Unified Modeling Language 2.0

Part 1 – Introduction

Prof. Dr. Harald Störrle
University of Innsbruck
mgm technology partners

Dr. Alexander Knapp
University of Munich
1 – Introduction
History and Predecessors

• The UML is the “lingua franca” of software engineering.

• It subsumes, integrates and consolidates most predecessors.

• Through the network effect, UML has a much broader spread and much better support (tools, books, trainings etc.) than other notations.

• The transition from UML 1.x to UML 2.0 has
  – resolved a great number of issues;
  – introduced many new concepts and notations (often feebly defined);
  – overhauled and improved the internal structure completely.

• While UML 2.0 still has many problems, it is much better than what we ever had before.

(current version (“the standard”)
formal/05–07–04 of August ‘05)

(c) 2005–2006, Dr. H. Störrle, Dr. A. Knapp
1 – Introduction
Usage Scenarios

• UML has not been designed for specific, limited usages.

• There is currently no consensus on the role of the UML:
  – Some see UML only as tool for sketching class diagrams representing Java programs.
  – Some believe that UML is “the prototype of the next generation of programming languages”.

• UML is a really a system of languages (“notations”, “diagram types”) each of which may be used in a number of different situations.

• UML is applicable for a multitude of purposes, during all phases of the software lifecycle, and for all sizes of systems – to varying degrees.
1 - Introduction

Usage Scenarios
UML is a coherent system of languages rather than a single language. Each language has its particular focus.

<table>
<thead>
<tr>
<th>Structure</th>
<th>Behavior</th>
<th>Interaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class Diagram</td>
<td>Use Case Diagram</td>
<td>Sequence Diagram</td>
</tr>
<tr>
<td>Composite Structure Diagram</td>
<td>Activity Diagram</td>
<td>Communication Diagram</td>
</tr>
<tr>
<td>Component Diagram</td>
<td></td>
<td>Timing Diagram</td>
</tr>
<tr>
<td>Deployment Diagram</td>
<td></td>
<td>Interaction Overview Diagram</td>
</tr>
<tr>
<td>Package Diagram</td>
<td></td>
<td>State Machine Diagram</td>
</tr>
</tbody>
</table>

- **Structure**: static structure (generic/snapshot), logical system structure, physical system structure, computing infrastructure / deployment, containment hierarchy.
- **Behavior**: abstract functionality, control flow and data flow.
- **Interaction**: message exchange over time, structure of interacting elements, coordinated state change over time, flows of interactions, event-triggered state change.
1 – Introduction

Diagram types also depend on their usage

- Each diagram type may be used in a multitude of settings, for each of which different rules and best practices may apply.

- For instance, class diagrams may be used during analysis as well as during implementation.

- During analysis, this class diagram is bad, or at least suspicious.

- During implementation, it is bad if and only if it does not correspond to the code (or other structure) it is used to represent.
1 – Introduction

Internal Structure: Overview

• The UML is structured using a metamodeling approach with four layers.
• The $M_2$-layer is called metamodel.

• The metamodel is again structured into rings, one of which is called superstructure, this is the place where concepts are defined ("the metamodel" proper).

• The Superstructure is structured into a tree of packages in turn.
1 – Introduction
Internal Structure: Layers

- $M_3$: Meta-Metamodel
- EBNF
- Meta Object Facility (MOF)

- $M_2$: Metamodel
- Java grammar
- Unified Modeling Language (UML)
- Common Warehouse Metamodel (CWM)

- $M_1$: Model
- a Java program
- Albatros Air Autopilot

- $M_0$: System
- an execution of a Java program
- a runtime state in a deployment of Albatros Air Autopilot
1 – Introduction
Internal Structure: Layers

M₃ | Meta-Metamodel
---|---
M₂ | Metamodel
---|---
M₁ | Model
---|---
M₀ | System

Class

Attribute

Class
InstanceSpecification

Movie +title: String

m: Movie

Movie +title: "Fight Club"

(c) 2005–2006, Dr. H. Störrle, Dr. A. Knapp
1 – Introduction

Internal Structure: Rings
1 – Introduction
Internal Structure: Packages
1 – Introduction
Diagrams and models

Diagram name

(pragmatic)

Diagram kind

Represent

Datastructure, instance of the metamodel

Present

Model

(c) 2005–2006, Dr. H. Störrle, Dr. A. Knapp
1 – Introduction

UML is not (only) object oriented

- A popular misconception about UML is that it is “object oriented” by heart – whatever that means.

- It is true that
  - UML defines concepts like class and generalization;
  - UML is defined using (mainly) a set of class models;
  - UML 2.0 redisCOVERS the idea of behavior embodied in objects.

- However, UML 2.0
  - also encompasses many other concepts of non– or pre–OO origin (Activities, StateMachines, Interactions, CompositeStructure, …);
  - may be used in development projects completely independent of their implementation languages (Java, Cobol, Assembler, …);
  - is not tied to any language or language paradigm, neither by accident nor purpose.
1 – Introduction

UML 1.x vs. UML 2.0

<table>
<thead>
<tr>
<th>UML 1.x</th>
<th>UML 2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Collaboration diagram</td>
<td>• Communication diagram</td>
</tr>
<tr>
<td>• ActivityGraph is a StateMachine</td>
<td>• Activity is a Behavior (“Petri–like”)</td>
</tr>
</tbody>
</table>

New features in UML 2.0

- Activities: exceptions, streams, structured nodes, ...
- traverse-to-completion
- Timing diagram
- interaction overview diagram
- composite structure diagram
- interaction operators
- collaborations

Other novelties

- Detail changes everywhere
- New overall structure
1 – Introduction

UML 1.x vs. UML 2.0

• UML 2.0 has several advantages over UML 1.x:
  – many powerful new concepts
  – much better definitions (i.e. semantics)
  – improved internal structuring

• However, even though UML 2.0 is much better defined than UML 1.5, the state is still not satisfying, e.g.
  – syntax
    • overloaded notation: too many synonyms, too much sugaring,
    • lack of notational orthogonality, some people don’t even want this,
  – semantics
    • lack of precise semantics: informal, unclear and contradictory definitions,
  – pragmatics
    • lack of methodological basis such as model consistency conditions, usage types etc.

• Still, it’s the best comprehensive (“unified”) modeling language we ever had.
1 – Introduction

Wrap up

• UML is the lingua franca of software engineering.

• It has many problems, yet it is better than anything we had before.

• It may be used during the whole software lifecycle. UML may help to plan, analyze, design, implement, and document software.

• The UML is structured
  – by a 4-layer metamodeling approach
    \[ M_0: \text{system}, M_1: \text{model}, M_2: \text{meta model}, M_3: \text{meta meta model}, \]
  – the metamodel is structured into 3 rings
    \[ \text{infrastructure, superstructure, extensions}, \]
  – the superstructure is organized as a tree of packages.
    \[ \text{e.g. Actions, Activities, Common Behaviors, Classes, …} \]

• UML is not “object oriented” by heart.
Unified Modeling Language 2.0

Part 2 – Classes and packages

Prof. Dr. Harald Störrle
University of Innsbruck
MGM technology partners

Dr. Alexander Knapp
University of Munich

(c) 2005–2006, Dr. H. Störrle, Dr. A. Knapp
2 – Classes and packages
A first glimpse
2 – Classes and packages

History and predecessors

• Structured analysis and design
  – Entity–Relationship (ER) diagrams (Chen 1976)

• Semantic nets
  – Conceptual structures in AI (Sowa 1984)

• Object-oriented analysis and design
  – Shlaer/Mellor (1988)
  – Coad/Yourdon (1990)
  – Wirfs-Brock/Wilkerson/Wiener (1990)
  – OMT (Rumbaugh 1991)
  – Booch (1991)
  – OOSE (Jacobson 1992)
2 – Classes and packages

Usage scenarios

• Classes and their relationships describe the vocabulary of a system.
  – **Analysis**: Ontology, taxonomy, data dictionary, …
  – **Design**: Static structure, patterns, …
  – **Implementation**: Code containers, database tables, …

• Classes may be used with different meaning in different software development phases.
  – meaning of generalizations varies with meaning of classes

<table>
<thead>
<tr>
<th></th>
<th>Analysis</th>
<th>Design</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concept</td>
<td>√</td>
<td></td>
<td>×</td>
</tr>
<tr>
<td>Type</td>
<td></td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Set of objects</td>
<td>×</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Code</td>
<td>×</td>
<td></td>
<td>√</td>
</tr>
</tbody>
</table>

(c) 2005–2006, Dr. H. Störrle, Dr. A. Knapp
2 – Classes and packages
Analysis class diagram (1)

![UML Diagram](image-url)
2 – Classes and packages

Classes

- **Classes describe a set of instances with common features (and semantics).**
  - Classes induce types (representing a set of values).
  - Classes are namespaces (containing named elements).

- **Structural features (are typed elements)**
  - **properties**
    - commonly known as attributes
    - describe the structure or state of class instances
    - may have multiplicities (e.g. 1, 0..1, 0..*, *, 2..5)
      (default: 0..* = *, but 1 for association ends)

- **Behavioral features (have formal parameters)**
  - **operations**
    - services which may be called
    - need not be backed by a method, but may be implemented otherwise

<table>
<thead>
<tr>
<th>Passenger</th>
</tr>
</thead>
<tbody>
<tr>
<td>name : Name</td>
</tr>
<tr>
<td>creditCard [0..1]</td>
</tr>
<tr>
<td>milesCard [0..1]</td>
</tr>
<tr>
<td>status / currentMiles</td>
</tr>
<tr>
<td>creditMiles(b : Booking)</td>
</tr>
<tr>
<td>consumeMiles(b : Booking)</td>
</tr>
<tr>
<td>cancelMiles()</td>
</tr>
</tbody>
</table>
Associations describe sets of tuples whose values refer to typed instances.

- In particular, structural relationship between classes
- Instances of associations are called links.

Association ends are properties.
- correspond to properties of the opposite class (but default multiplicity is 1)

Association ends may be navigable.
- In contrast to general properties
2 – Classes and packages

Association classes

- Association classes combine classes with associations.
  - not only connect a set of classifiers but also define a set of features that belong to the relationship itself and not to any of the classifiers

- each instance of Booking has one passenger and one flight
- each link of Booking is one instance of Booking
Data types are types whose instances are identified by their value.
- Instances of classes have an identity.
- may show structural and behavioral features

Enumerations are special data types.
- instances defined by enumeration literals
  - denoted by `Enumeration::EnumerationLiteral` or `#EnumerationLiteral`
- may show structural and behavioral features
2 – Classes and packages
Analysis class diagram (2)
2 – Classes and packages

Inheritance (1)

- Generalizations relate specific classes to more general classes.
  - instances of specific class also instances of the general class
  - features of general class also implicitly specified for specific class

- does not imply substitutability (in the sense of Liskov & Wing)
  - must be specified on specific class separately by { substitutable }

- Generalizations also apply to associations.
  - as both are Classifiers
Generalization sets detail the relation between a general and more specific classifiers.

- \{ \text{complete} \}  \quad \text{(opposite: \{\text{incomplete}\})}
  - all instances of general classifier are instances of one of the specific classifiers in the generalization set

- \{ \text{disjoint} \}  \quad \text{(opposite: \{\text{overlapping}\})}
  - no instance of general classifier belongs to more than one specific classifier in the generalization set

- default: \{ \text{disjoint, incomplete} \}

- several generalization sets may be applied to a classifier
  - useful for taxonomies
2 – Classes and packages

Constraints

- Constraints restrict the semantics of model elements.
  - Constraints may apply to one or more elements.
  - No prescribed language.
    - OCL is used in the UML 2.0 specification.
    - Also, natural language may be used.

*User defined constraint:*

\{ mc.number = ma.number \}

*UML predefined constraint:*

(Owner is either a person or a company)
• Packages group elements.
  - Packages provide a namespace for its grouped elements.
  - Elements in a package may be
    • public (+, visible from outside; default)
    • private (-, not visible from outside)
  - Access to public elements by qualified names
    • e.g., Flights::MilesAccount

Notational variants
• Package imports simplify qualified names.

<table>
<thead>
<tr>
<th>Package</th>
<th>Element</th>
<th>Visibility</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>X</td>
<td>private</td>
<td>separate private element import (otherwise public overrides private)</td>
</tr>
<tr>
<td></td>
<td>Q</td>
<td>public</td>
<td>all remaining visible elements of B</td>
</tr>
<tr>
<td>B</td>
<td>X</td>
<td>public</td>
<td>public import</td>
</tr>
<tr>
<td>B</td>
<td>Q</td>
<td>public</td>
<td>default visibility</td>
</tr>
<tr>
<td>B</td>
<td>R</td>
<td>private</td>
<td>private import, renaming</td>
</tr>
</tbody>
</table>
2 – Classes and packages

Packages (3)

• Package mergings combine concepts incrementally.
  – ... but use with care

  – The receiving package defines the increment.

  – The receiving package is simultaneously the resulting package.

  – Merging is achieved by (lengthy) transformation rules (not defined for behavior).

  – Package merging is used extensively in the UML 2.0 specification.
2 – Classes and packages

Metamodel
2 – Classes and packages
Design class diagram
2 – Classes and packages

Features

- ... belong to a namespace (e.g., class or package)

<table>
<thead>
<tr>
<th>Visibility kinds (no default)</th>
<th>visible to elements ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ public</td>
<td>that can access owning namespace (by membership, import, or access)</td>
</tr>
<tr>
<td># protected</td>
<td>with generalization to owning namespace</td>
</tr>
<tr>
<td>~ package</td>
<td>in the same package as the owning namespace</td>
</tr>
<tr>
<td>- private</td>
<td>in owning namespace only</td>
</tr>
</tbody>
</table>

- ... are redefinable (unless decorated by { leaf })
  - in classes that specialize the context class

- ... can be defined on instance or class level

<table>
<thead>
<tr>
<th>TravelStage</th>
<th>isStatic</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxDuration : int = 10</td>
<td>default value</td>
</tr>
</tbody>
</table>

(c) 2005–2006, Dr. H. Störrle, Dr. A. Knapp
2 – Classes and packages

Properties

<table>
<thead>
<tr>
<th>Aggregation kinds (default: none)</th>
<th>{ ordered }</th>
<th>{ unique }</th>
<th>Collection type</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>√</td>
<td>√</td>
<td>OrderedSet</td>
</tr>
<tr>
<td>shared</td>
<td>√</td>
<td>×</td>
<td>Sequence</td>
</tr>
<tr>
<td>composite</td>
<td>×</td>
<td>√</td>
<td>Set (default)</td>
</tr>
<tr>
<td></td>
<td>×</td>
<td>×</td>
<td>Bag</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Collection type</th>
<th>can be computed from other information (default: false)</th>
<th>can only be read, not written (default: false = unrestricted)</th>
<th>union of subset properties (implies derived)</th>
<th>which property this property is a subset of</th>
</tr>
</thead>
</table>
2 – Classes and packages

Behavioral features

• ... are realized by behaviors (e.g., code, state machine).
  − { abstract } (virtual) behavioral features declare no behavior
    • behavior must be provided by specializations
  − Exceptions that may be thrown can be declared
  − Limited concurrency control
    • { active } classes define their own concurrency control

BoardingControl

active class (with own behavior which starts on instance creation)

• in passive classes:

Call concurrency kinds (no default)

| { sequential } | *no concurrency management*
| { guarded }    | *only one execution, other invocations are blocked*
| { concurrent } | *all invocations may proceed concurrently*
2 – Classes and packages
Operations (1)

- An operation specifies the name, return type, formal parameters, and constraints for invoking an associated behavior.
  - «pre» / «post»
    - precondition constrains system state on operation invocation
    - postcondition constrains system state after operation is completed
  - { query }: invocation has no side effects
    - «body»: body condition describes return values
  - { ordered, unique } as for properties, but for return values
  - exceptions that may be thrown can be declared

Parameter direction kinds (default: in)

<table>
<thead>
<tr>
<th></th>
<th>one way from caller</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td><strong>one way from caller</strong></td>
</tr>
<tr>
<td>out</td>
<td><strong>one way from callee</strong></td>
</tr>
<tr>
<td>inout</td>
<td><strong>both ways</strong></td>
</tr>
<tr>
<td>return</td>
<td><strong>return from callee</strong> (at most 1)</td>
</tr>
</tbody>
</table>
2 – Classes and packages
Operations (2)

• **Several semantic variation points for operations**
  – What happens, if a precondition is not satisfied on invocation?
  – When inherited or redefined
    • invariant, covariant, or contravariant specialization?
    • How are preconditions combined?

• **No predefined resolution principle for inherited or redefined operations**
  – “The mechanism by which the behavior to be invoked is determined from an operation and the transmitted argument data is a semantic variation point.”
  – a single–dispatch, object–oriented resolution principle is mentioned explicitly in the UML 2.0 specification

• **Operation invocations may be synchronous or asynchronous.**
2 – Classes and packages
Signals and receptions

• A signal is a specification of type of send request instances communicated between objects.
  – Signals are classifiers, and thus may carry arbitrary data.
  – A signal triggers a reaction in the receiver in an asynchronous way and without a reply (no blocking on sender).

• A reception is a declaration stating that a classifier is prepared to react to the receipt of a signal.
  – Receptions are behavioral features and thus are realized by behavior (e.g., a state machine).
2 – Classes and packages

Interfaces

• Interfaces declare a set of coherent public features and obligations.
  – i.e., specify a contract for implementers (realizers)

```
<<interface>>
TravelHandling
+/ delay : Minutes
+ numOfBag : int = 0
+ delay() [query]
```

features to be offered

Several notations for client/provider relationship

- **client**

```
<<interface>>
B

A
```

- **provider**

```
B

A
```

**lollipop**

```
A
```

**joint**

```
A
```

(c) 2005–2006, Dr. H. Störrle, Dr. A. Knapp
2 – Classes and packages

Templates

subtype polymorphism  vs.  parameteric polymorphism

exposed parameterable elements

Template class (ParameterableElement)

List
  contents : O [n] {sequence}

FlightList

<<bind>> ⟨O -> Flight, n=20⟩

<<bind>> ⟨O -> Passenger, n=300⟩

Bound class (TemplateableElement)

O : Object, n : int
template parameters
template binding

Travel

MeansOfTransport

JourneyByAir
  Plane

TrainJourney
  Train

CoachTour
  Bus

MeansOfTransport

Plane

Train

Bus

T : MeansOfTransport

(c) 2005–2006, Dr. H. Störrle, Dr. A. Knapp
2 – Classes and packages

Object diagram

InstanceSpecification

<table>
<thead>
<tr>
<th>c42 : Connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>from=&quot;MUC&quot;</td>
</tr>
<tr>
<td>to=&quot;AKL&quot;</td>
</tr>
<tr>
<td>dep=07:45</td>
</tr>
<tr>
<td>arr=06:30 (+24)</td>
</tr>
<tr>
<td>status=&quot;planned&quot;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cp1 : ConnPart</th>
</tr>
</thead>
<tbody>
<tr>
<td>from=&quot;MUC&quot;</td>
</tr>
<tr>
<td>to=&quot;LHR&quot;</td>
</tr>
<tr>
<td>flNr=&quot;LH4754&quot;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cp2 : ConnPart</th>
</tr>
</thead>
<tbody>
<tr>
<td>from=&quot;LHR&quot;</td>
</tr>
<tr>
<td>to=&quot;LA&quot;</td>
</tr>
<tr>
<td>flNr=&quot;NZ4550V&quot;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cp3 : ConnPart</th>
</tr>
</thead>
<tbody>
<tr>
<td>from=&quot;LA&quot;</td>
</tr>
<tr>
<td>to=&quot;AKL&quot;</td>
</tr>
<tr>
<td>flNr=&quot;NZ2V&quot;</td>
</tr>
</tbody>
</table>

InstanceValue

<table>
<thead>
<tr>
<th>t42 : Travel</th>
</tr>
</thead>
<tbody>
<tr>
<td>dep=2003-09-23</td>
</tr>
<tr>
<td>arr=2003-09-24</td>
</tr>
<tr>
<td>class=&quot;economy&quot;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ts1 : TravelStage</th>
</tr>
</thead>
<tbody>
<tr>
<td>dep=2003-09-23</td>
</tr>
<tr>
<td>arr=2003-09-23</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ts2 : TravelStage</th>
</tr>
</thead>
<tbody>
<tr>
<td>dep=2003-09-23</td>
</tr>
<tr>
<td>arr=2003-09-23</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ts3 : TravelStage</th>
</tr>
</thead>
<tbody>
<tr>
<td>dep=2003-09-23</td>
</tr>
<tr>
<td>arr=2003-09-24</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>raw4711 : TravelHandling</th>
</tr>
</thead>
<tbody>
<tr>
<td>numOfBag=2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tsh1 : TravelStageHandling</th>
</tr>
</thead>
<tbody>
<tr>
<td>gate=&quot;D12&quot;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tsh2 : TravelStageHandling</th>
</tr>
</thead>
<tbody>
<tr>
<td>gate=&quot;A55&quot;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tsh3 : TravelStageHandling</th>
</tr>
</thead>
<tbody>
<tr>
<td>gate=&quot;C3&quot;</td>
</tr>
</tbody>
</table>

Slot with ValueSpecification

underlining and association end adornments are optional
2 – Classes and packages

Instances specifications

UML metamodel

M2

```
Class type InstanceSpecification

<<instanceOf>>

<<instanceOf>>
```

M1

<table>
<thead>
<tr>
<th>BoardingPass</th>
<th>b : BoardingPass</th>
</tr>
</thead>
<tbody>
<tr>
<td>surname   : String</td>
<td>surname = &quot;stoerrle&quot;</td>
</tr>
<tr>
<td>firstName  : String</td>
<td>firstName = &quot;harald&quot;</td>
</tr>
<tr>
<td>address    : String</td>
<td>address = &quot;mr&quot;</td>
</tr>
<tr>
<td>from       : String</td>
<td>from = &quot;AKL&quot;</td>
</tr>
<tr>
<td>to         : String</td>
<td>to = &quot;MUC&quot;</td>
</tr>
<tr>
<td>carrier    : String</td>
<td>carrier = &quot;ANZ&quot;</td>
</tr>
<tr>
<td>flightNr   : String</td>
<td>flightNr = &quot;NZ02&quot;</td>
</tr>
<tr>
<td>boardingTime: Time</td>
<td>boardingTime = 15:55</td>
</tr>
<tr>
<td>seat       : String</td>
<td>seat = &quot;45C&quot;</td>
</tr>
<tr>
<td>class      : Char</td>
<td>class = 'V'</td>
</tr>
<tr>
<td>numOfBag   : Int</td>
<td>numOfBag = 2</td>
</tr>
</tbody>
</table>

user model
2 – Classes and packages

UML 1.x vs. UML 2.0

• Most changes from UML 1.x to UML 2.0 on the technical side

• Metamodel consolidated in UML 2.0
  – categorization of elements by their properties
    • NamedElement, PackageableElement, RedefineableElement
  – only one level of modeling
    • InstanceSpecification (in contrast to Instance in UML 1.x), ValueSpecification
  – association ends are properties
  – clarification of template mechanism

• Only few new modeling elements in UML 2.0
  – properties (\{ unique, union, … \}) of properties
  – generalization sets (and powertypes)
2 – Classes and packages

Wrap up

• Classifiers and their Relationships describe the vocabulary of a system.

• Classifiers describe a set of instances with common Features.
  – StructuralFeatures (Property’s)
  – BehavioralFeatures (Operations, Receptions)

• Associations describe structural relationships between classes.
  – Association ends are Property’s.

• Generalizations relate specific Classifiers to more general Classifiers.

• Packages group elements
  – and provide a Namespace for grouped elements.

• InstanceSpecifications and links describe system snapshots.
Unified Modeling Language 2.0

Part 2a – Object Constraint Language

Prof. Dr. Harald Störrle  
University of Innsbruck  
MGM technology partners

Dr. Alexander Knapp  
University of Munich

(c) 2005–2006, Dr. H. Störrle, Dr. A. Knapp
2a – Object Constraint Language
A first glimpse

```uml
class Booking
  inv: ma.statusMiles > 10000
  implies status = Status::Albatros

property passenger : Flight
  pre: ma->notEmpty()
  creditMiles(b : Booking)
  consumeMiles(b : Booking)
  cancelMiles()

property flight : Flight
  pre: ma->notEmpty() and
  ma.flightMiles >= b.flight.miles
  consumeMiles(b : Booking)
  post: ma.flightMiles = ma.flightMiles@pre - b.flight.miles

class Passenger
  name : Name
  creditCard[0..1]
  milesCard[0..1]
  status
  / currentFlights : Sequence(Flight)

class Flight
  date : Date
  miles : int

class MilesAccount
  number
  flightMiles
  statusMiles
```

(c) 2005–2006, Dr. H. Störrle, Dr. A. Knapp
2a – Object Constraint Language
History and predecessors

• Predecessors
  – Model-based specification languages, like
    • Z, VDM, and their object-oriented variants; B
  – Algebraic specification languages, like
    • OBJ3, Maude, Larch

• Similar approaches in programming languages
  – ESC, JML

• History
  – developed by IBM as an easy-to-use formal annotation language
  – used in UML metamodel specification since UML 1.1
  – current version: OCL 2.0
    • specification: formal/06-05-01
2a – Object Constraint Language
Usage scenarios

- Constraints on implementations of a model
  - invariants on classes
  - pre-/post-conditions for operations
    - cf. protocol state machines
  - body of operations
  - restrictions on associations, template parameters, ...

- Formalization of side conditions
  - derived attributes

- Guards
  - in state machines, activity diagrams

- Queries
  - query operations

- Model–driven architecture (MDA)/query–view–transformation (QVT)
2a – Object Constraint Language
Language characteristics

• Integration with UML
  – access to classifiers, attributes, states, ...
  – navigation through attributes, associations, ...
  – limited reflective capabilities
  – model extensions by derived attributes

• Side-effect free
  – not an action language
  – only possibly describing effects

• Statically typed
  – inherits and extends type hierarchy from UML model

• Abstract and concrete syntax
  – precise definition new in OCL 2.0
2a – Object Constraint Language

Simple types

- **Predefined primitive types**
  - Boolean  true, false
  - Integer   -17, 0, 3
  - Real      -17.89, 0.0, 3.14
  - String    “Hello”

- **Types induced by UML model**
  - Classifier types, like
    - Passenger no denotation of objects, only in context
  - Enumeration types, like
    - Status    Status::Albatros, #Albatros
  - Model element types
    - OclModelElement, OclType, OclState

- **Additional types**
  - OclInvalid invalid (OclUndefined)
  - OclVoid    null
  - OclAny     top type of primitives and classifiers
2a – Object Constraint Language
Parameterized types

- **Collection types**
  - `Set(T)` — sets
  - `OrderedSet(T)` — like `Sequence` without duplicates
  - `Bag(T)` — multi-sets
  - `Sequence(T)` — lists
  - `Collection(T)` — abstract

- **Tuple types (records)**
  - `Tuple(a_1 : T_1, \ldots, a_n : T_n)`

- **Message type**
  - `OclMessage` — for operations and signals

**Examples**

- `Set{Set{ 1 }, Set{ 2, 3 }} : Set(Set(Integer))`
- `Bag{1, 2.0, 2, 3.0, 3.0, 3} : Bag(Real)`
- `Tuple{x = 5, y = false} : Tuple(x : Integer, y : Boolean)`
2a – Object Constraint Language

Type hierarchy

- Type conformance (reflexive, transitive relation \( \leq \))
  - \( \text{OclVoid}, \text{OclInvalid} \leq T \) for all types \( T \)
  - Integer \( \leq \) Real
  - \( T \leq T' \Rightarrow C(T) \leq C(T') \) for collection type \( C \)
  - \( C(T) \leq \text{Collection}(T) \) for collection type \( C \)
  - generalization hierarchy from UML model
    - \( B \leq \text{OclAny} \) for all primitives and classifiers \( B \)

Counterexample
  - \( \neg (\text{Set}(\text{OclAny}) \leq \text{OclAny}) \)

- Casting
  - \( v.\text{oclAsType}(T) \) if \( v : T' \) and \( (T \leq T' \) or \( T' \leq T) \)
  - upcast necessary for accessing overridden properties
    - but are (still) forbidden in the specification
2a – Object Constraint Language Expressions

- Local variable bindings
  ```
  let x = 1 in x+2
  ```

- Iteration
  ```
c->iterate(i : T; a : T' = e' | e)
  ```

Example:
- Many operations on collections are reduced to `iterate`

Example:
- `Set{1, 2}->iterate(i : Integer; a : Integer = 0 | a+i) = 3`
2a – Object Constraint Language

Expressions: Standard library (1)

- **Operations on primitive types** (written: \( v\ .\ op \ (\ldots) \))
  - Operations without () are mixfix
  
<table>
<thead>
<tr>
<th>Type</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>OclAny</td>
<td>=, &lt;&gt;, oclIsTypeOf( T ), oclIsKindOf( T ), ...</td>
</tr>
<tr>
<td>Boolean</td>
<td>and, or, xor, implies, not</td>
</tr>
<tr>
<td>Integer</td>
<td>+, −, *, /, div( i ), mod( i ), ...</td>
</tr>
<tr>
<td>Real</td>
<td>+, −, *, /, floor(), round(), ...</td>
</tr>
<tr>
<td>String</td>
<td>size(), concat( s ), substring( l, u ), ...</td>
</tr>
</tbody>
</table>

- **Operations on collection types** (written: \( v\rightarrow op \ (\ldots) \))
  
<table>
<thead>
<tr>
<th>Type</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collection</td>
<td>size(), includes( v ), isEmpty()</td>
</tr>
<tr>
<td>Set</td>
<td>union( s ), including( v ), flatten()</td>
</tr>
<tr>
<td>OrderedSet</td>
<td>append( s ), first(), at( i )</td>
</tr>
<tr>
<td>Bag</td>
<td>union( b ), including( v ), flatten()</td>
</tr>
<tr>
<td>Sequence</td>
<td>append( s ), first(), at( i ), asOrderedSet()</td>
</tr>
</tbody>
</table>
2a – Object Constraint Language
Expressions: Standard library (2)

- **Finite quantification**
  - `c->forall(i : T | e)` = `c->iterate(i : T; a : Boolean = true | a and e)`
  - `c->exists(i : T | e)` = `c->iterate(i : T; a : Boolean = false | a or e)`

- **Selecting values**
  - `c->any(i : T | e)` some element of `c` satisfying `e`
  - `c->select(i : T | e)` all elements of `c` satisfying `e`

- **Collecting values**
  - `c->collect(i : T | e)` collection of elements with `e` applied to each element of `c`
  - `c.p` collection of elements `v.p` for each `v` in `c` (short-hand for `collect`)

<table>
<thead>
<tr>
<th><code>C.allInstances()</code></th>
<th>all current instances of classifier <code>C</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>o.oclIsInState(s)</code></td>
<td>is <code>o</code> currently in state machine state <code>s</code>?</td>
</tr>
<tr>
<td><code>v.oclIsUndefined()</code></td>
<td>is value <code>v</code> undefined (null) or invalid?</td>
</tr>
<tr>
<td><code>v.oclIsInvalid()</code></td>
<td>is value <code>v</code> invalid?</td>
</tr>
</tbody>
</table>
2a – Object Constraint Language Evaluation

- Strict evaluation with some exceptions
  - \((\text{if } (1/0 = 0) \text{ then } 0.0 \text{ else } 0.0 \text{ endif}).oclIsInvalid() = true\)
  - \((1/0).oclIsInvalid() = true\)

- Short-cut evaluation for and, or, implies
  - \((1/0 = 0.0) \text{ and } \text{false} = \text{false}\)
  - \(\text{true or } (1/0 = 0.0) = \text{true}\)
  - \(\text{false implies } (1/0 = 0.0) = \text{true}\)
  - \((1/0 = 0.0) \text{ implies } \text{true} = \text{true}\)
  - \(\text{if } (0 = 0) \text{ then } 0.0 \text{ else } 1/0 \text{ endif} = 0.0\)

- In general, OCL expressions are evaluated over a system state.

\[\begin{align*}
\text{c42:Connection} & \text{ conn } \text{t42:Travel} \\
\text{from} = \text{MUC} & \text{dep} = \text{2003-09-23} \\
\text{to} = \text{AKL} & \text{arr} = \text{2003-09-24} \\
\text{dep} = \text{07:45} & \text{class} = \text{economy} \\
\text{arr} = \text{06:30 (+24)} & \text{numOfBag} = 2 \\
\text{status} = \text{planned} & \\
\end{align*}\]
2a – Object Constraint Language Connection to UML

- Import of classifiers and enumerations as types
- Properties accessible in OCL
  - Attributes
    - \( p \text{.} \text{milesCard} \) (with \( p : \text{Passenger} \))
  - Association ends
    - \( p \text{.} \text{flight}, p \text{.} \text{booking}, p \text{.} \text{booking} [\text{flight}] \)
  - \{ query \} operations

- Representation of multiplicities

<table>
<thead>
<tr>
<th>( a[1] : T )</th>
<th>( a : T )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a[0..1] : T )</td>
<td>( a : \text{Set}(T) ) or ( T )</td>
</tr>
<tr>
<td>( a[m..n] : T )</td>
<td>( a : \text{Set}(T) )</td>
</tr>
<tr>
<td>( a[*] : T ) { unordered }</td>
<td>( a : \text{Set}(T) )</td>
</tr>
<tr>
<td>( a[*] : T ) { ordered }</td>
<td>( a : \text{OrderedSet}(T) )</td>
</tr>
<tr>
<td>( a[*] : T ) { bag }</td>
<td>( a : \text{Bag}(T) )</td>
</tr>
</tbody>
</table>
2a – Object Constraint Language

Invariants

context Passenger
inv: ma.statusMiles > 10000 implies status = Status::Albatros

Notational variants

context Passenger
inv statusLimit: self.ma.statusMiles > 10000 implies self.status = Status::Albatros

context p : Passenger
inv statusLimit: p.ma.statusMiles > 10000 implies p.status = Status::Albatros
2a – Object Constraint Language
Semantics of invariants

- Restriction of valid states of classifier instances
  - when observed from outside

- Invariants (as all constraints) are inherited via generalizations
  - but how they are combined is not predefined

- One possibility: Combination of several invariants by conjunction

\[
\begin{align*}
\text{context} & \quad C \\
\text{inv:} & \quad I_1 \\
\text{context} & \quad C \\
\text{inv:} & \quad I_2 \\
\rightarrow & \quad \text{context} \\
& \quad C \\
\text{inv:} & \quad I_1 \ \text{and} \ I_2
\end{align*}
\]
2a – Object Constraint Language

Pre-/post–conditions

- In UML models, pre- and post-conditions are defined separately
  - not necessarily as pairs
  - «precondition» and «postcondition» as constraint stereotypes

```uml
class Passenger

context Passenger::consumeMiles(b : Booking) : Boolean
pre: ma->notEmpty() and
   ma.flightMiles >= b.flight.miles

post: ma.flightMiles = ma.flightMiles@pre-b.flight.miles and
   result = true
```

- Some constructs only available in post-conditions
  - values at pre-condition time
  - result of operation call
  - whether an object has been newly created
  - messages sent
2a – Object Constraint Language
Semantics of pre-/post-conditions

• **Standard interpretation**
  - A pre-/post-condition pair \((P, Q)\) defines a relation \(R\) on system states such that \((\sigma, \sigma') \in R\), if \(\sigma \models P\) and \((\sigma, \sigma') \models Q\).
    - system state \(\sigma\) on operation invocation
    - system state \(\sigma'\) on operation termination (\(Q\) may refer to \(\sigma\) by \(\text{@pre}\)).
  - Thus \((P, Q)\) equivalent to \((true, P\text{@pre} \text{ and } Q)\).

• **Meyer’s contract view**
  - A pre-/post-condition pair \((P, Q)\) induces benefits and obligations.
  - benefits and obligations differ for implementer and user

<table>
<thead>
<tr>
<th>user</th>
<th>obligation</th>
<th>benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>satisfy (P)</td>
<td>(Q) established</td>
<td></td>
</tr>
<tr>
<td>implementer</td>
<td>if (P) satisfied, establish (Q)</td>
<td>(P) established</td>
</tr>
</tbody>
</table>
2a – Object Constraint Language
Combining pre-/post-conditions

- **Standard interpretation**
  - joining pre- and post-conditions conjunctively
    
    \[
    \text{context } C::\text{op ()} \\
    \text{pre: } P_1 \quad \text{post: } Q_1 \\
    \text{pre: } P_2 \quad \text{post: } Q_2
    \]

- **Alternative interpretation**
  - case distinction (like in protocol state machines)
  - only useful for pre-/post-condition pairs
    
    \[
    \text{context } C::\text{op ()} \\
    \text{pre: } P_1 \quad \text{post: } Q_1 \\
    \text{pre: } P_2 \quad \text{post: } Q_2
    \]
context Subject::hasChanged()  
post: observer^update(self) 
    in calls on hasChanged, 
    some update message with argument 
    self will have been sent to observer 

context Subject::hasChanged()  
post: observer^update(? : Subject) 
    the actual argument 
    does not matter 

context Subject::hasChanged()  
post: let messages : Set(OclMessage) =  
    observer^update(? : Subject) 
    all those 
    messages 
    in messages->notEmpty() and 
    messages->forall(m | 
    result of message call 
    m.result().oclIsUndefined() and 
    whether it has finished 
    m.hasReturned() and 
    its actual parameter value 
    m.subject = self)
2a – Object Constraint Language
Initial values and derived properties

- **Initial values**
  - fix the initial value of a property of a classifier

```
package Booking
  context Passenger::status
  init: Status::Swallow
endpackage
```

- **{ derived } properties**
  - define how the value of a property is derived from other information

```
context Passenger::currentFlights : Sequence(Flight)
derive: self->collect(booking)
  ->select(date = today()).flight->asSequence()
```
2a – Object Constraint Language
Query bodies and model features

• Bodies of { query } operations
  - define the value returned by a query operation
  - can be combined with a precondition
  
  ```
  context TravelHandling::delay() : Minutes
  body: tsh.delay->sum()
  ```

• Definition of additional model features
  - defined for the context classifier
  
  ```
  context TravelStageHandling
  def: isEarly() : Boolean = self.delay < 0

  context TravelHandling
  def: someEarly() : Boolean = tsh->exists(isEarly())
  ```
2a – Object Constraint Language
UML/OCL 1.x vs. UML/OCL 2.0

• Improvements in OCL 2.0
  – Model extensions by definitions
  – Explicit flattening of collections
  – Clarification of type hierarchy
  – Precise abstract and concrete syntax
  – Formal semantics
    • but only as a non-normative appendix

• New features in OCL 2.0
  – Specification of initial values, derived attributes
  – Specification of messages

• (still) Open problems
  – semantics of definitions
    • inheritance, recursion
  – semantics of pre-/post-conditions
2a – Object Constraint Language

Wrap up

• Formal language for specifying
  - invariants
  - pre-/post-conditions
  - query operation bodies
  - initial values
  - derived attributes
  - modeling attributes and operations

\[
\text{context } C \text{ inv: } I
\]
\[
\text{context } C::\text{op()} : T \\
\text{pre: } P \text{ post: } Q
\]
\[
\text{context } C::\text{op()} : T \text{ body: } e
\]
\[
\text{context } C::p : T \text{ init: } e
\]
\[
\text{context } C::p : T \text{ derive: } e
\]
\[
\text{context } C \text{ def: } p : T = e
\]

• Side-effect free
• Typed language

• OCL specifications provide
  - verification conditions
  - assertions for implementations
Unified Modeling Language 2.0

Part 2b – Profiles

Dr. Harald Störrle
University of Innsbruck
MGM technology partners

Dr. Alexander Knapp
University of Munich

(c) 2005–2006, Dr. H. Störrle, Dr. A. Knapp
2b – Profiles
A first glimpse
2b – Profiles
Usage scenarios

• Metamodel customization for
  – adapting terminology to a specific platform or domain
  – adding (visual) notation
  – adding and specializing semantics
  – adding constraints
  – transformation information

• Profiling
  – packaging domain–specific extensions
  – “domain–specific language” engineering
2b – Profiles
Stereotypes (1)

- Stereotypes define how an existing (UML) metaclass may be extended.

- Stereotypes may be applied textually or graphically.

- The UML specification does not tell how to define a visual stereotype.
- Visual stereotypes may replace original notation.
  - But the element name should appear below the icon...
• Stereotypes may define meta-properties.
  – commonly known as “tagged values”
• Stereotypes can be defined to be required.
  – Every instance of the extended metaclass has to be extended.
  – If a required extension is clear from the context it need not be visualized.
• Profiles package extensions.
2b – Profiles

Metamodel

- Based on infrastructure library constructs
  - Class, Association, Property, Package, PackageImport
2b – Profiles
Metamodeling with Profiles

• Profile extension mechanism imposes restrictions on how the UML metamodel can be modified.
  – UML metamodel considered as “read only”.
  – No intermediate metaclasses, no meta-associations
  – “As part of a profile, it is not possible to have an association between two stereotypes or between a stereotype and metaclass.”

• Stereotypes metaclasses below UML metaclasses.
• How to introduce meta-associations between stereotypes?
  1. Add constraints specializing some existing associations
  2. Extend metaclass Dependency by a stereotype and define specific constraint on this stereotype

• Access to stereotypes in OCL via \texttt{v.stereotype}
# 2b – Profiles

## UML 1.x vs. UML 2.0

<table>
<thead>
<tr>
<th>UML 1.x</th>
<th>UML 2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>String-based extension mechanism</strong></td>
<td><strong>Stereotypes are metaclasses</strong></td>
</tr>
<tr>
<td>– Stereotypes</td>
<td>– Tagged values replaced by meta-properties</td>
</tr>
<tr>
<td>– Tagged values</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Required extensions</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Packaging of extensions into profiles</strong></td>
</tr>
</tbody>
</table>

(c) 2005–2006, Dr. H. Störrle, Dr. A. Knapp
2b – Profiles

Wrap up

- Metamodel extensions
  - with stereotypes and meta-properties
  - for restricting metamodel semantics
  - for extending notation

- Packaging of extensions into profiles
  - for declaring applicable extensions
  - “domain-specific language” engineering
Unified Modeling Language 2.0

Part 2c – Systems Modeling Language

(SysML)

Prof. Dr. Harald Störrle
University of Innsbruck
MGM technology partners

Dr. Alexander Knapp
University of Munich

(c) 2005–2006, Dr. H. Störrle, Dr. A. Knapp
Nowadays very much talked about: Systems Modeling Language (SysML).
2c – SysML
SysML vs. UML

- Protracted struggle between two competing proposals fueled by massive commercial interests.
- Apart from mere customization to match systems engineering standards and terminology, it also introduces some physical aspects:
  - continuous flows,
  - handling of physical items.

outside of what can be described with UML means?

official OMG diagram
2c – SysML
Diagram types of SysML

significant new aspects:
- item flow
- continuous variables
- activation disabling
- control operators

class diagram

assembly diagram

official OMG diagram
2c – SysML

Wrap up

• Tries to extend UML towards systems engineering, i.e. physical/continuous systems.

• Probably the most talked about and largest UML profile.

• After a long and fierce debate, now finally OMG approved.

• Semantics completely unclear, seems to go even more into the direction of Petri-nets.
Unified Modeling Language 2.0

Part 3 – Use Cases

Prof. Dr. Harald Störrle
University of Innsbruck
MGM technology partners

Dr. Alexander Knapp
University of Munich

(c) 2005–2006, Dr. H. Störrle, Dr. A. Knapp
3 – Use Cases
A first glimpse

- Displayed aspects
  - System boundary and context of system
  - Users and neighbor systems
  - Functionalities
  - Relationships between functionalities (calling/dependency, taxonomy)
  - Functional requirements
  - Some non-functional (“quality”) requirements as comments/annotations
3 – Use Cases

History and predecessors

• 1970’s
  – Structured methods (SADT etc.) use top-level DFD as context diagram
  – Structured methods use function trees

• 1980’s
  – Jacobson (Objectory) introduces the concept of use case as an aid for communicating with domain experts

• 1997
  – UML 1.3 encompasses Use Cases
3 – Use Cases

Usage scenarios

- **Use case inventory/ domain architecture**
  - complete catalog of all subdomains and (groups of) business processes and business functions
  - overview of system’s (domain) capabilities

- **“Classical” use cases**
  - illustrate context of individual functionality
  - useful in design/documentation of business processes (i.e. analysis phase and reengineering)

- **Use Case / Test case table**
  - schematic detail description of business process/function/test case

- **Function tree**
  - describe functional decomposition of system behavior
  - useful for non–OO construction and for re–architecting pre–OO systems
3 – Use Cases

Types of use cases

• The UML provides only the concept of use case. In many applications, however, there are two fundamentally different kinds of use cases:

  – business processes (“processes”)
    • white box, large scale, long running (suspendable), customized processes
    • either dialogue or batch processes
    • directly support the business or domain of the system, create or destroy value
    • are subject to rearrangement when business changes
    • may contain some manual steps and business functions

  – business functions (“services”)
    • black box, small(er) scale, short(er) running, atomic, reusable function
    • small recurring functionality, plausibility, user dialogue, interface call, . . .
    • implements stable functionality likely not to be affected by business changes
    • is executed fully automatic
3 – Use Cases

Main concepts (concrete syntax)

- **Class** (also possible: **Component**)
- **Actor**
- **UseCase**
  - extends (is a **Dependency**)
  - includes (is a **Dependency**)
  - **Association**
3 – Use Cases

Inclusion & extension

• Inclusion
  - plain old call
  - directed from caller to callee
  - may occur once or many times

• Extension
  - covers variant or exceptional behavior
  - relationship is directed from exception to standard case
  - may or may not occur
  - occurs at most once
3 – Use Cases

Extension points

- An extension occurs at a (named) ExtensionPoint, when a specific condition is satisfied.

- In a way, ExtensionPoints are similar to *user exits* or *hooks*.

- In real world systems, there are *many* ExtensionPoints, most of which are poorly documented.
3 – Use Cases

Any level of abstraction is ok

- A use case represents an individual functionality of a system.

- Systems exist on every level of granularity.

- Thus, use cases may be used for functionality of any granularity:
  - from high level business processes,
  - via (web) services,
  - to individual methods or functions.
3 – Use Cases

Emulating function trees

- Structured methods relied on functional decomposition.

- Although this is not state of the art these days, and UseCases have been introduced in an attempt to get away from it:
  - many systems out there are constructed using these principles,
  - many people out there have this mindset.

- For e.g. reengineering purposes, it is frequently helpful to be able to represent function trees.

- This can be done using UseCases and Includes–Relationships.
3 – Use Cases

Generalization

- As for all Classifiers, UseCases may be arranged in taxonomic hierarchies.

- This is particularly useful for catalogues of functionalities.

- From methodological point of view, abstract use cases are similar to functional subsystems.
3 – Use Cases

Semantics

• Use cases have no semantics in UML.

• There are many consistency conditions in conjunction with other models, but that’s methodology, and beyond the scope of this tutorial.
3 – Use Cases

UML 1.x vs. UML 2.0

- no changes conceptually
- slight adaptations in the metamodel
3 – Use Cases

Wrap up

• Use cases may be used to represent a high-level view of functionality, as in
  – functionality overview / domain architecture
  – detail description of context of individual use case
  – function tree (particularly for reengineering and documentation purposes)

• The UML still does not come with a (textual) schema for describing use cases.

• Basically, use cases in UML 2.0 are the same as in UML 1.x.
Unified Modeling Language 2.0

Part 4 – Architecture

Prof. Dr. Harald Störrle
University of Innsbruck
MGM technology partners

Dr. Alexander Knapp
University of Munich

(c) 2005–2006, Dr. H. Störrle, Dr. A. Knapp
4 – Architecture
A first glimpse

Context & domain architecture

Composite structure ("assembly") diagrams

Deployment

Collaboration
4 – Architecture
History and Predecessors

• Context and domain architecture diagram
  – 1970’s: SADT et al. use top level DFD as context diagram
  – 1988: Shlaer/Mellor introduce domain chart

• Part/port/connector-concepts, composite structure (“assembly”) diagram
  – 1976: SDL (block/gate/channel)
  – 1978: SARA (module/socket/interconnection)
  – 1993: RAPIDE (module/type/binding)
  – 1994: ROOM (actor/port/connector)
  – 1999: UML/RT,… (capsule/port/connector)
  – 2000: UML 1.3 (subsystem/-/-)

• collaboration
  – 1997: Catalysis (D’Souza, Wills)

• computing system structure diagram (“deployment”)
  – traditional
4 – Architecture
Usage Scenarios / Architectural views

• **Context diagram**
  - define the system’s boundaries in terms of its users and neighbor systems
  - define names/abbreviations for systems and neighbor systems

• **Domain architecture**
  - provide overview of high-level domain components
  - define names/abbreviations for subsystems

• **Composite structure diagram (system assembly diagram)**
  - define ports (“system interfaces”) with names and abbreviations
  - define connections between interfaces

• **Composite structure diagram (class assembly diagram)**
  - as above on fine level of granularity
  - define (programming language) interfaces for ports, too

• **Collaboration**
  - document design decisions (“patterns”) on any level of granularity

• **System structure diagram**
  - physical nodes and connections between them
better name: assembly diagrams
4 – Architecture
Usage: Stepwise refinement
4 – Architecture
Usage: Quantity structures

- Quantity structures are indispensable for the dimensioning of large systems:
  - numbers of (concurrent) users,
  - amount of data traffic on a given interface,
  - availability, MTBF,...
  - number of (collaborating) systems or components (dynamic architectures).

- Quantity structures are not covered directly in UML so we need to use comments (but that’s no problem).
4 – Architecture

Usage: Plumbing components together

- Connecting Ports amounts to delegate/call–Dependencies.

- Using the joint–notation reveals details about the number and direction of connections.

- From left to right:
  - two connectors, one in each direction
  - one connector with direction
  - and one connector without direction.
Components

- UML 1.x components were just binaries. In UML 2.0, components are defined much more comprehensively.
  
  "A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment.

- A component defines its behavior in terms of provided and required interfaces. As such, a component serves as a type, whose conformance is defined by these provided and required interfaces (encompassing both their static as well as dynamic semantics). One component may therefore be substituted by another only if the two are type conformant. [...]

- A component is modeled throughout the development life cycle [...]."
4 – Architecture
Metamodel: Parts and ports

- Classifier
- StructuredClassifier
- EncapsulatedClassifier
- Component
- Class
- Property
- ConnectorEnd
- Port
- Interface

Dashed outlines: defined in another package
4 – Architecture Collaboration

- The purpose of collaborations is to abstractly describe a form of linkage without being specific.

- Declared as the way to describe design and analysis patterns.

- Might help in early stages of architectural design.

- Could also be used to describe global constraints.

- May be nested and composed.

- Methodologically, Collaborations are the structural equivalent to UseCases.
4 – Architecture

System structure

CommunicationPath is a kind of Association

Node

Boarding Machine

Gate Terminal

Check-In Machine

Check-In Terminal

0..50

2..80

0..20

10..150

max. throughput:
20 Passengers/minute

Airport Mainframe

Albatros Data Center

1..3

120

for every 1000 Passengers/day 2 redundant ADSL lines with 3Mbit/s each for backup

99.99% availability
(i.e. less than 1h/year of downtime)

Comment for quantitative information
4 – Architecture Stereotyping
4 – Architecture

Deployment

- A Deployment is a mapping of artifacts to system nodes.
  - Artifacts may include
    - binaries
    - component instances
  - System nodes may include
    - hardware (Device)
    - software (ExecutionEnvironment)
- Formally, a deployment is a Deployment Relationship.
- It may be presented either as placing the deployed items or their names on the deployment target.
4 – Architecture
Metamodel: Deployment

dashed outlines:
defined in another package
4 – Architecture Semantics

• Mappings from assemblies to Architecture Description Languages (ADLs) or SDL should be possible. Is it much use? Can there be a uniform semantics for all kinds of ADLs?

• Collaborations seem to have no formal semantics.

• System structures may be mapped to quantitative models for analytical purposes.

• Deployments might be turned into deployment descriptors of application servers.
4 – Architecture

UML 1.x vs. UML 2.0

**UML 1.x**

- “system boundary”
- components are binaries
- patterns as templates

**UML 2.0**

- Parts/Ports
- artifacts
- components are life cycle units
- patterns (=collaborations) are now first class citizens
4 – Architecture

Wrap up

• Popular concepts of architectural modeling ("capsule"/"actor", "port") have finally been included into UML. The metamodeling is a little dodgy, though.

• Deployments, artifacts and related concepts have been extended, and they are now first-class citizens.

• Components have finally got a decent definition as life cycle units, artifacts and deployments are now first-class citizens.
Unified Modeling Language 2.0

Part 5 – Activities

Prof. Dr. Harald Störrle
University of Innsbruck
MGM technology partners

Dr. Alexander Knapp
University of Munich

(c) 2005–2006, Dr. H. Störrle, Dr. A. Knapp
5 – Activities
A first glimpse

• Activity diagrams present all kinds of control flow and data flow.
• They are kind of dual to state machines: focus is on actions rather than states.
• The UML 1.x notation has been kept (with a different meaning), and much extended.
5 – Activities
History and predecessors

• 1962
  – Petri-nets

• 1969
  – Flow charts (IBM, ISO)

• 1970‘s
  – Nassi–Shneiderman–diagrams

• 1980‘s
  – Structured Methods (SADT etc.) introduce data flow diagrams
  – Methodologies like IDEF are based on these notations

• 1990‘s
  – event process chains (particularly in BPR & SAP R/3 context)
5 – Activities
Usage scenarios

- Activity diagrams have applications throughout the whole software life cycle for many purposes.

- Analysis
  - design or document processes in the application domain (business processes)

- Design
  - design or document processes as compositions of preexisting elements like manual tasks or automated jobs

- Implementation
  - document existing programs (i.e. functions, services, …)
  - design algorithmic processes with an intention of turning them into implementation language code
5 – Activities
Main concepts

Partition
InitialState
ForkNode
DecisionNode
MergeNode
JoinNode
FinalState

ObjectNode
ActivityNode
ObjectFlow
ActivityEdge

(c) 2005–2006, Dr. H. Störrle, Dr. A. Knapp
5 – Activities

Main concepts

Partitions may be represented explicitly

refined Activity

stereotyped ObjectNode

FlowFinal
5 – Activities

Pins

- Data flow may be represented
  - explicitly,
  - by dataflow nodes attached to control flow,
  - by “Pins” on Activities, or
  - as combinations.
5 – Activities

Activity parameters

- Pins act as parameters for Activities.

![Diagram of Activity parameters]

ActivityParameter
5 – Activities

Pin types

- a) streaming
- b) streaming
- c) exception
- d) unidirectional
- e) collection data

ParameterSet may be used to define applicable sets of parameters

- Parameter sets
  - a) \{x, y\}
  - b) \{x\}, \{y\}
  - c) \{x\}, \{x, y\}
5 – Activities

Data flow details

• Data flow defines the transport of data items between buffers by activities.

• Buffers may have capacities and orderings.

• Apart form the transport as such, data flow may also define
  – selection of a particular data item, and
  – transformation of data items.

• It is often useful to denote the state of a data item in a buffer.
5 – Activities

Streaming

- Streaming means that data is processed pipeline-style.

- Streaming–like behavior was not expressible in UML 1.x.

- Streaming is expressed by
  - solid black pins
  - explicit annotation at pins
  - black arrowhead arcs, or
  - stream mode at expansion regions.
5 – Activities

Exceptions

- ExceptionEdge
- unhandled Exception
- handler Activity

(c) 2005–2006, Dr. H. Störrle, Dr. A. Knapp
5 – Activities
Raising exceptions

InterruptibleActivityRegion

AD check luggage item

- weigh luggage item
- luggage item is removed

- :LuggageError
  kind="item too heavy"

- compute total luggage weight

  - weight>15 kg
    - :LuggageError
      kind="item is removed"

  - weight>20 kg and status < "Albatros"

  - weight>15 kg and status < "Adler"
    - :LuggageError
      kind="item too heavy"

  - weight>12 kg and no bonus miles card
5 – Activities

Expansion regions for mass data processing

- Activities are often used to specify processing of mass data (batch runs) rather than individual items.

- UML offers ExpansionRegions to support this usage scenario.

- An expansion region declares a portion of an activity as applicable to a whole bunch of items.
5 – Activities
Expansion Regions

- An expansion region may be processed in one of three modes
  - iterative,
  - concurrent,
  - stream.
5 – Activities
Structured nodes

• Structured nodes for
  – sequence,
  – loop,
  – conditional.

• No/insufficient syntax (let alone semantics) defined by standard.

• We’re probably best of with a Nassi–Shneiderman–like notation.
5 – Activities

Metamodel

Activity

ActivityEdge

ObjectFlow

guard: Guard

transformation: Behavior

selection: Behavior

Incoming * \rightarrow 0..1 \rightarrow Outgoing

target 1 \rightarrow 1 \rightarrow source

ActivityNode

ObjectName

inState: State

upperBound: ValSpec

ExecutableNode

Action

StructuredActivityNode

ConditionalNode

LoopNode

ExpansionRegion

SequentialNode

ControlNode

ForkNode

JoinNode

MergeNode

DecisionNode

InitialNode

FinalNode

ActivityEdge

ControlFlow

ObjectFlow

CentralBufferNode

DataStoresNode

Pin

OutputPin

InputPin

ActivityParameterNode

(c) 2005–2006, Dr. H. Störrle, Dr. A. Knapp
5 – Activities

Semantics

• The standard declares “Petri–like semantics”. The naive approach is
  – intuitive for simple control and data flow
  – reasonable for structured nodes
  – technically difficult for exceptions
  – a little awkward for streams and ExpansionRegions.

• There are a number of semantical problems, though, and integrating the bits and pieces is a challenge.

• Still, it is the most convincing approach so far.
5 – Activities

Semantics: Petri-net vs. CCS

Activity = a.P

P = receive_order.Q

Q = R + S

S = fill_order.T

R = t.T

T = U\{join,join\}

U = produce_goods.V

V = ship_goods.W

X = send_invoice.Y

Y = receive_payment.Y

Y* = join

Y' = join.Z

Z = close_order.0

Spot the error!
5 - Activities
Problem 1: Scope of exceptions
5 - Activities

Problem 2: Accidental synchronization of streams
Problem 3: Traverse-to-completion

• Transforming an Activity into a Petri-net following the naive approach results in artificial places that have no direct equivalent in the underlying Activity.

• The UML, however, disallows buffering in control nodes.
5 – Activities
Semantics

- The standard declares that activities have a “Petri–like” semantics, but lacks a formal definition of what that means.

- A straight–forward approach of mapping activities to Petri–nets soon runs into a semantic quagmire.

- Other algorithmic target languages (e.g. BPEL or Workflow Execution Languages), and other formalisms (e.g. CCS) would encounter the same problems, plus their own.

- Abstract descriptions using special–purpose logics are only at the beginning.

- Many open questions that will trouble us for some time to come.
# 5 – Activities

## UML 1.x vs. UML 2.0

<table>
<thead>
<tr>
<th>UML 1.x</th>
<th>UML 2.0</th>
</tr>
</thead>
</table>
| • ActivityGraph subclass of StateMachine  
  • thus implicit rtc-semantics | • Activity on same level as StateMachine  
  • new “Petri–like” semantics |
| | • Many new concepts |
| |  – Exceptions  
  • InterruptibleActivityRegion  
  • ExceptionEdge, ProtectedNode  
  – StructuredNodes  
  – FlowFinal  
  – Streaming  
  – Collection data  
  – ActivityParameters |
| | • Many new notations  
  – Pins, “attached” dataflow notation, … |
5 – Activities

Wrap up

- Presents control flow and data flow for analysis, design, and implementation level models.

- Not a special kind of StateMachine any more.

- Petri-net inspired semantics, though currently not entirely clear.

- Many new concepts and notations, including
  - Exception handling
  - Data streaming
  - Collection data handling
  - Structured nodes (loops, expansion regions)
  - Pin-notation for dataflow.

- Overall: Activity diagrams are now the algorithmic description language – not only within the UML.
Unified Modeling Language 2.0

Part 6 – State machines

Dr. Harald Störrle
University of Innsbruck
MGM technology partners

Dr. Alexander Knapp
University of Munich

(c) 2005–2006, Dr. H. Störrle, Dr. A. Knapp
6 – State machines
History and predecessors

• 1950’s: Finite State Machines
  – Huffmann, Mealy, Moore

• 1987: Harel Statecharts
  – conditions
  – hierarchical (and/or) states
  – history states

• 1990’s: Objectcharts
  – adaptation to object orientation

• 1994: ROOM Charts
  – run–to–completion (RTC) step
6 – State machines

Usage scenarios

• Object life cycle
  – Behavior of objects according to business rules
  – in particular for active classes

• Use case life cycle
  – Integration of use case scenarios
  – Alternative: activity diagrams

• Control automata
  – Embedded systems

• Protocol specification
  – Communication interfaces
6 – State machines
States and transitions

- State machines model behavior
  - using states interconnected ...
  - with transitions triggered ...
  - by event occurrences.

States and transitions include:
- States (e.g., Reserved, Booked, StartedOff)
- Transitions (e.g., change(event), pay(), cancel())
- Guards (e.g., guard(constraint))
- Effects (e.g., effect(callaction))
6 – State machines
Relation to class diagrams

- State machines are defined in the context of a BehavioredClassifier.

- **Context** defines which
  - events can occur
  - features are available

```
<table>
<thead>
<tr>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Booking</td>
</tr>
<tr>
<td>kind</td>
</tr>
<tr>
<td>pay()</td>
</tr>
<tr>
<td>cancel()</td>
</tr>
<tr>
<td>startOff()</td>
</tr>
<tr>
<td>change()</td>
</tr>
<tr>
<td>handle()</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
</tr>
<tr>
<td>Booked</td>
</tr>
<tr>
<td>StartedOff</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>CallAction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Passenger</td>
</tr>
<tr>
<td>name: Name</td>
</tr>
<tr>
<td>creditMiles(b: Booking)</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Booking</td>
</tr>
<tr>
<td>kind &lt;&gt; #Economy</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CallEvent</td>
</tr>
<tr>
<td>Booked</td>
</tr>
<tr>
<td>cancel()</td>
</tr>
<tr>
<td>change()</td>
</tr>
<tr>
<td>startOff()</td>
</tr>
<tr>
<td>handle()</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CallAction</td>
</tr>
<tr>
<td>Passenger</td>
</tr>
<tr>
<td>name: Name</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flight</td>
</tr>
<tr>
<td>miles: int</td>
</tr>
</tbody>
</table>
```

(c) 2005–2006, Dr. H. Störrle, Dr. A. Knapp
6 – State machines
Triggers and events (1)

StateMachine Boarding

- BoardingControl
  - Ready

- ReadBoardingPass
  - entry / check validity
  - exit / read passenger id
  - closeFlight / defer

- CheckBoardingPass
  - entry / start query
  - do / blink
  - closeFlight / defer

- AcceptBoardingPass
  - entry / eject
  - do / release turnstile
  - closeFlight / defer

- RejectBoardingPass
  - entry / eject
  - closeFlight / defer

- SignalEvent
- completion event (no explicit trigger)

- TimeEvent (relative)
- deferred event

- ChangeEvent
6 – State machines
Triggers and events (2)

• **CallEvent**
  - receipt of a (a)synchronous Operation call
  - triggering after Behavior of Operation executed

• **SignalEvent**
  - receipt of an asynchronous Signal instance
  - reaction declared by a Reception for the Signal

• **TimeEvent**
  - absolute reference to a time point (at \( t \))
  - relative reference to trigger becoming active (after \( t \))
    • presumably meaning relative to state entry

• **ChangeEvent**
  - raised each time condition becomes true
    • may be raised at some point after condition changes to true
    • could be revoked if condition changes to false
6 – State machines
Triggers and events (3)

• completion event
  – raised when all internal activities of a state are finished
    • do activity, subregion
    • no metamodel element for completion events
  – dispatched before all other events in the event pool

• deferred events
  – events that cannot be handled in a state but should be kept in
    the event pool
    • reconsidered when state is changed
    • no predefined deferring policy

• internal transitions
  – ... are executed without leaving and
    entering their containing state
    • normally, on transition execution states are left and entered
6 – State machines
Behaviors

**SM Boarding**

- **BoardingControl Ready**
  - **ReadBoardingPass**
    - entry / check validity
    - exit / read passenger id
    - closeFlight / defer
  - **CheckBoardingPass**
    - entry / start query
    - do / blink
    - closeFlight / defer
  - **AcceptBoardingPass**
    - entry / eject
    - do / release turnstile
    - closeFlight / defer
    - after(10s) / block turnstile
  - **RejectBoardingPass**
    - entry / eject
    - closeFlight / defer
    - when(turnstile sensor="turn") / block turnstile

**Behaviors**
- **entry Behavior** (on entering a state)
- **exit Behavior** (on exiting a state)
- **do activity Behavior** (concurrently while in state, may be interrupted)
- **entry Behavior** (on entering a state)
6 – State machines

How state machines communicate

signals: asynchronous (no waiting)
calls: asynchronous or synchronous (waiting for RTC of callee)

No assumptions are made on timing between event occurrence, event dispatching, and event consumption.

Event occurrences for which no trigger exists may be discarded (if they are not deferred).
Hierarchical states allow to encapsulate behavior and facilitate reuse.
However, they are rarely used this way.
UML 2.0 provides concepts supporting this usage.
- entry and exit points

Transition triggering is prioritized inside-out, i.e., transitions deeper in the hierarchy are considered first.
6 – State machines
Hierarchical states (2)

OLC FlightHandling

- Preparation
  - startCheckIn
    - CheckIn
      - closeCheckIn
      - passengerCheckIn
    - passengerLeave
  - Boarding
    - BoardingControlReady
    - ReadBoardingPass
    - CheckBoardingPass
      - passenger has not checked in
        - Reject BoardingPass
      - passenger has checked in
        - Accept BoardingPass
      - passengerBoard
  - Closing
    - closeFlight
      - 5 min. to start & all checked-in passengers boarded

Detailed (non-orthogonal) composite State
Default entry
Region
Substates

(c) 2005–2006, Dr. H. Störrle, Dr. A. Knapp
6 – State machines
Orthogonal regions

- **Simple State**: containing no Region
- **Composite State**: containing at least one Region
  - simple composite State: exactly one
  - orthogonal composite State: at least two

Orthogonal states are “concurrent” as a single event may trigger a transition in each orthogonal region “simultaneously”
6 – State machines
Forks and joins

Fork Pseudostate
(one incoming, at least two outgoing Transitions;
outgoing Transitions must target States in different Regions of an orthogonal State)

Join Pseudostate
(restrictions dual to forks)

all Regions must be entered simultaneously

(\(X\) \(\rightarrow\) \(A\) \(\rightarrow\) \(B\) \(\rightarrow\) \(Z\))
(\(X\) \(\rightarrow\) \(C\) \(\rightarrow\) \(D\) \(\rightarrow\) \(Z\))

all Regions are left simultaneously
(if FinalStates are reached)

(\(Y\) \(\rightarrow\) \(A\) \(\rightarrow\) \(Z\))
(\(Y\) \(\rightarrow\) \(C\) \(\rightarrow\) \(Z\))
Entry and exit points (Pseudostates)
- provide better encapsulation of composite states
- help avoid “unstructured” transitions
6 – State machines

Entry and exit points (2)

Notational alternatives

Semantically equivalent

"unstructured" transitions
6 – State machines

History states

- History states represent the last active
  - substate (shallow history), or
  - configuration (deep history)

of a region.

shallow history Pseudostate
(enter last State in this Region)

deep history Pseudostate
(enter last States in this Region and all sub-Regions)
6 – State machines

Metamodel
Run-to-Completion Step: Overview

- Choose an event from the event pool (queue)
- Choose a maximal, conflict-free set of transitions enabled by the event
- Execute set of transitions
  - exit source states (inside-out)
  - execute transition effects
  - enter target states (outside-in)
thereby generating new events and activities
6 – State machines

Run-to-Completion Step: Preliminaries (1)

- **Active state configuration**
  - the states the state machine currently is in
  - forms a tree
    - if a composite state is active, all its regions are active

- **Least–common–ancestor (LCA) of states** $s_1$ and $s_2$
  - the least region or orthogonal state (upwards) containing $s_1$ and $s_2$
6 – State machines

Run–to–Completion Step: Preliminaries (2)

• Compound transitions
  – transitions for an event are “chained” into compound transitions
    • eliminating pseudostates like junction, fork, join, entry, exit
    • this is not possible for choice pseudostates where the guard of outgoing
      transitions are evaluated dynamically (in contrast to junctions)
  – several source and target states
6 – State machines
Run-to-Completion Step: Preliminaries (3)

• Main source / target state $m$ of compound transition $t$
  - Let $s$ be LCA of all source and target states of $t$
  - If $s$ region: $m = \text{direct subvertex of } s \text{ containing all source states of } t$
  - If $s$ orthogonal state: $m = s$
  - Similarly for main target state
  - All states between main source and explicit source states are exited, all state between main target and explicit target states are entered.

• Conflict of compound transitions $t_1$ and $t_2$
  - Intersection of states exited by $t_1$ and $t_2$ not empty

• Priority of compound transition $t_1$ over $t_2$
  - $s_i$ “deepest” source state of transition $t_i$
  - $s_1$ (direct or transitive) substate of $s_2$
RTC(env, conf) ≡
\[
\left[ event \leftarrow \text{fetch}() \right.
\]
\[
\left. step \leftarrow \text{choose steps}(conf, event) \right.
\]
\[
\text{if } step = \emptyset \land event \in \text{deferred}(conf) \text{ then } \text{defer}(event) \text{ fi}
\]
\[
\text{for } transition \in step \text{ do}
\]
\[
\text{conf} \leftarrow \text{handleTransition}(env, conf, transition)
\]
\[
\text{od}
\]
\[
\text{if } isCall(event) \land event \notin \text{deferred}(conf) \text{ then } \text{acknowledge}(event) \text{ fi}
\]
\[
conf]\]
6 – State machines

Run-to-Completion Step (2)

\[
\text{steps}(env, \text{conf}, \text{event}) \equiv \\
\left[ \begin{array}{l}
\text{transitions} \leftarrow \text{enabled}(env, \text{conf}, \text{event}) \\
\{\text{step} \mid (\text{guard}, \text{step}) \in \text{steps}(\text{conf}, \text{transitions}) \land env \models \text{guard} \} \end{array} \right]
\]

\[
\text{steps}(\text{conf}, \text{transitions}) \equiv \\
\left[ \begin{array}{l}
\text{steps} \leftarrow \{(\text{false}, \emptyset)\} \\
\text{for } \text{transition} \in \text{transitions} \text{ do} \\
\quad \text{for } (\text{guard}, \text{step}) \in \text{steps}(\text{conf}, \text{transitions} \setminus \{\text{transition}\}) \text{ do} \\
\quad\quad \text{if } \text{inConflict}(\text{conf}, \text{transition}, \text{step}) \\
\quad\quad\quad \text{then if } \text{higherPriority}(\text{conf}, \text{transition}, \text{step}) \\
\quad\quad\quad\quad \text{then } \text{guard} \leftarrow \text{guard} \land \neg \text{guard(transition)} \text{ fi} \\
\quad\quad\quad \text{else } \text{step} \leftarrow \text{step} \cup \{\text{transition}\} \\
\quad\quad\quad\quad \text{guard} \leftarrow \text{guard} \land \text{guard(transition)} \text{ fi} \\
\quad\quad \text{steps} \leftarrow \text{steps} \cup \{(\text{guard}, \text{step})\} \text{ od} \\
\quad\text{od} \\
\text{od}
\end{array} \right]
\]
6 – State machines

Run-to-Completion Step (3)

\[ \text{handleTransition}(\text{conf}, \text{transition}) \equiv \]
\[
\left[ \begin{array}{l}
\text{for } \text{state } \in \text{insideOut}(\text{exited}(\text{transition})) \text{ do} \\
\quad \text{uncomplete}(\text{state}) \\
\quad \text{for } \text{timer } \in \text{timers}(\text{state}) \text{ do } \text{stopTimer}(\text{timer}) \text{ od} \\
\quad \text{execute}(\text{exit}(\text{state})) \\
\quad \text{conf } \leftarrow \text{conf } \setminus \{\text{state}\} \\
\text{od} \\
\text{execute}(\text{effect}(\text{transition})) \\
\text{for } \text{state } \in \text{outsideIn}(\text{entered}(\text{transition})) \text{ do} \\
\quad \text{execute}(\text{entry}(\text{state})) \\
\quad \text{for } \text{timer } \in \text{timers}(\text{state}) \text{ do } \text{startTimer}(\text{timer}) \text{ od} \\
\quad \text{conf } \leftarrow \text{conf } \cup \{\text{state}\} \\
\quad \text{complete}(\text{conf}, \text{state}) \\
\text{od} \\
\text{conf} \right]
\]
6 – State machines
Semantic variation points

• Some semantic variation points have been mentioned before.
  – delays in event pool
  – handling of deferred events
  – entering of composite states without default entry

• Which events are prioritized?
  – Completion events only
  – All internal events (completion, time, change)

• Which (additional) timing assumptions?
  – delays in communication
  – time for run-to-completion step
    • zero-time assumption
6 – State machines
State machine refinement

- State machines are behaviors and may thus be refined.

`Control`

`Control {extended}`

not refined (may be omitted)

no refinement possible
Protocol state machines specify which behavioral features of a classifier can be called in which state and under which condition and what effects are expected.

- particularly useful for object life cycles and ports
- no effects on transitions, only effect descriptions
Several operation specifications are combined conjunctively:

context C::op()
pre: inState(S_1) and P_1
post: Q_1 and inState(S_3)

context C::op()
pre: inState(S_2) and P_2
post: Q_2 and inState(S_4)

results in

context C::op()
pre: (inState(S_1) and P_1) or (inState(S_2) and P_2)
post: (inState@pre(S_1) and P_1@pre) implies (Q_1 and inState(S_3))
and (inState@pre(S_2) and P_2@pre) implies (Q_2 and inState(S_4))
6 – State machines
UML 1.x vs. UML 2.0

• Consolidated metamodel
  – introduction of regions instead of composite states only
  – no transitions between regions of an orthogonal state
    • the “more esoteric case” of UML 1.x

• New encapsulation means
  – entry and exit points

• Protocol state machines
  – side-effect free

• Clarification of semantic variation points
  – but, still, no formal semantics
6 – State machines
How things work together

• **Static structure**
  – sets the scene for state machine behavior
  – state machines refer to
    • properties
    • behavioral features (operations, receptions)
    • signals

• **Interactions**
  – may be used to exemplify the communication of state machines
  – refer to event occurrences used in state machines

• **OCL**
  – may be used to specify guards and pre-/post-conditions
  – refers to actions of state machines (*OclMessage*)

• **Protocols and components**
  – state machines may specify protocol roles
6 – State machines

Wrap up

• State machines model behavior
  – object and use case life cycles
  – control automata
  – protocols

• State machines consist of
  – Regions and ...
  – … (Pseudo)States (with entry, exit, and do–activities) …
  – connected by Transitions (with triggers, guards, and effects)

• State machines communicate via event pools.

• State machines are executed by run–to–completion steps.
Unified Modeling Language 2.0

Part 7 – Interactions

Dr. Harald Störrle
University of Innsbruck
MGM technology partners

Dr. Alexander Knapp
University of Munich
7 – Interactions
A first glimpse

sequence diagram

communication diagram

timing diagram

all three are semantically equivalent
7 – Interactions
History and predecessors

• simple sequence diagrams
  – 1990’s
    • Message Sequence Charts (MSCs) used in TelCo–industry
    • several OO–methods use sequence diagrams

• complex sequence diagrams
  – 1996: Complex MSCs introduced in standard MSC96
  – 1999: Life Sequence Charts (LSCs)

• communication diagrams
  – 1991: used in Booch method
  – 1994: used in Cook/Daniels: Syntropy

• timing diagrams
  – traditionally used in electrical engineering
  – 1991: used in Booch method
  – 1993: used in early MSCs

• interaction overview
  – 1996: high–level MSCs (graphs of MSCs as notational alternative)
7 – Interactions

Usage scenarios

- **Class/object interactions**
  - design or document message exchange between objects
  - express synchronous/asynchronous messages, signals and calls, activation, timing constraints

- **Use case scenarios**
  - illustrate a use case by concrete scenario
  - useful in design/documentation of business processes (i.e. analysis phase and reengineering)

- **Test cases**
  - describe test cases on all abstraction levels

- **Timing specification/documentation**

- **Interaction overview**
  - organize a large number of interactions in a more visual style
  - defined as equivalent to using interaction operators
7 – Interactions

Syntactical variants

- **Sequence diagram**
  - traditional sequence diagrams + interaction operators
  - focuses on exchanging many messages in complex patterns among few interaction partners

- **Communication diagram**
  - “collaboration diagram” in UML 1.x
  - focuses on exchanging few messages between (many) interaction partners in complex configuration

- **Timing diagram**
  - new in UML 2.0, oscilloscope-type representation, not necessarily metric time
  - focuses on (real) time and coordinated state change of interaction partners over time

- **Interaction overview diagram**
  - looks like restricted activity diagram, but isn’t
  - arrange elementary interactions to highlight their interaction
7 – Simple Interactions

Main concepts

- Interaction partner
- Lifeline
- OccurrenceSpecification aka. EventOccurrence
- call
- reply
- asynchronous signal

Diagram: C/S-Protocol 1

(c) 2005–2006, Dr. H. Störrle, Dr. A. Knapp
7 – Simple Interactions

Message types

- instantiation Event
- lost & found Messages (i.e.: very slow messages)
- termination Event
- non-instantaneous Message
7 – Simple Interactions

Activation

- external Event
- activation bar
- activation suspended
- warp lines (non-UML)
- nested activation

Diagram:

Client

Server

S/QD C/S-Protocol 2

start()
callService(self, parameter)
jobNo = nextNumber()
jobParameter.store(jobNo, parameter, client)
waitingClients.nq(client)
pendingRequests.nq(jobNo)
result = execute(jobParameters.dq())

receiveResult(jobNo, result)
done

waitingClients.dq()
7 – Interactions

Usage: Use case scenarios

• Interaction participants are actors and systems rather than classes and objects.

• May be refined successively.

• Useful also for specifying high-level non-functional requirements such as response times.

• All kinds of interaction diagrams may be applied, depending on the circumstances.
7 – Interactions

Usage: Class interactions

- Interaction participants are classes and objects rather than actors and systems.

- Again, successive refinement may be applied in different styles:
  - break down processing of messages
  - break down structure of interaction participants.

- All kinds of interaction diagrams may be applied, depending on the circumstances.
7 – Interactions

Usage: Test cases

• Like any other interaction, but with a different intention.

• Typically accompanied by a tabular description of purpose, expected parameters and result (similar to use case description).

<table>
<thead>
<tr>
<th>identifier</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>TF-AAA.CIA-4</td>
<td>Check In (automatic) too much luggage</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>test goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>If a passenger has too many pieces of luggage and tries to check in using the check in machine, he should be referred to the check in counter.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>precondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>passenger is booked on respective flight</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>luggage, bonus mile card, booking data</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>passenger is referred to counter</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>postcondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>luggage is not checked in, passenger is checked in</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>remarks, open questions</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
</tr>
</tbody>
</table>
7 – Interactions

Usage: Timing specification

- For embedded and real-time systems, it may be important to specify absolute timings and state evolution over time.

- This is not readily expressed in sequence diagrams, much less communication diagrams.

- UML 2.0 introduces timing diagrams for this purpose.
7 – Interactions
Abstraction in timing diagram

- An alternative syntax presents states not on the vertical axis but as hexagons on the lifeline.

- Timing diagrams present the coordination of (the states of) several objects over (real) time.
7 – Interactions

Usage: Interaction overview

- Organize large number of interactions in a more visual style
- Defined as equivalent to using interaction operators

![Diagram](image)

- Sequence probably equivalent to `seq`
- Choice/merge equivalent to `alt/opt`

Also allowed: `fork/join` (said to be equivalent to `par`, but …)
7 – Interactions
Complex interactions

- A complex interaction is like a functional expression:
  - an InteractionOperator,
  - one or several InteractionOperands (separated by dashed lines),
  - (and sometimes also numbers or sets of signals).
7 – Interactions

Interaction operators (overview)

- **strict**
  - operand-wise sequencing
- **seq**
  - lifeline-wise sequencing
- **loop**
  - repeated seq
- **par**
  - interleaving of events
- **region (aka. “critical”)**
  - suspending interleaving
- **consider**
  - restrict model to specific messages
  - i.e. allow anything else anywhere
- **ignore**
  - dual to consider
- **ref**
  - macro-expansion of fragment
- **alt**
  - alternative execution
  - syntactic sugar for alt
- **opt**
  - optional execution
- **break**
  - abort execution
  - sometimes written as “brk”
- **assert**
  - remove uncertainty in specification
  - i.e. declare all traces as valid
- **neg**
  - declare all traces as invalid
  (→ three-valued semantics)
7 – Interactions
Main concepts (metamodel)
7 – Interactions
Semantics

• The meaning of an interaction is
  – a set of valid traces, plus
  – a set of invalid traces.
  
  *(ignore the latter for the time being)*

• Traces are made up of occurrences of events such as
  – sending/receiving a message,
  – instantiating/terminating an object, or
  – time/state change events.

• Two types of constraints determine the valid traces:
  1) send occurs before receive,
  2) order on lifelines is definite.

This diagram contains the following seven constraints:

1) a→d, e→b, f→c
2) a→b, b→c, d→e, e→f

The set of resulting traces is:
{ a.d.e.b.f.c, a.d.e.f.b.c }. 
7 – Interactions

Interaction operators seq & strict

- **seq**
  - compose two interactions sequentially lifeline-wise (default!)
- **strict**
  - compose two interactions sequentially diagram-wise
7 – Interactions

Interaction operator loop

- **loop**
  - repeated application of `seq`
    
    \[
    \text{loop}(P, \text{min}, \text{max}) = \text{seq}(P, \text{loop}(P, \text{min}-1, \text{max}-1))
    \]
    
    \[
    \text{loop}(P, 0, \text{max}) = \text{seq}(\text{opt}(P), \text{loop}(P, 0, \text{max}-1))
    \]
    
    \[
    \text{loop}(P, *) = \text{seq}(\text{opt}(P), \text{loop}(P, *))
    \]

  for some interaction fragment P
7 – Interactions

Interaction operators: interleaving

- **par**
  - shuffle arguments
- **region**
  - execute argument atomically, i.e. disallow interleaving

![Diagram](image)

- SND(a) → RCV(a)
- SND(b) → RCV(b)

- SND(a).RCV(a).SND(b).RCV(b)
- SND(a).SND(b).RCV(a).RCV(b)
- SND(b).SND(a).RCV(b).RCV(a)
- SND(b).RCV(b).SND(a).RCV(a)
7 – Interactions

Interaction operators alt, opt, brk: choice

• alt
  – alternative complete execution of one of two interaction fragments

• opt
  – optional complete execution of interaction fragment:
    \[ \text{opt}(P) = \text{alt}(P, \text{nop}) \]

• break
  – execute interaction fragment partially, skip rest, and jump to surrounding fragment
7 – Interactions

Interaction operators: abstraction

- ignore, consider
  - dual way of expressing:
    - allow the ignorable messages (!) anywhere
    - present only those messages that are to be considered
    - $\langle \text{ignore}(P, Z) \rangle = \text{shuffle}(\left[ P \right], Z^*)$
7 – Complex Interactions

Interaction operator ref & parameters

• ref
  - refers to a fragment defined elsewhere (macro-expansion)
  - Formal and actual parameters (bindings) are declared in the diagram head.

• Signals to the containing classifier appear as arrows form the diagram border.

(c) 2005–2006, Dr. H. Störrle, Dr. A. Knapp
7 – Interactions

Interaction operators: negation

- The semantics of neg and assert is unclear.
- In contrast to that the other operators, they refer not just to the positive traces, but to invalid and inconclusive traces as well.

- **neg**
  - declare all valid traces as invalid
  - inconclusive traces: unknown

- **assert**
  - remove uncertainty by declaring all inconclusive traces as invalid
## UML 1.x vs. UML 2.0

<table>
<thead>
<tr>
<th>UML 1.0</th>
<th>UML 2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>• collaboration diagram</td>
<td>• communication diagram</td>
</tr>
<tr>
<td></td>
<td>• timing diagram</td>
</tr>
<tr>
<td></td>
<td>• interaction overview diagram</td>
</tr>
<tr>
<td></td>
<td>• complex interaction</td>
</tr>
<tr>
<td></td>
<td>• Lifeline is now a first-class citizen</td>
</tr>
</tbody>
</table>
7 – Interactions

Wrap up

• Complex interactions like high-level MSCs added.

• New diagram types:
  – timing diagrams (like oscilloscope), and
  – interaction overview (similar to restricted activity diagram)
  – renamed collaboration diagram to communication diagram

• Completely new metamodel.

• Almost formal three-valued semantics of valid, invalid and inconclusive interleaving traces of events.

• Some semantical problems are yet to be solved.
Unified Modeling Language 2.0

Part 8 – Tools

Dr. Harald Störrle
University of Innsbruck
MGM technology partners

Dr. Alexander Knapp
University of Munich

(c) 2005–2006, Dr. H. Störrle, Dr. A. Knapp
8 – Tools
Overview

- UML 2.0 modeling tools
  - subjective selection for test
  - not an evaluation

- What has been covered
  - UML 2.0 diagrams
  - UML 2.0 metamodel
  - import/export
  - special features

- There are many more, like
  - Omondo: Omondo for Eclipse
  - Sparx Systems: Enterprise Architect

Rhapsody
TAU
MagicDraw
Software Modeler
Poseidon
Together Architect

(c) 2005–2006, Dr. H. Störrle, Dr. A. Knapp
8 – Tools

I-Logix: Rhapsody
8 – Tools
I-Logix: Rhapsody

• Tested version: Rhapsody V6.0 in C++
  – mainly targeted on embedded systems design and real-time operation systems

• Fair UML 2.0 support
  – but sometimes deviating terminology

• Nice features
  – code generation based on templates
    • mainly for state machines
  – support for structured analysis/design

www.ilogix.com
8 – Tools
Telelogic: TAU/Developer
8 – Tools
Telelogic: TAU/Developer

• Tested version: TAU V2.4

• Fair UML 2.0 support
  – import from XMI (Rose, Together)

• Nice features:
  – code generation based on libraries
  – continuous consistency checks
    • some messages not overly instructive
  – UML 2 textual syntax
8 – Tools
NoMagic: MagicDraw
8 – Tools

NoMagic: MagicDraw

• Tested version: MagicDraw 11.5 Enterprise

• Very good UML 2.0 support
  – sometimes deep nesting of property sheets
  – export as XMI 2.1, EMF

• Nice features
  – OCL syntax check
    • but not more
  – metamodel-based model comparison
  – model metrics

www.nomagic.com
IBM: Rational Software Modeler
8 – Tools

IBM: Rational Software Modeler

- Tested version: Rational Software Modeler Trial
- Good UML 2.0 support
  - some features are deep down in property sheets
  - export as UML2 (XMI 2.0), EMF, ...
- Nice features
  - Integrated into Eclipse
  - ensures consistency by selection from available features and drawing restrictions
    - but not for constraints

www.ibm.com/rational
8 – Tools
Gentleware: Poseidon
8 – Tools
Gentleware: Poseidon

• Tested version: Poseidon 4.2.1 community edition
  – professional versions include code generation, version control, Eclipse integration, ...

• Good UML 2.0 support
  – but no templates, composite structures, ...
  – export as XMI 1.2

• Nice features
  – UML 2.0 diagram interchange
  – Community edition for free

www.gentleware.com
8 – Tools
Borland: Together Architect
8 – Tools

Borland: Together Architect

- Tested version: Together Architect 2006, version 8.0

- Fair UML 2.0 support
  - export as XMI 2.0

- Nice features
  - Eclipse integration
  - Good OCL support
    - OCL-based model transformations
  - ECore API

www.borland.com
## 8 - Tools

### Comparison (1)

<table>
<thead>
<tr>
<th></th>
<th>Rhapsody</th>
<th>TAU/Developer</th>
<th>Magic Draw</th>
<th>Software Modeler Trial</th>
<th>Poseidon CE</th>
<th>Together Architect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Composite structure</td>
<td>×</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>×</td>
<td>●</td>
</tr>
<tr>
<td>Component</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
</tr>
<tr>
<td>Object</td>
<td>●</td>
<td>×</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
</tr>
<tr>
<td>Deployment</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Package</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
</tbody>
</table>

- **●** good (all important features present)
- ○ average (some important features missing)
- × not available

(c) 2005-2006, Dr. H. Störrle, Dr. A. Knapp
# 8 – Tools
## Comparison (2)

<table>
<thead>
<tr>
<th></th>
<th>Rhapsody</th>
<th>TAU/Developer</th>
<th>Magic Draw</th>
<th>Software Modeler Trial</th>
<th>Poseidon CE</th>
<th>Together Architect</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Activity</strong></td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
</tr>
<tr>
<td><strong>Use case</strong></td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
</tr>
<tr>
<td><strong>State machine</strong></td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td><strong>Sequence</strong></td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td><strong>Communication</strong></td>
<td>●</td>
<td>×</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td><strong>Interaction overview</strong></td>
<td>×</td>
<td>●</td>
<td>●</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td><strong>Timing</strong></td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

- ● good (all important features available)
- ○ average (some important features missing)
- × not available
8 – Tools
Which one is best for me?

• Many tools claim to support UML (or even UML 2.0), but few do justice to this claim.

• Of those that come anywhere close to UML 2.0, there is no single best tool.

• If you want to select a tool for you, your company, or your organization, go ahead as follows.
  - Make a short list of 3–6 candidate tools following crude criteria like price, platform, size of tool manufacturer, previous experience, and expert advice.
  - Determine evaluation criteria like required notations, input/output file formats, reporting/printing capabilities, code generation facilities and so on.
  - Evaluate all candidate tools – a UML expert will be able to do a reasonable (superficial) analysis of any tool in half a day.
Unified Modeling Language 2.0

Part 9 – Best Practices for the management of large models

Dr. Harald Störrle
University of Innsbruck
MGM technology partners

Dr. Alexander Knapp
University of Munich
9 – Best practices
Management of large models

• Creating and handling small models presents some challenges.

• But managing large models is a problem in its own right, which comes in addition to all of these:
  – versioning, diff/patch, merge
  – migration between tool platforms
  – long term maintenance of models
  – round-trip with manual interference
  – measures, queries, checks, analysis of models
  – simulation, code generation
  – reporting

• Today, we don’t have appropriate tool support for the majority of these tasks, and it is very cumbersome to do it by hand.
9 – Best practices
What exactly means “large” for a model?

- **Project size (only model-related activities!)**
  - Manpower: 5–100 Persons (rather: Person-equivalents)
  - Duration: 1–10 Years
  - Budget/Cost: 1–50 Mio €

- **Number of model elements (“population”)**
  - 200–5,000 classes (times 10–20 attributes)
  - 100–1,000 business processes (times 10–20 functions, steps)
  - 5–10/10–20/20–50 systems/subsystems/segments
  - 50–200 interfaces

- **Compare with large scale software systems, e.g. SAP R/3**
  - over 100 Mio LoC, more than 33,000 database tables
  - 14 systems, 35 subsystems, ca. 32,000 programs
  - ca. 2,500 interfaces
9 – Best practices
Probleme and gaps in large modeling projects

Characteristics of large projects

- **overal situation**
  - often extremely “political“ environment
  - inhomogeneous, large organisation
  - long and critical previous project history
  - very long project duration
  - extreme expectations, big dissappointments
  - hostile competitors involved („Mehrfrontenkrieg“)

- **Qualifications**
  - Customer
  - Colleagues
  - oneself

- **Work organisation**
  - several companies and organisations involved
  - distribution over several places

Specific for modeling projects

- **Tools**
  - inappropriate tools previous decisions
  - Untaugliche Werkzeuge gesetzt
  - überhaupt keine Werkzeuge verfügbar
  - Versionsverwaltung/Diff selten
  - Releases, Auslieferung, Sicherung

- **structuring of models and method**
  - “the usual suspects” are insufficient

- **Quality of models**
  - what does it mean in the first place?
  - consistency
  - coherene and validity
  - clear focus („big picture“)
  - adherance to conventions

many gaps… …but each gap is also a starting point!
9 – Best practices
Starting points in large modeling projects

- Model structure and methodology
  - no/few established standards thus much leeway
  - impact on almost all other areas
  - requires intensive training and coaching („Navigation overview“)

- Model design
  - Layout, naming conventions
  - Guidelines for model sizes and levels of abstractions
  - Change markers
  - Plan header
  - Attribute states (open questions, defaults, errors)

- Organisation of project
  - Quality assurance criteria
  - Distributed work
  - Process of modeling, tasks

- Conviction
  - large and demotivated teams must be convinced and activated
  - support for standards such as
    - poster of model inventory
    - navigation overview
    - coaching (less useful: trainings)
    - handzettel mit Arbeitsanleitungen

- Automatisation / Integration
  - XMI (e.g. Modelbridge)
  - self-made tools, e.g.
    - naming conventions
    - measures for size/complexity
    - reporting
    - generating
9 – Best practices
Model, Diagram, Plan

XMI, MDL, ADL, …

UML, EPK, ERD, …

projectspecific

(c) 2005–2006, Dr. H. Störrle, Dr. A. Knapp
9 – Best practices
Model, Diagram, Plan

• Model
  – A Modell is an abstract entity, existing e.g. in a data structure
  – Parts of models may be models again
  – standardised (XMI) or proprietary (MDL, ADL, …)

• Diagram
  – a diagram is a either
    • the visual presentation of a model,
    • or an informal sketch.
  – A diagram defines a model: the one consisting of those model elements that are presented visually.

• Plan
  – A plan is a diagram in a frame of reference.
9 – Best practices

Model information

• Title
  - Name
  - pragmatic type

• Text field
  - Author/Manager
  - Customer/Project
  - date/version
  - view, phase, intention
  - scale, section, unit
  - QA status

• Legend
  - Stereotypes

SQD C/S-Protocol 2

Legend

UML element with textual stereotype

visual stereotype

<< unix-server >>

<< mainframe >>

<< switch >>
9 – Best practices
Role model civil architecture: detail section of a plan
9 – Best practices
Role model civil architecture: plan header (DIN 6771)

<table>
<thead>
<tr>
<th>Verwendungsbereich</th>
<th>(zul. Abw.)</th>
<th>(Gebenicht)</th>
<th>(Entw.)</th>
<th>(Gewicht)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Werkstoff</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reihennummer</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modell-Nr.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Datum</th>
<th>Name</th>
<th>(Benennung)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bearb.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Siepr.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Norm</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Firma: Zeichnungshersteller</th>
<th>(Zeichnungsnummer)</th>
<th>Blatt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Zust</th>
<th>Änderung</th>
<th>Datum</th>
<th>Name</th>
<th>(Unpt.)</th>
<th>(Erst. f)</th>
<th>(Erst. a)</th>
</tr>
</thead>
</table>

- other relevant ISO standards
  - ISO 128:1996 Technical Drawings (in 29 parts)
  - ISO 3098:1997 Lettering (in 7 parts)
  - ISO 7200:1984 Technical drawings — Title blocks
  - ISO 5455:1979 Scales
  - ISO 5457:1999 Drawing sheet formats for technical documentation
  - ISO 13567:1998 Technical product documentation — Organization and naming of layers for CAD
9 – Best practices
Role model civil architecture: plan header real life example
9 - Best practices
Role model civil architecture: plan header in tools

• If there is no support for model headers (almost always the case) use comment boxes: more effort but feasible and better than nothing.

• Administrative information of this kind should be presented (partially) in a plan header.

• Filling slots like predefined Values and state transitions should be supported.

• Reports on qa-status, version, model type etc. are important.

• If there is no support for model headers (almost always the case) use comment boxes: more effort but feasible and better than nothing.
9 – Best practices

Contents of a legend

• legend
  - Depending on the audience, one might need descriptions of
  - the complete notation
  - stereotypes only
  - colour coding of model changes

• change marker
  - Lists of added, removed, and modified model elements
9 – Best practices
Putting a legend in a plan

The diagram illustrates a process flow with various activities and decisions. The legend on the right side explains the symbols used in the diagram:

- **eEPK**: Event or Process Knowledge
- **UML AD**: UML Activity Diagram
  - **Name**: Activity or Event
  - **Ereignis**: Event
  - **and**: Logical AND operator
  - **or**: Logical OR operator
  - **xor**: Exclusive OR operator
  - **{and}**: Multiple AND
  - **{or}**: Multiple OR
  - **{xor}**: Multiple XOR
  - **Daten**: Data
  - **Klasse**: Class

The process starts with a Brief going in and involves tasks such as checking the claim and processing it by the Poststelle and Sachbearbeiter. The flow includes decisions like granting approval or formulating rejection, and further activities such as bonus miles calculation and booking. The diagram also highlights the need for handling exceptions and completing the case.
9 – Best practices
Change markers
9 – Best practices
Change markers
9 – Best practices
States of attributes

[Diagram showing states of attributes with transitions: unchecked → check → checked, unchecked → modify, unchecked → ??, unchanged → ??, unchanged → ok → empty, unchanged → ?? → open question, unchanged → <Value> → filled, unchecked → modify → unchecked.

States: work in progress, submitted for approval, partially approved, qa approved.}
9 – Best practices
Alternatives for model storage – Pros/cons

File
• e.g. Magic Draw, Rose

• storage in a single file
  – size
  – conflicting access
  – distributed work

• storage in redundant files
  – consistency

• storage of non-overlapping parts
  in a directory tree
  – references
  – integration

Repository/database
• e.g. StP, Adonis

• storage in tool–repository
  – distributed work
  – versioning
  – back up

• structuring facilities
  – …of the tool
  ➔ grouping / tree

  – …of the modeling language
  ➔ packages, classes
9 – Best practices
Versioning – the problem

• Only very few tools have appropriate functionality. Marketing is often more advanced than reality.

• It is possible to store your models in a CM tool, but...
  – Some tools are DB-based (e.g. StP, Adonis), so that models must be extracted/exported first (often manually), which is error prone and tedious.
  – The extraction format may be (that is, in reality it always is) difficult to interpret and process (e.g. diff of XMI files including diagrams).
  – Even if the modell is well structured, this does not guarantee that the modell-Dump is well structured, too.

• So, probably there is no model version control system available when you want it!
• Therefore, you need to resort to the „poor man‘s repository“.
9 – Best practices
Versioning – Alternatives

• **Case 1: a small group of modellers**
  - versioning only by backups
  - coordination directly (bilaterally) between all people involved
  - may become critical under spatial distribution

• **Case 2: model structure similar to project structure**
  - The whole model is structured in 1 overarching part and n more specific parts, depending only on the overarching part.
  - Each of these n+1 parts is created and modified by exactly one group (everybody else may read). Within each group, case 1 applies.
  - The groups are coordinated by a special group, e.g. formed by the group leaders.

• **Case 3: Chaos**
  - Get a new job.
9 – Best practices
Creating good diagrams

• Naming conventions
  − There must be conventions for names and abbreviations.
  − There must be a glossary to describe the terminology of the project, including domain-specific names.

• Graphic design conventions
  − The graphic of a diagram (layout, color, size, pen etc.) is essential for the usability of the model it represents, e.g.:
    • discussing and modeling,
    • presentation,
    • quality assurance,
    • implementation.
  − Thus, a good graphical design is an essential part of the model, equally important than the “contents itself”.

  − Bad diagrams often indicate bad models, for modeling errors are less apparent when there are many other errors around.
9 – Best Practices
Creating good diagrams: Names

- A name should express what an element is about. Good names are important!

- The same things should follow a consistent naming schema, so that the name already hints at what an element is supposed to be.
  - system/subsystem/group of use cases: noun, gerund + noun, e.g. Payment
  - business process: gerund + noun, e.g. awarding Miles
  - business function: verb noun, e.g. select flight
  - class/attribute: noun, e.g. passenger number, flight, booking
9 – Best Practices
Creating good diagrams: Names

- **Subsysteme**
  - Noun | Nounphrase
    (also substantivised verb)
  - Names of previously used systems (abbreviations!)
    → "Document management"
    → "Order book"
    → "Invoice"

- **Schnittstellen**
  - From ‘-’ To
  - fixed and well known names
    → "DS052"
    → "DMS-BInfo"

- **Business process and functions**
  - Verb Nounphrase
    → "file application"
    → "assess payment according to law XY and check solvency (manually)"

- **Conditions**
  - [Nounphrase] (Adjective | Adverb)
    → "tax identification code present"
    → "done"

- **Adherence of conventions**
  - Glossary
  - Automated checker
9 – Best Practices
Creating good diagrams: Layout

- A model should be complete, non-redundant, clear, and adequate.

- **Completeness**
  - All relevant facts are contained in the model and displayed according to their importance.

- **Non-redundancy**
  - No part of a model is displayed more than once except there is a good reason.

- **Clarity**
  - There should be around 7±2 (main) elements per screen/paper page.
  - If necessary, split diagrams or introduce abstractions. If the resulting system of diagrams is a tree, the tree should be balanced.

- **Adequacy**
  - know your audience: what aspect is particularly interesting for *this* audience?
  - What is the purpose of this diagram, why do I draw it in the first place? Is this goal achieved?
  - Is there a better way to achieve this goal, such as using another diagram type, another layout altogether, or something else?
9 – Best Practices
Creating good diagrams: Layout

• Observe the Gestalt–laws
  – present similar things similarly
    • Things presented differently will be perceived as different things.
  – uniform size, color, orientation, alignment
    • Things of similar importance should be present in approximately the same size.
    • Things presented in the same way will be perceived as similar.
  – Non-uniform presentation transports (unwanted) information

• Observe reading order
  – left right, top bottom (at least in the west)
  – clockwise arrangement for states

• Layout
  – Avoid crossings, strive for clarity

• Further aspects
  – Use colors, pen sizes, fonts, etc. very sparingly (consider printability)
  – If you do use them, use them carefully, and make sure who you’re talking to.
9 – Best Practices
Industrial experiences

- Contrary to common belief, many domain experts are quite happy when confronted with UML diagrams – analysis level only, of course.

- With UML 2, many things may now be captured, which were difficult to capture before.

- The tool support is not yet sufficient, however, partly due to the enormous complexity of the UML.

- Bottom line: it’s a step ahead, but we’re not yet there.
9 – Best Practices
A look into the crystal ball

• It’s very likely, that a version UML 2.1 will be coming to sort out the problems that are currently contained in the UML.
• There might also be UML 2.2 and UML 2.3 – but will there be a UML 3.0?

• There can only be one *unified* modeling language, though there will probably be simpler modeling languages.

• Domain–specific languages are neither unified, nor (usually) simpler than UML, and hard evidence of their other claimed benefits are nowhere to be seen.