Architecture As Language

Markus Voelter

www.voelter.de
voelter@acm.org

About

Markus Voelter
Independent Consultant/itemis

About

Markus Voelter
Independent Consultant/itemis

Software Architecture
DSLs & MDSD
Product Lines

About

Markus Voelter
Independent Consultant/itemis

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

Based on actual practical experience

Currently in use with several 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 Tooling

...

Part 2
Theory and Concepts

2.7	Model Validation
2.8	Generating Code
2.9	Standards, UML and ADLs
2.10	Component Implementation
2.11	Documentation
2.12	Process

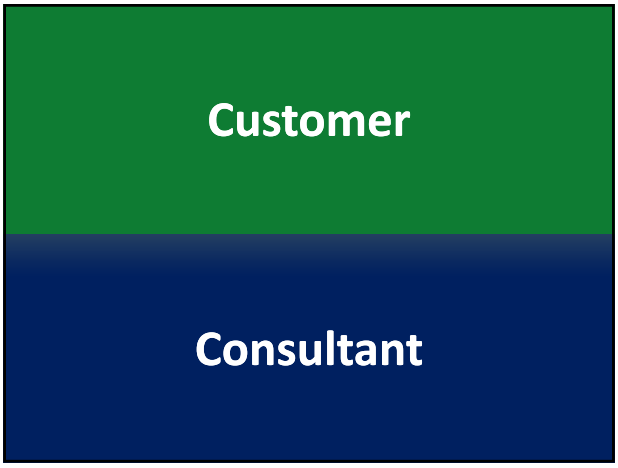
More Reading:

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

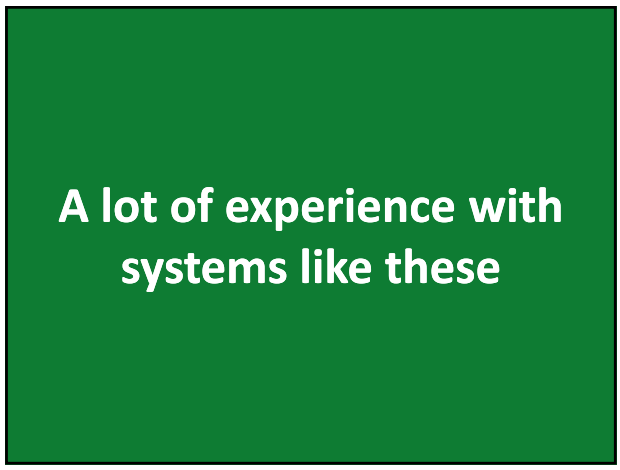


1.1
A Real-World Story
System Background

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

1.2

A Real-World Story

The Starting Point

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

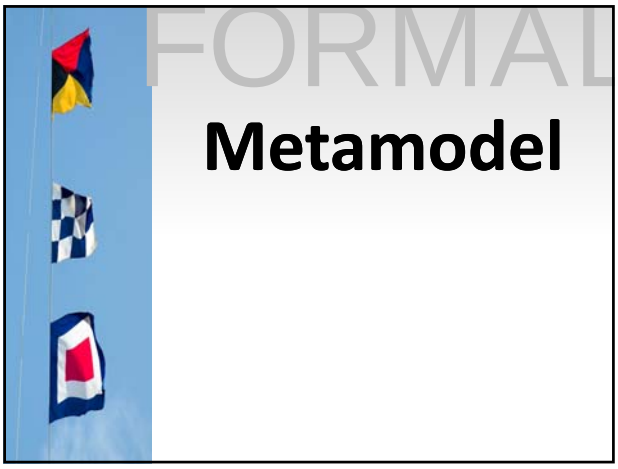
What is a language?

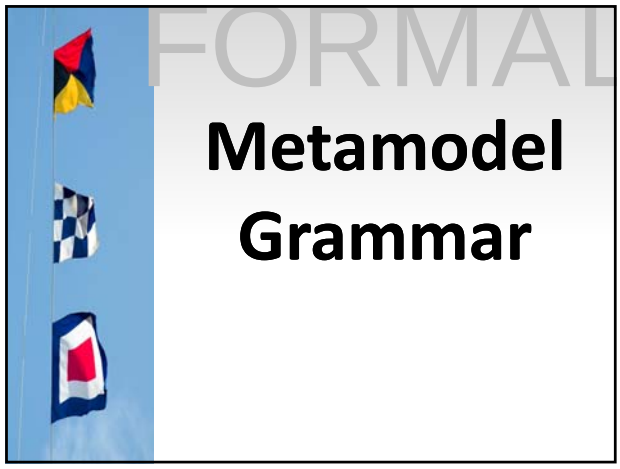
INFORMAL

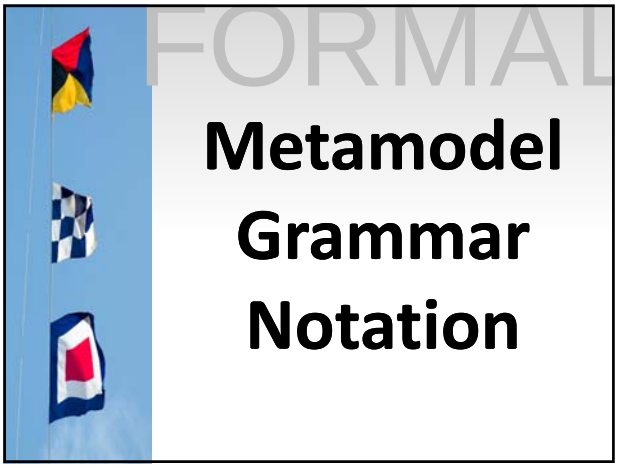
Set of well-defined terms

INFORMAL

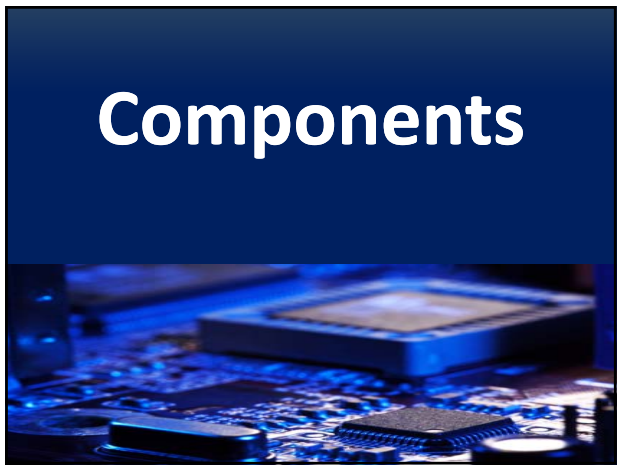
Stakeholders agree on meaning


FORMAL
Metamodel


FORMAL
Metamodel
Grammar


FORMAL
Metamodel
Grammar
Notation

1.4
A Real-World Story
Developing the Language


Components

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.**

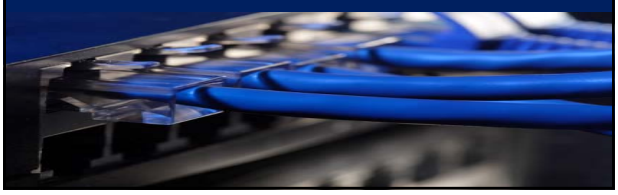
One DelayCalculator typically talks to many InfoScreens

Port:
Communication Endpoint

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

component InfoScreen {
  provides default: IInfoScreen
}

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

How to actually connect instances and their ports?

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)
```

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
  }
}
```

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 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…

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

replicated singleton flights {
  flights: FlightInfo[]
}

component DelayCalculator {
  publishes flights
}

component InfoScreen {
  consumes flights
}
```

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

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

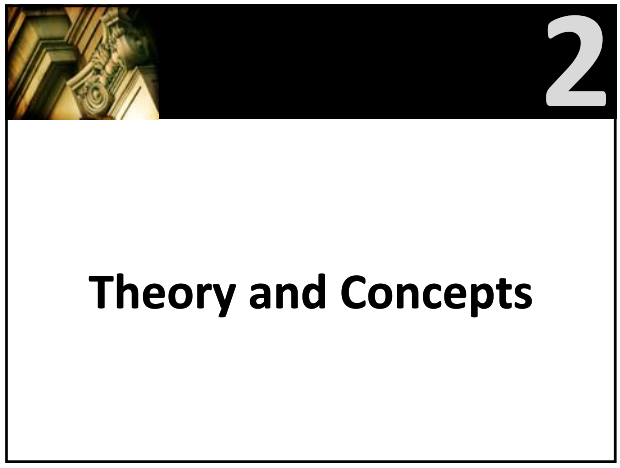
We still need messages.
Semantics?

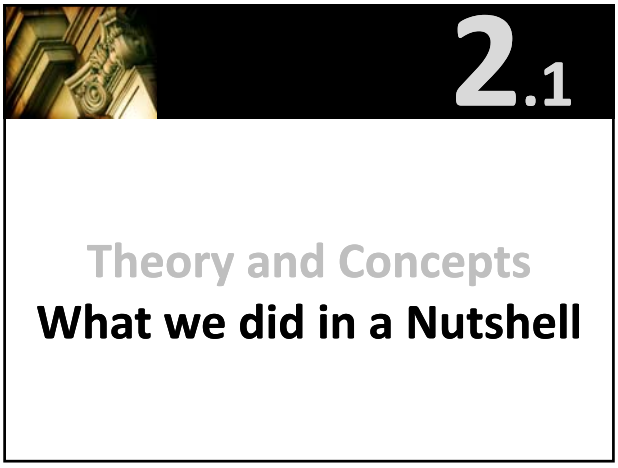
Message Sequences:
Protocol State Machines

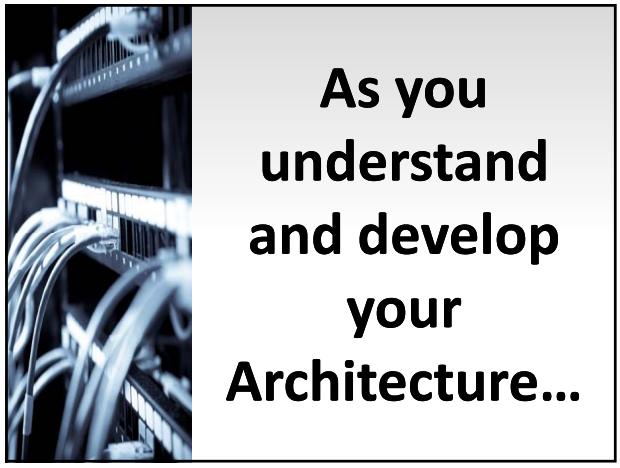
```
interface IAircraftStatus {
  oneway message registerAircraft(aircraft: ID! )
  oneway message unregisterAircraft(aircraft: ID! )
  oneway message reportPosition(aircraft: ID!,
                pos: Position! )
  request-reply message reportProblem {
    request (aircraft: ID!, problem: Problem!,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

None of those things is new or unique.

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



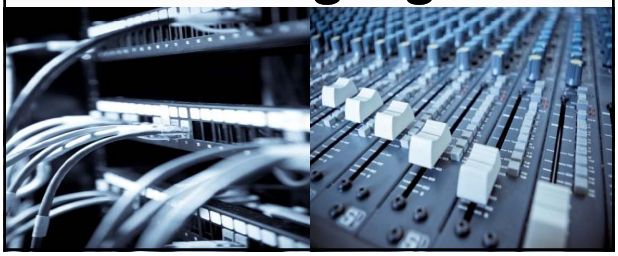




Develop a language to express it!

Language resembles architectural concepts

We express the application(s) with the language.



Architecture

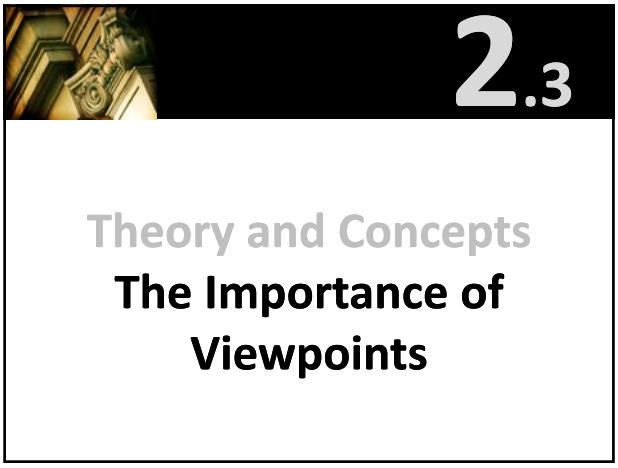
DSL

2.2

Theory and Concepts

Domain Specific Languages

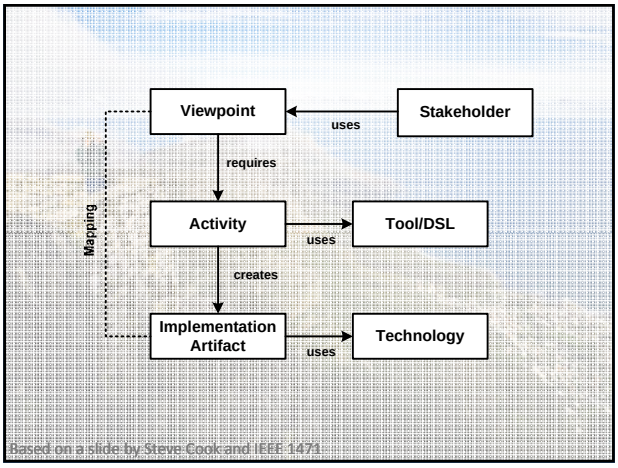
A DSL is a **focussed**, **processable language** for describing a specific **concern** when building a system in a specific **domain**. The **abstractions** and **notations** used are natural/suitable for the **stakeholders** who specify that particular concern.

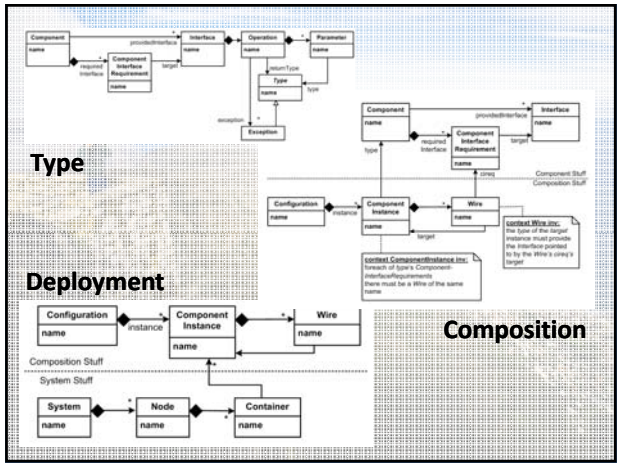


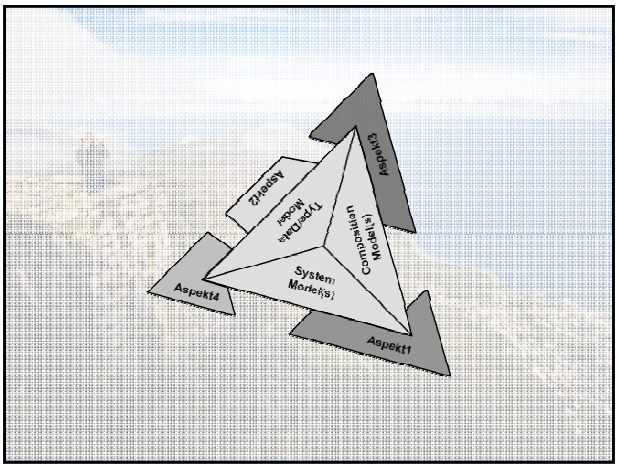
2.3

Theory and Concepts

The Importance of Viewpoints



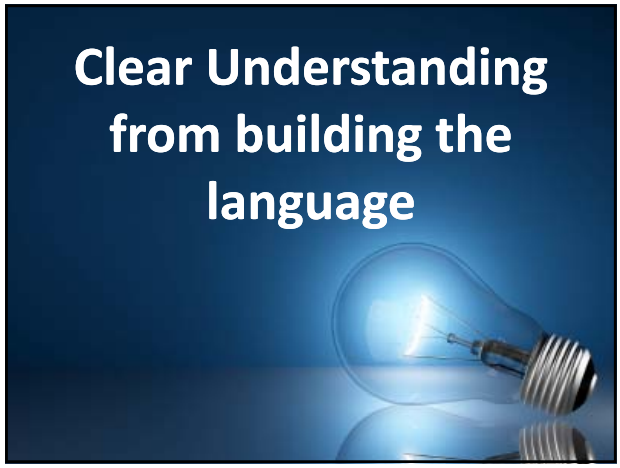




2.4

Theory and Concepts

Benefits



Clear Understanding from building the language

Unambigious Vocabulary

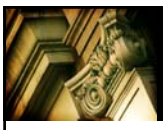
Concepts independent from Technology

Programming Model can be defined based on Conceptual Arcitecture

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

2.5

Theory and Concepts
Why Textual?



2.5

Theory and Concepts
… or: why not graphical?

Languages and Editors are easier to build

Languages and Editors are easier to build

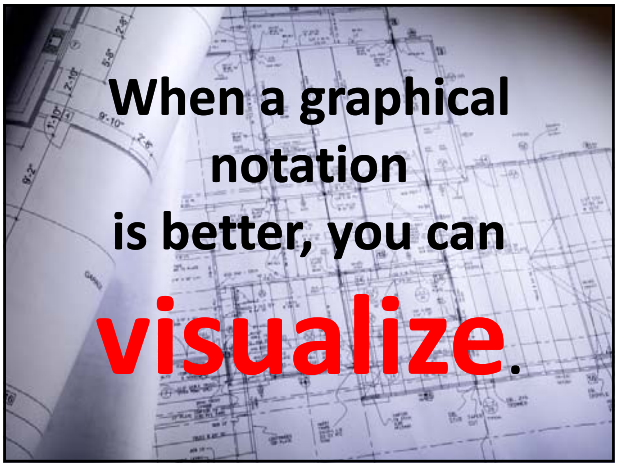
Evolve Language and simple editor as you understand and discuss the architecture, in real time!

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*.

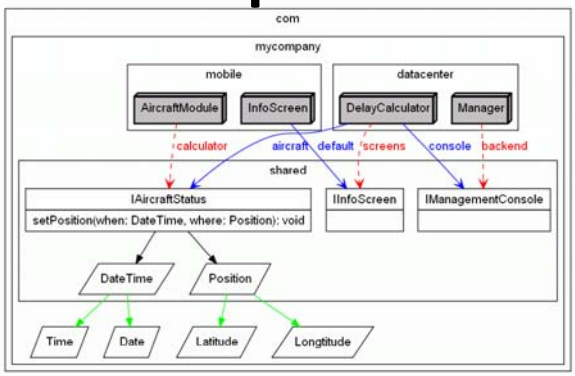
Many Developers prefer textual notations



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.

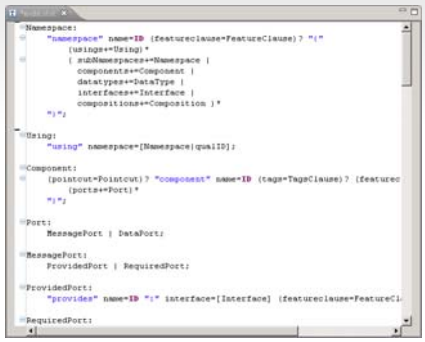
But: We've been building software with text only for a long time. We know it works.

2.6

Theory and Concepts
Tooling

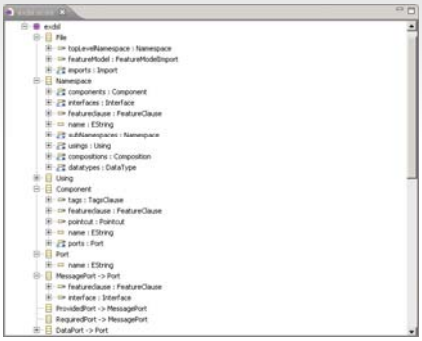
Several tools available.
Example: oAW Xtext

Specify Grammar

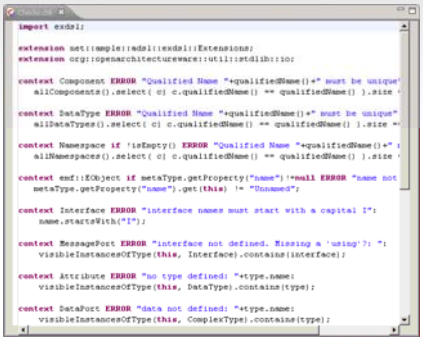


Antlr Grammar and
Parser is generated
from this specification

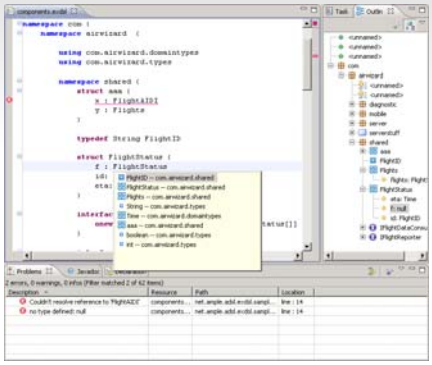
Generated Metamodel



Specify Constraints

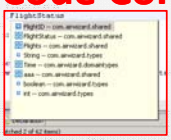


Generated Editor

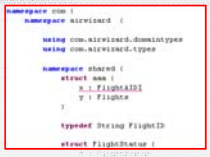


Generated Editor

Code Completion



Generated Editor

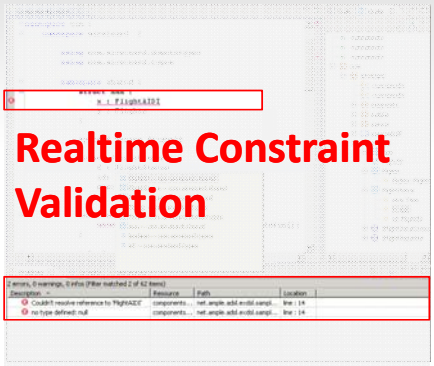


Syntax Coloring

Custom Keyword Coloring

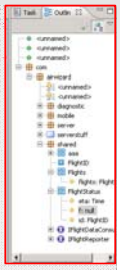
Generated Editor

Realtime Constraint Validation



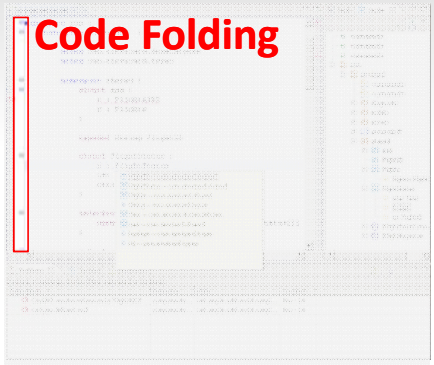
Generated Editor

Customizable Outlines



Generated Editor

Code Folding



Generated Editor

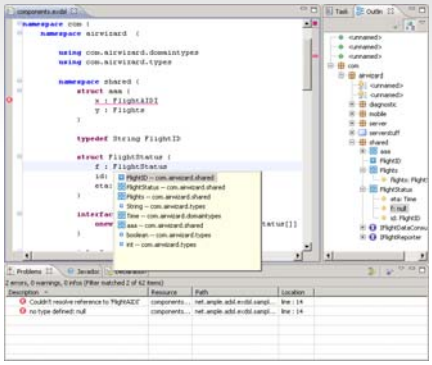
Goto Definition

Find References

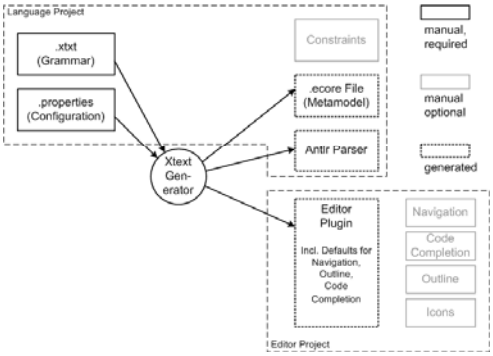
Cross-File References

Model as EMF

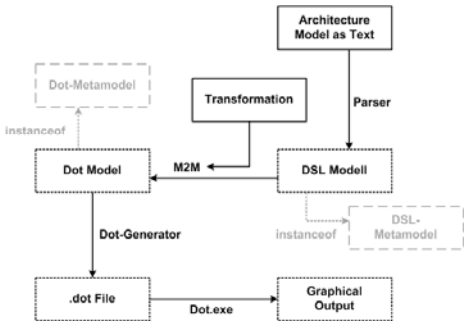
Generated Editor



Xtext Overview



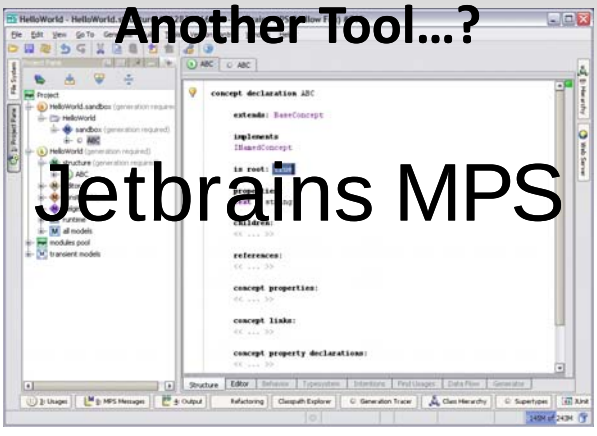
GrapViz Overview



GrapViz Trafo Code

Another Tool…?

Another Tool…?



Another Tool…?

INTENTIONAL™ SOFTWARE

2.7

Theory and Concepts

Model Validation

Grammar is not expressive enough

Simple Examples:

Name-Uniqueness

Type Checks

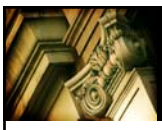
Non-Nullness

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



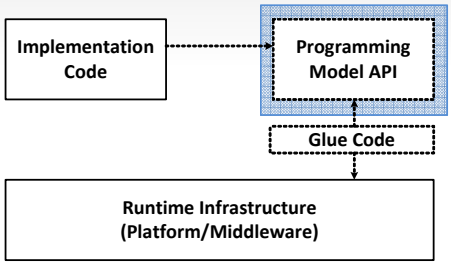
2.8

Theory and Concepts
Generating Code

Since we already have a formal model….

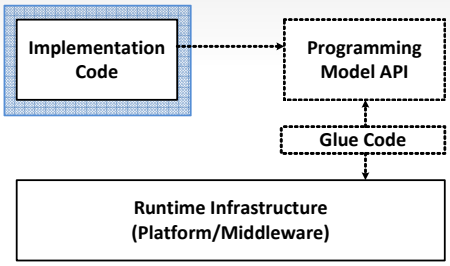
Generate API

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



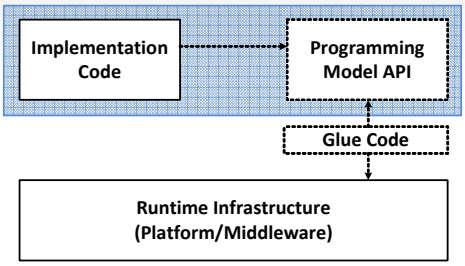
Implementation

Implementation only depends on the generated programming model API



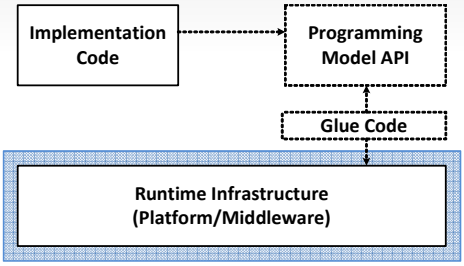
Programming Model

Generated API + Usage Idioms
Completely Technology-Independent



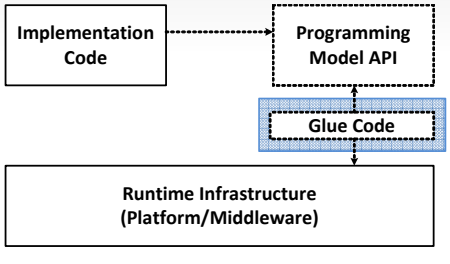
Runtime Infrastructure

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



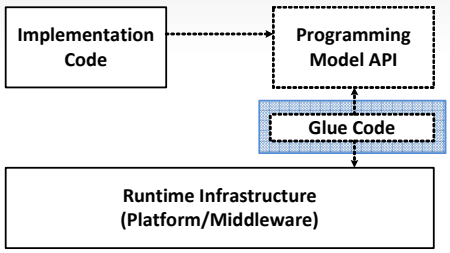
Glue Code

Aka Technology Mapping Code
Maps API to selected platform



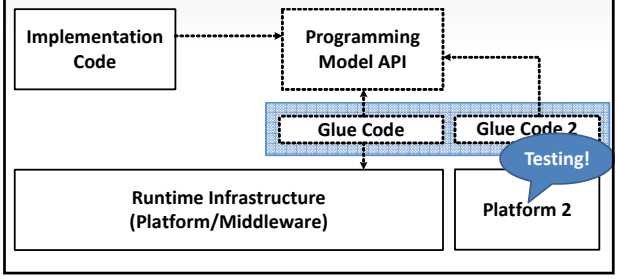
Glue Code

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



Several Platforms

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

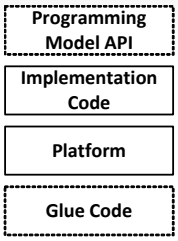


Benefits:

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

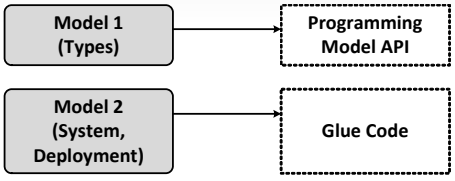
Code Gen Sequence

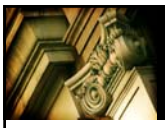
1) Generate API
2) Write Impl Code
3) Select Platform
4) Generate Glue Code



Separate Models

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





2.9

Theory and Concepts

Standards, UML and ADLs

Formal Architecture Description is not new: ADLs, UML

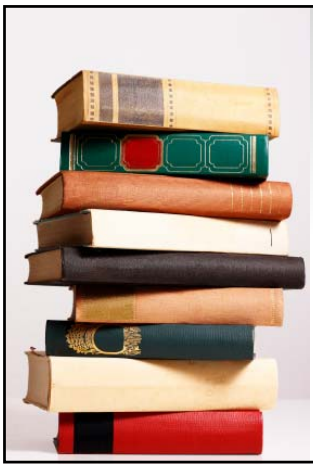
But all of those use existing, generic languages!

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
Component Implementation

By Default:
Manualy Programming against the Prog Mod API

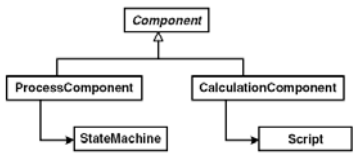
Alternatives:
Generate, based on config

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

To integrate behavior, **create specific abstractions** in the architecture DSL.



Note:
A lot of implementation can be generated automatically (no config) from the model
(persistence, serialization, …)

2.11

Theory and Concepts
Documentation

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.

2.12

Theory and Concepts

Process

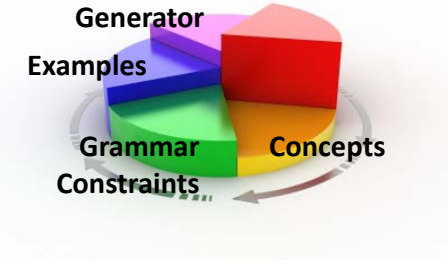
Iterate!

Iterate!

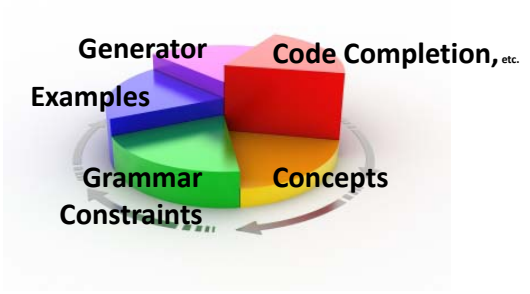
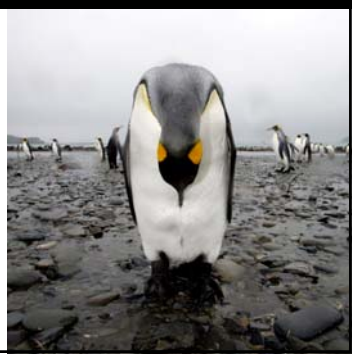
Iterate!

Iterate!

Iterate!



Iterate!



THE END.
Thank you.
Questions?