Software Architecture

Benjamin Satzger
Distributed Systems Group

TU Wien

http://www.infosys.tuwien.ac.at/staff/bsatzger
Terms

- Models
  - Unified Modeling Language (UML)
  - Architecture Description Language (ADL)
  - Domain-specific Language (DSL)

- Model-driven Software Development (MDSD)
  - Models are not only used for documentation, but as central artifacts
  - Also called Model-driven Development (MDD) or Model-driven Engineering (MDE)

- Model-driven Architecture (MDA)
  - A specialization of MDSD
  - Defined using the Meta-object Facility (MOF)
Overview

- Modeling Architectures
- Model-Driven Design & Development
- Architectural Views
MODELING ARCHITECTURES
A model is a pattern, plan, representation (especially in miniature), or description designed to show the main object or workings of an object, system, or concept.


Models in software development: In software development a model is an abstract representation of a system.

– Vogel et al.: Software-Architektur
Meta-modeling

- To be able to **automatically process model data**, it must be **described precisely** in a formal language.
- To achieve this, models are typically described through models, which are then called **meta-models**.
- A model is hence the **instance** of its meta-model.
- The cascade of abstraction by creating a meta-model for models can be continued arbitrarily, leading to a number of **modeling-levels**.
Modeling Languages

- Meta-models are also used to describe modeling languages.
- Meta-models define the structure or abstract syntax of the modeling language.
- In addition, a concrete syntax is needed.
  - Textual or graphical notation to create models.
- Finally, the modeling languages require semantics that describe the meaning of the models.
  - Often the semantics are given informally.
  - Hence, these modeling languages are called semi-formal languages.
Architecture Description Languages (ADL)

- An ADL is a modeling language specifically designed to describe software and/or system architectures

Properties of ADLs:

- **Formal representation** of the architecture using a textual or graphical notation on a very high abstraction level

- **Readable by humans and machines**

- Can by **analyzed** with regard to architectural properties, such as consistency, completeness, performance, etc.

- Sometimes, support for **code generation**
ADL Main Concepts

- Components
  - Syntactic or semantic **specification of functional or non-functional aspects** of an architectural elements using **interfaces**
  - The **exported and imported** interfaces of a component are described

- Connectors
  - Components communicate via connectors that define **how components interact with each other**
  - Connectors can represent various communication means such as RPC, HTTP, Unix-Pipes, ...
ADL Main Concepts

- Architectural Configurations
  - Describes the architectural structure by defining how the components and connectors are connected with each other
Illustration of the ADL Main Concepts

Configuration

Connector 1

Component 1

Connector 2

Component 2

Component 3
Simple Client/Server Example in an ADL

Example in ACME (CMU):

System CS_System = {
Component customer = {Port send-request}
Component databaseAccess = {Port receive-request}
Connector rpc = {Roles {caller, callee}}
Attachments : {
  customer.send-request to rpc.caller;
  databaseAccess.receive-request to rpc.callee
}
}
Unified Modeling Language (UML): MOF

- The Meta Object Facility (MOF) provides basic meta-meta model elements to build meta-models
- The OMG’s meta-modeling architecture defines 4 modeling levels
- It is itself defined using the elements of the UML infrastructure
Four modeling levels of the OMG

Modeling Level | Model | Analogy
---|---|---
M3 | MOF | EBNF-Formalism
M2 | UML | Java Grammar
M1 | UML Model A, UML Model B | Java Program P
M0 | A specific instance of A, Another instance of A | Execution of Program P

conforms to 14
Unified Modeling Language (UML): MOF

- The UML specification itself is split into the **UML Infrastructure** and the **UML Superstructure** specifications.
- The **UML infrastructure** defines elements used in both the meta-meta-model of UML (MOF) and the superstructure.
- The **UML meta-model** (i.e., the language definition) is defined in the **UML superstructure**.
  - The infrastructure is merged into the superstructure.
Application of the UML Infrastructure in MOF and UML

Diagram:
- Infrastructure Library
  - Core
  - Profiles
- MOF
- UML [Superstructure]

Relationships:
- «uses» from Core to Profiles
- «instanceOf» from MOF to UML [Superstructure]
UML diagram overview

Diagram

Structure Diagram
- Class Diagram
- Composite Structure Diagram
- Deployment Diagram
- Package Diagram

Behavior Diagram
- Activity Diagram
- Use Case Diagram

Interaction Diagram
- Sequence Diagram
- Communication Diagram
- Interaction Overview Diagram
- Timing Diagram

Notation: UML

UML2 Meta-Model Excerpt for Defining Component Architectures
Example Component Model in UML2

Student

Manager

Office

Course

ICourseForStudent

ICourseForManager

«uses»
Example: Modeling an Architectural Variant of the Adapter Pattern
On Modeling Architectures with UML

- UML supports modeling the **component & connector view**, import and export of interfaces, architectural configurations
- Little support for specific architectural properties, e.g., for analysis, as in ADLs
- Little support for modeling system integration because system integration concepts are missing
  - desktop application, server application, hosted application, …
  - messaging, queues, pub/sub, …
  - databases, file systems, …

Extending the UML Meta-Model

- According to the UML standard there are two ways to extend the language:
  - the hard extension produces an extension of the language meta-model, i.e., a new member of the UML family of languages is specified
  - the soft extension results in a profile, which is a set of stereotypes, tag definitions, and constraints that are based on existing UML elements with some extra semantics according to a specific domain
Defining UML Profiles

- A **stereotype** can extend any element (**meta-class**) of the meta-model
  - new types of classes, components, actors, …
  - new types of relationships
  - new features like attributes

- (OCL) **Constraints** can be used to formally define the semantics of the stereotyped meta-classes
Example: Stereotypes for modeling Callback

```
<<stereotype>>
IEvent

<<stereotype>>
ICallback

<<stereotype>>
EventPort

<<stereotype>>
CallbackPort

<<metaclass>>
Interface

required
provided

<<metaclass>>
Port

<<stereotype>>
Callback

<<metaclass>>
Connector
```
Example: Stereotypes for Modeling Callback

[Diagram showing stereotypes for modeling callback]
Example: OCL Constraints for Defining Callback Components

-- An event port is typed by IEvent as a
-- provided interface
inv: self.basePort.required->size()=1
   and self.basePort.required->forAll(
      i:Core::Interface|
      ICallback.baseInterface->exists(j|j=i))

-- And: An event port is typed by ICallback
-- as a required interface.
inv: self.basePort.provided->size()=1
   and self.basePort.provided->forAll(
      i:Core::Interface|
      IEvent.baseInterface->exists(j|j=i))
Example: OCL Constraints for Defining Callback Components

-- A callback port is typed by ICallback as a provided interface
inv: self.basePort.required->size()=1
   and self.basePort.required->forAll(
      i:Core::Interface|
      IEvent.baseInterface->exists(j|j=i))

-- And: A callback port is typed by IEvent as a required interface.
inv: self.basePort.required->size()=1
   and self.basePort.required->forAll(
      i:Core::Interface|
      ICallback.baseInterface->exists(j|j=i))

-- A Callback connector has only two ends.
inv: self.baseConnector.end->size()=2
Example: OCL Constraints for Defining Callback Components

-- A Callback connector connects an EventPort
-- of a component to a matching CallbackPort of
-- another component. An EventPort matches a
-- CallbackPort if the provided IEvent interface
-- of the former matches the required IEvent
-- interface of the latter, and the required
-- ICallback interface of the former matches
-- the provided ICallback interface of the latter:

inv: self.baseConnector.end->forAll(
  e1,e2:Core::ConnectorEnd|e1<>e2 implies(
    (e1.role->notEmpty() and e2.role->notEmpty()) and
    (if EventPort.basePort->exists(p|
      p.oclAsType(Core::ConnectableElement)=e1.role)
    then (CallbackPort.basePort->exists(p|
      p.oclAsType(Core::ConnectableElement)=e2.role)
    and
      e1.role.oclAsType(Core::Port).required=
      e2.role.oclAsType(Core::Port).provided
    else CallbackPort.basePort->exists(p|
      p.oclAsType(Core::ConnectableElement)=e1.role)
    endif))
Example: Stereotypes for Modeling Callback
Example: Using the Stereotypes in a Case Study
Model-driven Software Development

- Models are not only used for documentation, but as central artifacts
  - Parts of the system are specified in models and the system is generated from these models
  - Such models must be precise and as expressive as possible
- Model Driven Architecture (MDA) of the OMG is one specific flavor of model-driven software development (MDSD or MDD)
Model-driven Software Development: Relevant Concepts

- Application Model
  - defined in DSL Concrete Syntax
  - 1..* use
  - 1..* Transformation
  - produces Schematic Recurring Code

- Meta-Model (DSL Abstract Syntax)
  - 1 based on Meta-Model
  - 1..* defined using Transformation
  - represents

- Meta-Meta-Model
  - 1 based on
  - uses Individual Code

Distributed Systems Group
Domain

- The starting point usually is a domain

A domain is a bounded area of knowledge or interest.

Examples (from the world of Software) include:
- eBanking
- Embedded Software
- Web-Based eBusiness Applications
- Control Software for 4-Cylinder Diesel Engines
- Astronomical Image Processing Software

Domain-driven Design and Development

- **Domain-Driven Design and Development** means to make software development domain-related and not (just) SW-technology-related
  - Goal: Make software development in a certain domain more efficient
  - This is achieved by mapping (business) domain concepts into software artifacts using a **domain model**
- A domain model establishes a **common model** between the (business) domain and IT stakeholders
Domain-Specific Languages (DSLs)

- A DSL is a *tailor-made (computer) language for a specific problem domain*
  - DSLs differ from general purpose languages (GPL), such as C, C#, Java, Perl, Ruby, or Tcl, that can be applied to arbitrary problem domains
  - DSLs are often not Turing complete and only provide abstractions suitable for one particular problem domain
- This specialization results in **significant gains in expressiveness and ease of use**
  - Often **less code** is needed to achieve a result
  - Often DSL code is rather **easy to read and comprehend**
Embedded vs. External DSLs

- **Embedded DSL** (also called internal DSL) is defined as an extension to an existing GPL
  - It uses the *syntactic elements of the underlying GPL*
  - It can **directly access all features and tools** of the host language
  - Instead of using a generator, often embedded DSLs are **interpreted**
  - Typical examples of embedded DSLs are DSLs in **scripting languages** and **dynamic languages**, such as Ruby, Python, Tcl, or Perl
Embedded vs. External DSLs

- An **External DSL** is defined in a different format than the intended target language(s)
  - It can use **all kinds of syntactical elements**
  - DSL designers may define **any possible syntax**, be it **textual** or **graphical**
  - An external DSL is **not bound to a certain host language or platform**
  - It can be mapped to target platforms via **transformations**
MDSD: How it works

- Developer/designers develops **models** based on a meta-model
  - Often models and meta-models are expressed using a DSL

- **Code generation** (e.g., using templates) is used to transform the model into executable code
  - Alternatively, the models can be interpreted
  - In most cases, only schematic, recurring code is generated; other code is written manually
  - In between, model-to-model transformations can happen
Example Generation Process

Constraints

```plaintext
# correlation group package must be stereotyped as CorrelationGroup
CorrelationIdentifier:addInvariant | 
 | [FOL forAll cg (self.correlationGroup)] |
 | [FOL allInstances iFOL allInstances Correlation:CorrelationGroup | |
 | [$cg basePackage == $cg ] |
 | ]}
#
# each CorrelationGroup must have a CorrelationIdentifier
CorrelationGroup:addInvariant | 
 | [FOL exists cid | [FOL allInstances Correlation:CorrelationIdentifier | |
 | [$cid correlationGroup == [self basePackage] |
 | ]]
```

Transformation Templates

```plaintext
# # Template for the Sequence structure
# DEFINE Activity(core:View iv, core:View cv, core:View hcv) FOR orchestration:Sequence
 # <sequence>
 # <expand Activity(iv, cv, hcv) FOR EACH element>
 # </sequence>
 # ENDDEFINE
#
# # Template for the Flow structure
# DEFINE Activity(core:View iv, core:View cv, core:View hcv) FOR orchestration:Flow
 # <flow>
 # <expand Activity(iv, cv, hcv) FOR EACH element>
 # </flow>
 # ENDDEFINE
```
Example: J2ME App, Model in Graphical Syntax

NumberEntry
- a: Number ["a"] \{a>0\}
- b: Number ["b"] \{b>0\}

Form
- \["Abort"]
- \["Result"]

CalcResult
- \["Exit"]
- \["Again"]
- c: Number; \quad c = a + b

\["The Result is "+c\] \{align=center\}
Example: J2ME App, Abstract Syntax

**Element**

- **UIElement**
  - name: String

- **Form**

- **Display**
  - label: String
  - constraints: String

- **Field**
  - varName: String
  - label: String
  - constraints: String

- **Calculation**
  - declaredVars: String
  - expression: String

- **Start**

- **End**

**Flow**

- **AutoFlow**
  - transportedVars: String

- **Button**
  - label: String
  - transportedVars: String

**Relationships**

- 1 to 1
- 1 from

**Arrows**

- outbound *
- inbound *
Feasibility of MDSD

- MDSD requires:
  - Creating an infrastructure consisting of modeling tools, generators, platforms, etc.
  - Much effort for domain analysis

- These efforts do usually not pay off for using the MDSD infrastructure only once
  - Multiple incarnations of an MDSD infrastructure are more feasible
Software Product Lines and System Families

- Multiple incarnations of MDSD often leads to **software product lines**
  - Idea: Identification of software system families
  - The members of the family have a number of **technical and domain concerns in common**
  - Often they rely on the same technical infrastructure

- Supporting **reuse** of
  - components and frameworks
  - models, generators, transformations
Options for Modeling Architectures Revisited

- Use informal box-and-line diagrams
- Use an existing ADL that fits your modeling needs
- Use UML and extend the existing UML meta-classes using UML’s extension mechanisms
- Define a **custom architecture meta-model** as a foundation to define your own architectural DSL

Often not precise enough (e.g., for MDSD)

Often no ADL fits exactly

Sometimes this is awkward and tedious and UML has a steep learning curve
Benefits of a Custom Architecture Meta-model / DSL

- Communicating the architecture among stakeholders can be specifically supported and eased

- It can support architecture documentation:
  - The custom architecture meta-model can serve as a glossary of architecture concepts used
  - Its models can serve as a glossary of the components, interactions, etc.

- The act of formalizing the architecture often means to **clarify** the architecture, too

- The approach lends toward **technology-neutral** architecture definition
Example: Defining a Custom Architecture Meta-Model

- System
  - Node
  - Deployment
    - Parameter Value
    - Wire
    - Config Parameter
    - Component Dependency
      - Component
        - Business Component
          - must contain exactly one facade component
        - Domain Component
        - Facade Component

There must be a wire for each dependency of the deployed component.

Deployment must provide a parameter value for each parameter of the deployed component.
Example: Incrementally Building an Architecture DSL

- **Goals:**
  - Incrementally define an **Architecture DSL** for an existing flight management system that evolved over 10-15 years
  - **Reason:** Define the building blocks (types of things) from which the actual system will be built
  - **Abstract from technologies** used in this system
  - **Distributed system**, needs to be upgraded incrementally
    - **Versioning** of components is required

Example from:
[http://www.voelter.de/data/articles/ArchitectureAsLanguage-PDF.pdf](http://www.voelter.de/data/articles/ArchitectureAsLanguage-PDF.pdf)
Example: Incrementally Building an Architecture DSL

- Step 1: Define the notion of components as architectural building blocks

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

- Step 2: Define interface to allow for component interaction

```java
component DelayCalculator implements IDelayCalculator {}
component InfoScreen implements IInfoScreen {}
component AircraftModule implements IAircraftModule {}
interface IDelayCalculator {}
interface IInfoScreen {}
interface IAircraftModule {}
```
Example: Incrementally Building an Architecture DSL

- Problem: Components cannot yet use interfaces
- Step 3: Redesign to support interface export/import

```
component DelayCalculator {
  provides IDelayCalculator
  requires IInfoScreen
}
component InfoScreen {
  provides IInfoScreen
}
component AircraftModule {
  provides IAircraftModule
  requires IDelayCalculator
}
interface IDelayCalculator {}
interface IInfoScreen {}
interface IAircraftModule {}
```
Example: Incrementally Building an Architecture DSL

- Step 4: Define instances of components

  instance screen1: InfoScreen
  instance screen2: InfoScreen
  ...

- Step 5: Introduce Ports using the provides/requires keywords

  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 “wire” the system?
Example: Incrementally Building an Architecture DSL

- Step 6: Redesign of the DelayCalculator to support two role-specific export interfaces

```
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
}
```
Example: Incrementally Building an Architecture DSL

- Step 7: How to specify “wiring”

```java
instance dc: DelayCalculator
delay
instance screen1: InfoScreen
instance screen2: InfoScreen
connect dc.screens to (screen1.default, screen2.default)
```

For the complete example see: http://www.voelter.de/data/articles/ArchitectureAsLanguage-PDF.pdf
Architectural Views

View: A representation of a whole system from the perspective of a related set of concerns.

Viewpoint: A specification of the conventions for constructing and using a view. A pattern or template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis.

– IEEE Standard 1471, Architecture of a SW-intensive system
Architectural Views

- Architectural Views make complex systems understandable
  - Only look at the aspects of the system that are relevant for a task at hand
- Architectural Views should either support:
  - different stakeholders or tasks
  - or represent different architectural modeling or abstraction levels
Views for Architectural Modeling or Abstraction Levels

- Idea/Analogy: Introduce a zoom-factor for the architecture models
- Refinement of models in multiple levels
- Example: 3 Levels of Models
  - Organizational View (very-high-level): Structure of the organization in which the architecture is created
  - System View (high-level): Systems and system interconnections in the organizations and to external systems
  - Component View (low-level): Components of one system, functional and non-functional aspects
Use Case View: Central in the 4+1 View Model

- All architectural decisions are based on use cases and scenarios of the system in focus
- Foundation of all other architectural views
- Used for validating the other views
- This view can be described using case diagrams and use case specifications

Views of 4+1 View Model

- The **logical view** describes the (object-oriented system) system in terms of abstractions, such as classes and objects
  - The logical view typically contains class diagrams, sequence diagrams, and collaboration diagrams
  - Other types of diagrams can be used where applicable

- The **development view** describes the structure of modules, files, and/or packages in the system
  - The package diagram can be used to describe this view
Views of 4+1 View Model

- **The process view** describes the processes of the system and how they communicate with each other
  - Activity diagrams may be used to describe this view
- **The physical view** describes how the system is installed and how it executes in a network of computers
  - Deployment diagrams are often used to describe this view
Example: View-based Modeling Framework (VbMF)
Example: View-based Modeling Framework (VbMF)

Separation of concerns
Example: View-based Modeling Framework (VbMF)

Separation of technical and domain-oriented views
Many thanks for your attention!

Benjamin Satzger

Distributed Systems Group
TU Wien

http://www.infosys.tuwien.ac.at/staff/bsatzger