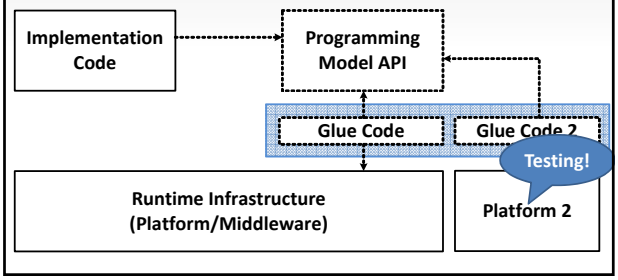
Glue Code

Contains Configuration Files for Platform
Might require „mix in models“



Several Platforms

Different Platforms, not Languages
Support for Scaling (non-functional req)

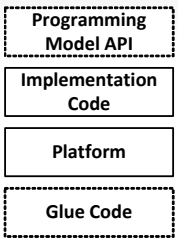


Benefits:

More Efficient Impl.
Technology Independent
Consistence/Quality
Architecture-Conformance

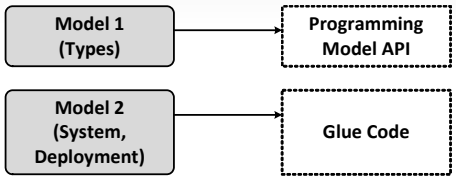
Code Gen Sequence

1) Generate API
2) Write Impl Code
3) Select Platform
4) Generate Glue Code



Separate Models

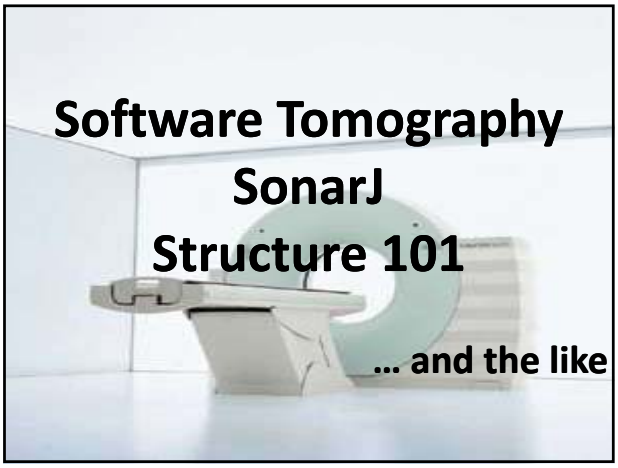
for stuff relevant for the API vs. system/deployment stuff



2.8

Theory and Concepts

Architecture Analysis Tools



Software Tomography
SonarJ
Structure 101

… and the like

How do you describe Architecture?

Constraint-Driven:

Assemble Packages into Components
Define layers/columns
Define and Check valid Dependencies

You do not get a way to define and express architectural abstractions beyond those!

Dependency Analysis
Anti Pattern Detection
Metrics
[Trend Analysis]
[Visualization]

Police Approach:

Detect Problems After the fact!

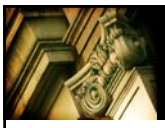
Anti Pattern Detection Metrics

Useful for manually written parts of the system

Dependency Analysis

Useful for manually written parts of the system

Constraints generated from the model.



2.9

Theory and Concepts

Standards,
UML and ADLs

Formal Architecture Description **is not new:**
ADLs, UML

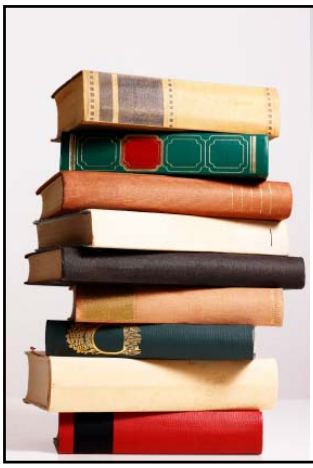
But all of those use
existing, generic
languages!

This misses the point!

Trying to express **your specific** architecture with **predefined** abstractions is not useful!

You want to build a language to capture your own architectural abstractions as your learn things.

Where are standards useful?



People have to learn architectural concepts anyway.

Is UML with a profile still a standard language?

On which meta level do I want to standardize? M2 (UML), M3 (MOF)?

Isn‘t a DSL based on MOF as „standard“ as a profile based on UML?

UML Profiles instead?
You‘ll think more about UML-ities than your own concepts

UML Profiles instead?
Tool integration issues (repository, diff/merge, versioning)

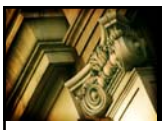
UML Profiles instead?
Tools are often complex, heavyweight, bloated.
Acceptance limited.

UML Generally Useless?
No. UML can be used for docu-mentation (sequence diagrams, eg)

NB: You can do all of this with UML!

However, my exprience shows it is much less productive and agile.

If you have a UML-based organization, use UML.



2.10

Theory and Concepts
Why not use a 3GL?

Architectural Abstractions are not first class citizens in 3GLs

Classes can represent everything – especially with annotations (meta data)

Problems:
shoehorning into classes; like UML

Problems:
annotations = stereotypes

Problems:
limited analyzability (not a model)

Problems:
mix architectural and implementation concerns

2.11

Theory and Concepts

My Notion of Components

Definitions Abound:
A Building Block for software systems

Definitions Abound:
Something with *explicit* context dependencies only

Definitions Abound:
Something that contains business logic and runs in a container

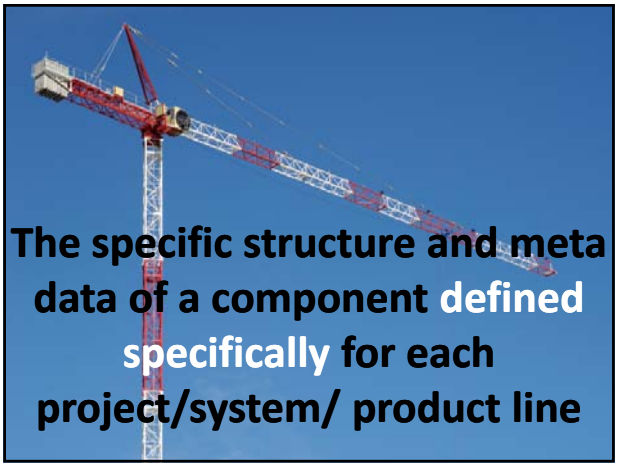
My Understanding:
Smallest architecturally relevant building block

My Understanding:
Black Box - Architecturally

My Understanding:
All architecturally relevant aspects specified declaratively (model)

My Understanding:
Analyzable, Composable by tools

My Understanding:
Running in container, provides technical services at component boundary

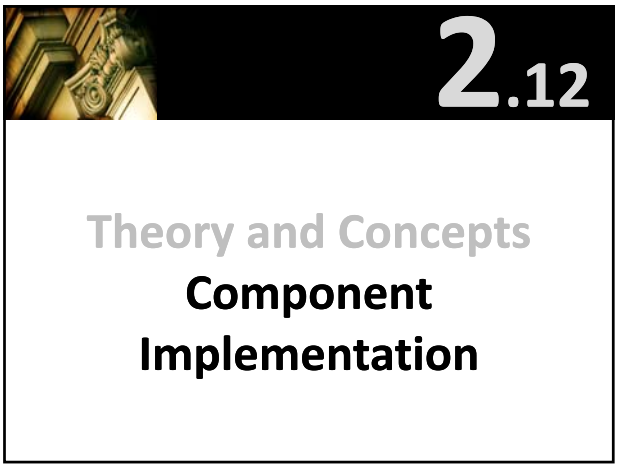


The specific structure and meta data of a component defined specifically for each project/system/ product line

This is what we do with the ...

Architecture

DSL



By Default:
Manualy Programming against the Prog Mod API

Alternatives:
Generate, based on config

Alternatives:
Generate, based on config
State Machines

Alternatives:
Generate, based on config
State Machines
Rule Languages + Engines

Alternatives:

Generate, based on config

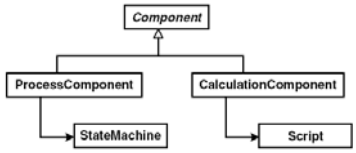
State Machines

Rule Languages + Engines

Specific DSL

specific abstractions in the architecture DSL.

ADD THE CASCADING STUGGF HERE



Note:

A lot of implementation can be generated automatically (no config) from the model

(persistence, serialization, …)

2.13

Theory and Concepts

The Role of Patterns

Pattern:

Proven solutions to recurring problems, including their applicability, trade-offs and consequences.

Architecture Patterns:
"Blueprint" for a specific architectural style

Architecture Patterns:
"Blueprint" for a specific architectural style

➔ Inspires the concepts for architecture DSL

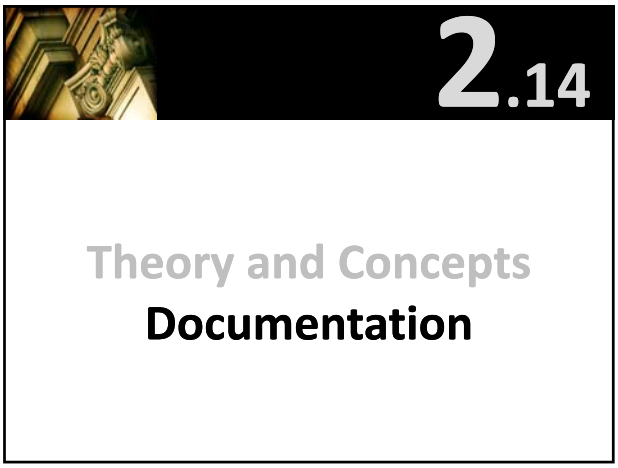
Design Patterns:
More concrete, more implementation-related than arch. patterns

Design Patterns:
More concrete, more implementation-related than arch. patterns
➔ Typically does not end up in the DSL, but …

Design Pattern:
More concrete, more implementation-related than arch. patterns
➔ put into the code generator

It's not the code generator who decides about the implementation of a pattern – it's still the (generator) developer!

A Pattern is more than the solution UML diagram!



The DSL and the „programs“ are documentation.

So I don‘t have to write…
… more docs?

Not quite.

1

Grammar is a good formal definition, but not a teaching tool.

Grammar is a good formal definition, but not a teaching tool.

➔ Tutorials

Tutorials:

Architectural Concepts (Meta Model)
How to use the language
How to use the programming model
(How to generate code)
(How to add manual code)

Tutorials:

Architectural Concepts (Meta Model)
How to use the language
How to use the programming model
(How to generate code)
(How to add manual code)

Example Driven!

2

Grammar describes what the architecture looks like.

Grammar describes what the architecture looks like.

But: Why is it that way?

Rationales

Rationales

Conceptual Architecture

Relate to Grammar

Rationales

Conceptual Architecture

Relate to Grammar

Technological Decisions

Non-Func. Req.

2.15

Theory and Concepts

Expressing Variability

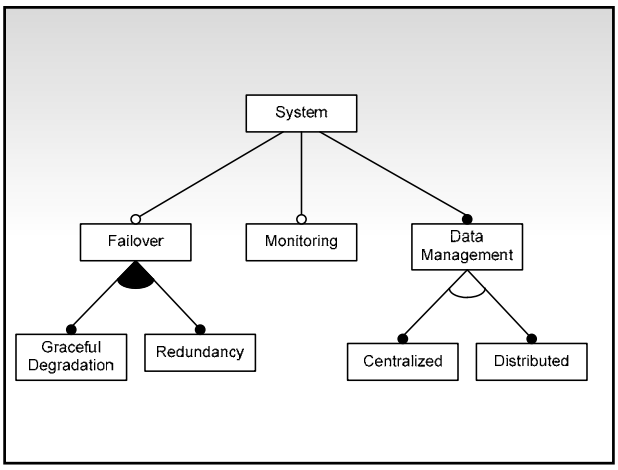
Different **Variants** of the System for different customers.

How do I **express** this in the models?

Negative Variability: Conditionally taking something away

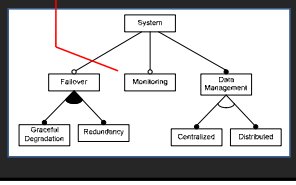
Negative Variability: Conditionally taking something away

Feature Models

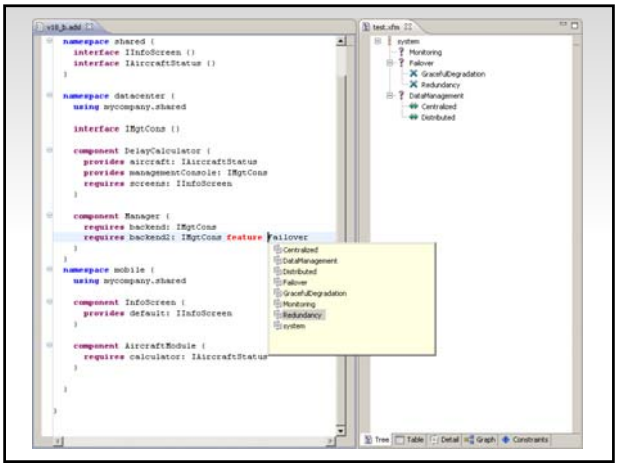



```
component DelayCalculator {
  provides default: IDelayCalculator
  requires screens[0..n]: IInfoScreen
  provides mon: IMonitoring feature monitoring
}
```

```
component DelayCalculator {
  provides default: IDelayCalculator
  requires screens[0..n]: IInfoScreen
  provides mon: IMonitoring feature monitoring
}
```




```
namespace monitoringStuff feature monitoring {

  component MonitoringConsole {
    requires devices:[*]: IMonitor
  }

  instance monitor: MonitoringConsole

  dynamic connect monitor.devices query {
    type = IMonitor
  }

}
```



Positive Variability:
Conditionally adding something to a minimal core

Positive Variability:
Conditionally adding something to a minimal core

Aspects

```
namespace monitoring {

  component MonitoringConsole …
  instance monitor: …
  dynamic connect monitor.devices …

  aspect (*) component {
    provides mon: IMonitoring
  }
}
```

```
component DelayCalculator {
  …
}
component AircraftModule {
  …
}
component InfoScreen {
  …
}
```

```
component DelayCalculator {
  …
}
component AircraftModule {
  …
}
component InfoScreen {
  …
}
```

```
aspect (*) component {
   provides mon: IMonitoring
}
```

```
component DelayCalculator {
  …
  provides mon: IMonitoring
}
component AircraftModule {
  …
  provides mon: IMonitoring
}
component InfoScreen {
  …
  provides mon: IMmonitoring
}
```

Weaver is **generic**:

works with all (container) model elements

aspect (*) *<type>*

all instances of *type*

aspect (tag=bla) <type>

all instances with tag bla

aspect (name=S*) <type>

all instances whose name starts with S

AO + Features

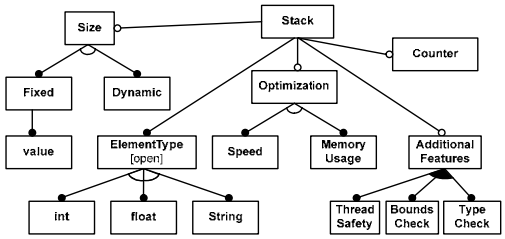
```
namespace monitoring feature monitoring {

  component MonitoringConsole …
  instance monitor: …
  dynamic connect monitor.devices …

  aspect (*) component {
    provides mon: IMonitoring
  }
}
```

2.16

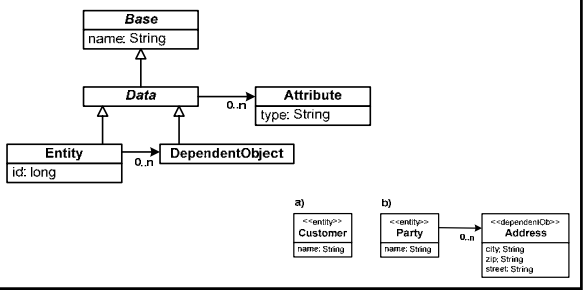
Theory and Concepts

PLE – Big Picture

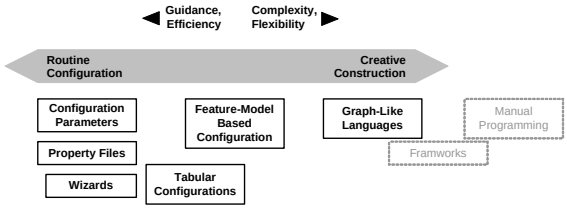
Configuration
vs. Customization



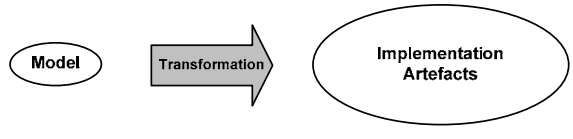
Customization
vs. Configuration



Customization vs. Configuration



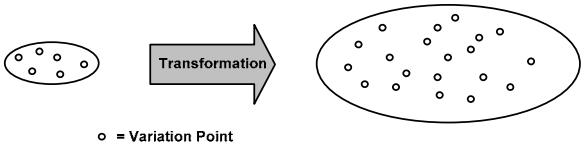
MDSD - Thumbnail



more abstract
less detailed

less abstract
more detailed

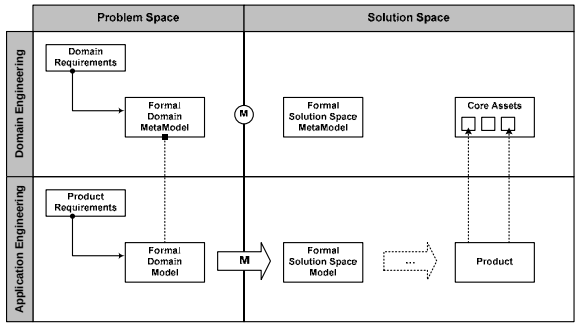
MD-PLE - Thumbnail



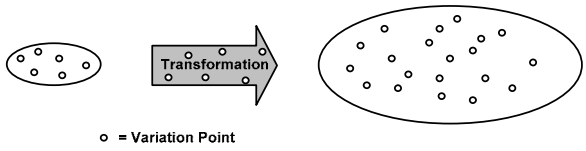
= Variation Point

fewer!

MD-PLE: Models



MD-PLE – Thumbnail II

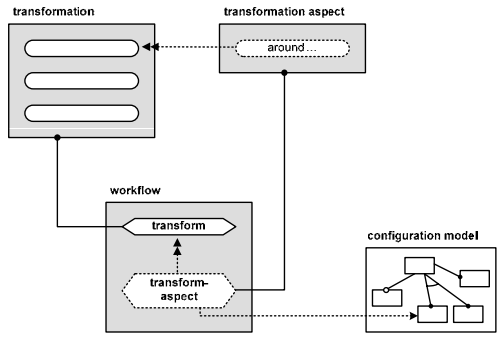


o = Variation Point

more options

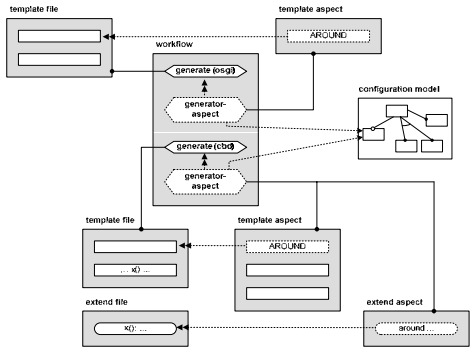
Variability in the process chain

Transformation AO (oAW)



Variability in the process chain

Generator AO (oAW)



2.17

Theory and Concepts

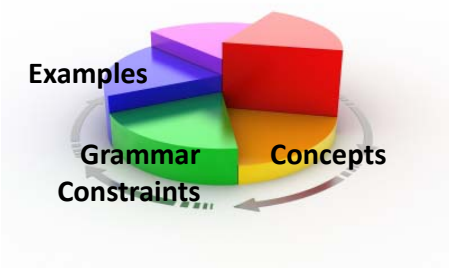
Process

Iterate!

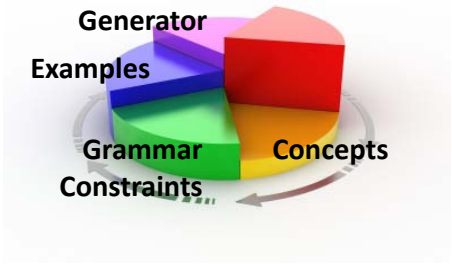
Iterate!

Iterate!

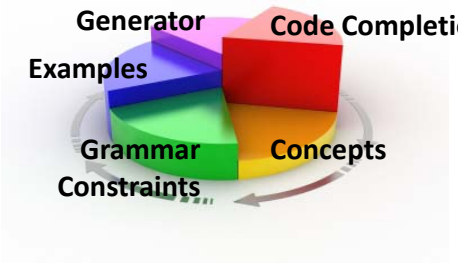
Iterate!



Iterate!



Iterate!



Reviews

still necessary!

Reviews

still necessary!
but simplified, because
architecture is more explicit

Architectural Concepts
Grammar, Constraints
Examples
Programming Model

Editor Customization
(API) Code Generator

Glue Code Generator

Uses Language
Generates Code
Writes Manual Code
against gen. API Code

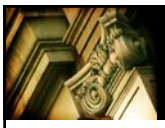
New language for each system?

Common Base?

Reusable Language Modules?

Language Family? PLE?

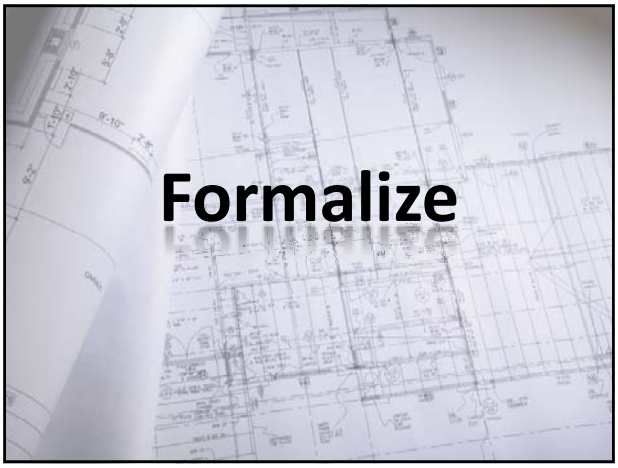
Another Talk!



2.18

Theory and Concepts

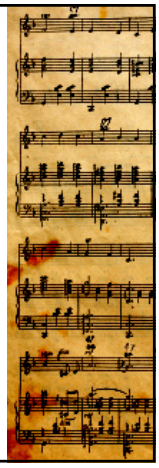
Underlying Concepts



Declaration

Implementation

Notation

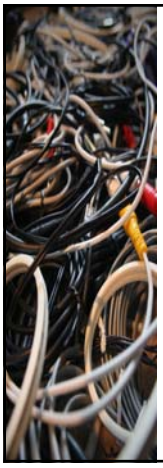


Types & Instances

Viewpoints

Hierarchical Decomposition

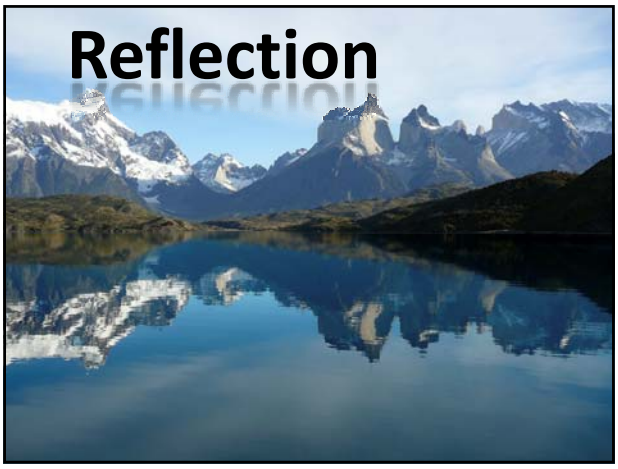
Translate



Handle Crosscuts

Go Meta

Reflection



Automate

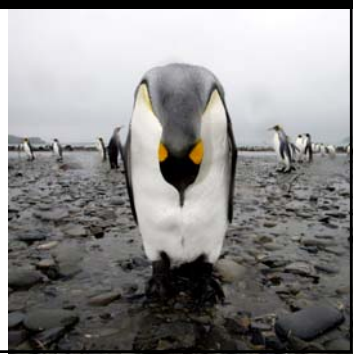
Orthogonality

Contracts

Isolate Technology

Don‘t Overspecify

Make Explicit

Platforms

THE END.
Thank you.
Questions?