2. Metamodelling and Metaprogramming

1. Metalevels and the metapyramid
2. Metalevel architectures
3. Metaobject protocols (MOP)
4. Metaobject facilities (MOF)
5. Component markup

Prof. Dr. Uwe Aßmann
Technische Universität Dresden
Institut für Software- und Multimediatechnik
http://st.inf.tu-dresden.de

11-0.1, Apr 5, 2011

Mandatory Literature

- ISC, 2.2.5 Metamodelling
- Rony G. Flatscher. Metamodeling in EIA/CDIF — Meta-Metamodel and Metamodels. ACM Transactions on Modeling and Computer Simulation, Vol. 12, No. 4, October 2002, Pages 322–342. http://doi.acm.org/10.1145/643120.643124

Other Literature

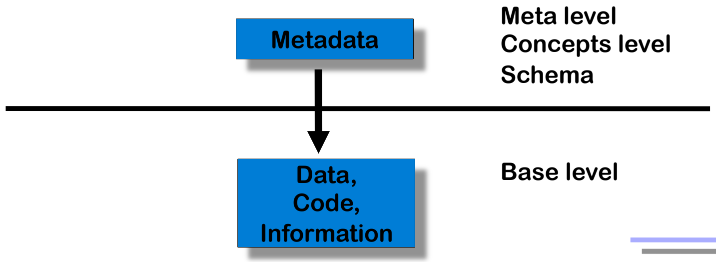
- Ira R. Forman and Scott H. Danforth. Metaclasses in SOM-C++ (Addision-Wesley)
- Squeak – a reflective modern Smalltalk dialect http://www.squeak.org
- Hauptseminar on Metamodelling held in SS 2005
- OMG MOF 2.0 Specification http://www.omg.org/spec/MOF/2.0/
- MDA Guide http://www.omg.org/cgi-bin/doc?omg/03-06-01
- J. Frankel. Model-driven Architecture. Wiley, 2002. Important book on MDA.
- G. Kizcales, Jim des Rivieres, and Daniel G. Bobrow. The Art of the Metaobject Protocol. MIT Press, Cambridge, MA, 1991
- Gregor Kiczales and Andreas Paepcke. Open implementations and metaobject protocols. Technical report, Xerox PARC, 1997

2.1. An Introduction into Metalevels

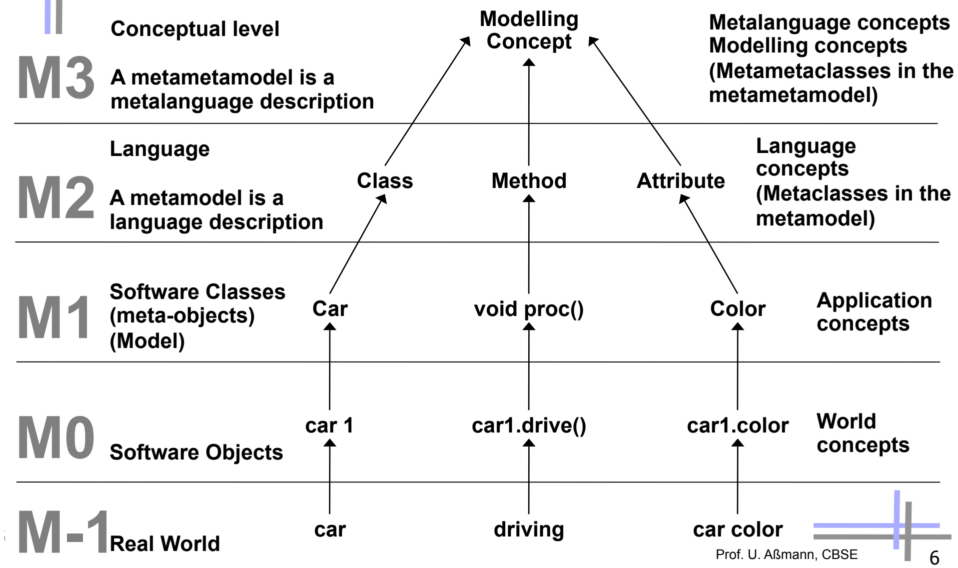
"A system is about its domain.
A reflective system is about itself"
Maes, 1988

Metadata

- Meta: greek for "describing"
- Metadata: describing data (sometimes: self describing data). The type system is called metamodel
- Metalevel: the elements of the meta-level (the meta-objects) describe the objects on the base level
- Metamodeling: description of the model elements/concepts in the metamodel
- Metalanguage: a description language for languages



Metalevels in Programming Languages (The Meta-Pyramid)

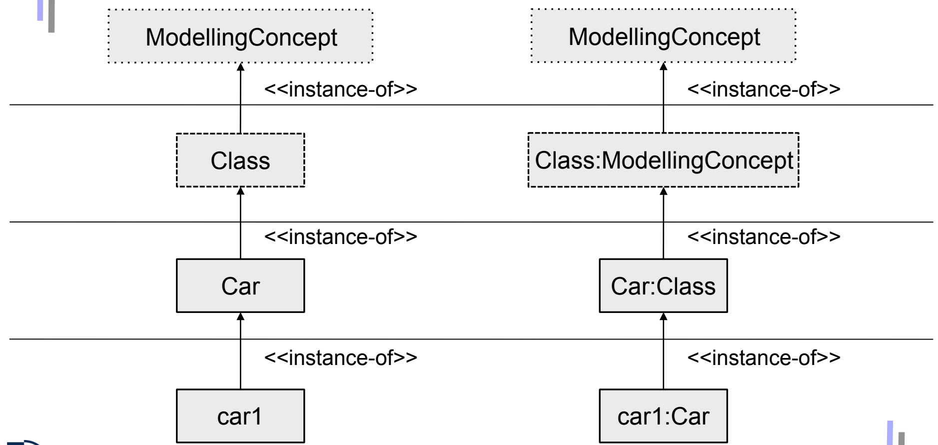


Different Types of Semantics and their Metalanguages (Description Languages)

- Structure
  - Described by a *context-free grammar* or a *metamodel*
  - Does not regard context
- Static Semantics (context conditions)
  - Described by *context-sensitive grammar (attribute grammar, denotational semantics, logic constraints),* or a *metamodel*
  - Describes context constraints, context conditions
  - Can describe consistency conditions on the specifications
    - "If I use a variable here, it must be defined elsewhere"
    - "If I use a component here, it must be alive"
- Dynamic Semantics
  - Interpreter in an *interpreter language (e.g., lambda calculus),* or a *metaobject protocol*
  - Sets of runtime states or terms

Notation

- We write metaclasses with dashed lines, metametaclasses with dotted lines



Classes and Metaclasses

- Metaclasses are *schemata* for classes, i.e., describe what is in a class

Classes in a software system

```
class WorkPiece       { Object belongsTo; }
class RotaryTable     { WorkPiece place1, place2; }
class Robot           { WorkPiece piece1, piece2; }
class Press           { WorkPiece place; }
class ConveyorBelt    { WorkPiece pieces[]; }
```

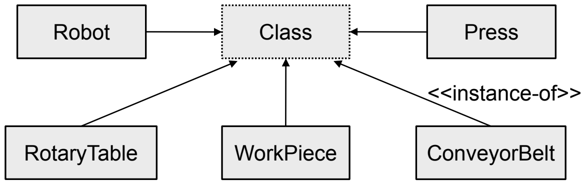
Metaclasses

```
public class Class {
   Attribute[] fields;
   Method[] methods;
   Class(Attribute[] f, Method[] m) {
     fields = f;
     methods = m; }}

public class Attribute {
   Object type;
   Object value; }

public class Method {
   String name; List parameters, MethodBody body; }

public class MethodBody { ... }
```


Creating a Class from a Metaclass

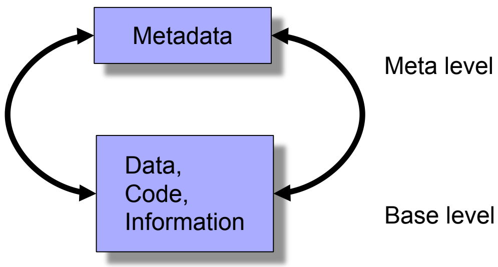
- Using the constructor of the metaclass (Pseudojava used here)
- Then, classes are special objects, instances of metaclasses

```
Class WorkPiece       = new Class(
          new Attribute[]{ "Object belongsTo" },
          new Method[]{});
Class RotaryTable     = new Class(
          new Attribute[]{ "WorkPiece place1", "WorkPiece place2" },
         new Method[]{});
Class Robot           = new Class(
          new Attribute[]{ "WorkPiece piece1", "WorkPiece piece2" },
         new Method[]{});
Class Press           = new Class(
          new Attribute[]{ "WorkPiece place" }, new Method[]{});
Class ConveyorBelt    = new Class(
          new Attribute[]{ "WorkPiece[] pieces" }, new Method[]{});
```



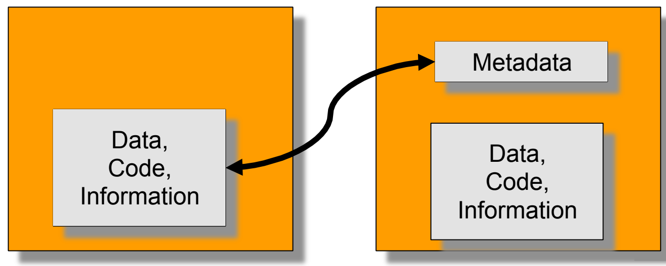
Reflection (Self-Modification, Intercession, Metaprogramming)

- Computation about the metamodel in the model is *reflection*
  - Reflection: thinking about oneself with the help of metadata
  - The application can look at their own skeleton and change it
    - Allocating new classes, methods, fields
    - Removing classes, methods, fields
- This self modification is also called *intercession* in a meta-object protocol



Introspection

- Read-only reflection is called *introspection*
  - The component can look at the skeleton of itself or another component and learn from it (but not change it!)
- Typical application: find out features of components
  - Classes, methods, attributes, types
- Very important in component supermarkets



Reading Reflection (Introspection)

Used for generating something based on metadata information

```
for all c in self.classes do
    generate_for_class_start(c);

    for all a in c.attributes do
        generate_for_attribute(a);
    done;

    for all m in c.methods do
        generate_for_method(m);
    done;

    generate_for_class_end(c);
done;
```

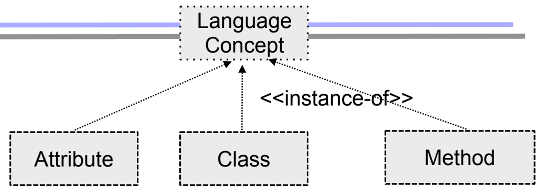

Full Reflection (Intercession)

Generating code, interpreting, or loading it

```
for all c in self.classes do
    helperClass = makeClass(c.name+"Helper");

    for all a in c.attributes do
       helperClass.addAttribute(copyAttribute(a));
    done;

    self.addClass(helperClass);
done;
```

A reflective system is a system in which the application domain is *causally connected* with its own domain.
Patti Maes

Metaprogramming on the Language Level




```
enum { Singleton, Parameterizable } BaseFeature;
public class LanguageConcept {
   String name;
   BaseFeature singularity;
   LanguageConcept(String n, BaseFeature s) {
     name = n;
     singularity = s;   }
}
```

Metalanguage concepts
Language description concepts
(Metametamodel)

Language concepts
(Metamodel)

```
LanguageConcept Class = new LanguageConcept("Class", Singleton);
LanguageConcept Attribute = new LanguageConcept("Attribute", Singleton);
LanguageConcept Method  = new LanguageConcept("Method", Parameterizable);
```


Made It Simple

- Level M0: objects
- Level M1: programs, classes, types
- Level M2: language
- Level M3: metalanguage, language description language

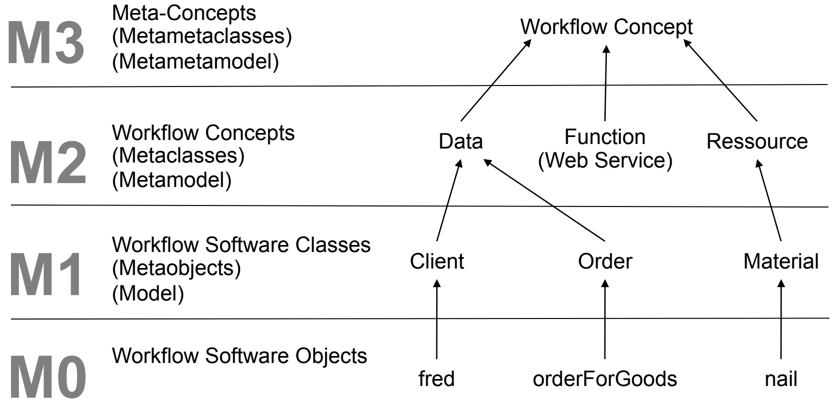
Use of Metamodels and Metaprogramming

To model, describe, introspect, and manipulate all sorts of objects, models, and languages:

- UML
- Workflow systems
- Databases (Common Warehouse Model, CWM)
- Programming languages
- Component systems, such as CORBA
- Composition systems, such as Invasive Software Composition
- ... probably all systems...

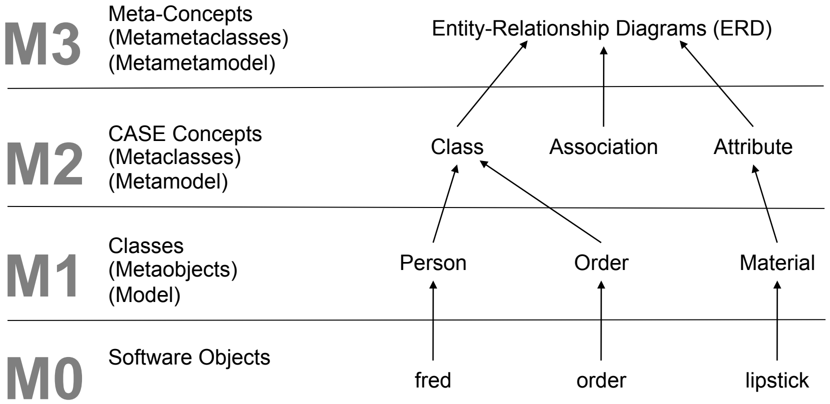
Metapyramid in Workflow Systems and Web Services (e.g., BPEL)

- It is possible to specify workflow languages with the metamodelling hierarchy
- BPEL and other workflow languages can be metamodeled



Metaprymid CASE Data Interchange Format (CDIF)

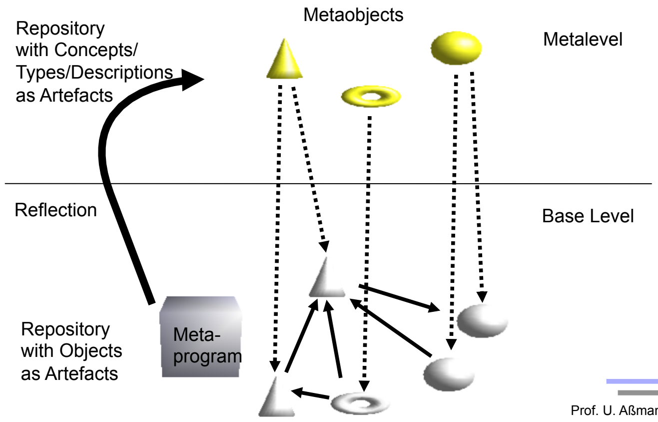
CDIF uses entities and relationships on M3 to model CASE concepts on M2



2.2 Metalevel Architectures

Reflective Architecture

- A system with a reflective architecture maintains *metadata* and a *causal connection* between meta- and base level.
  - The metaobjects describe structure, features, semantics of domain objects. This connection is kept consistent
- Metaprogramming is programming with metaobjects



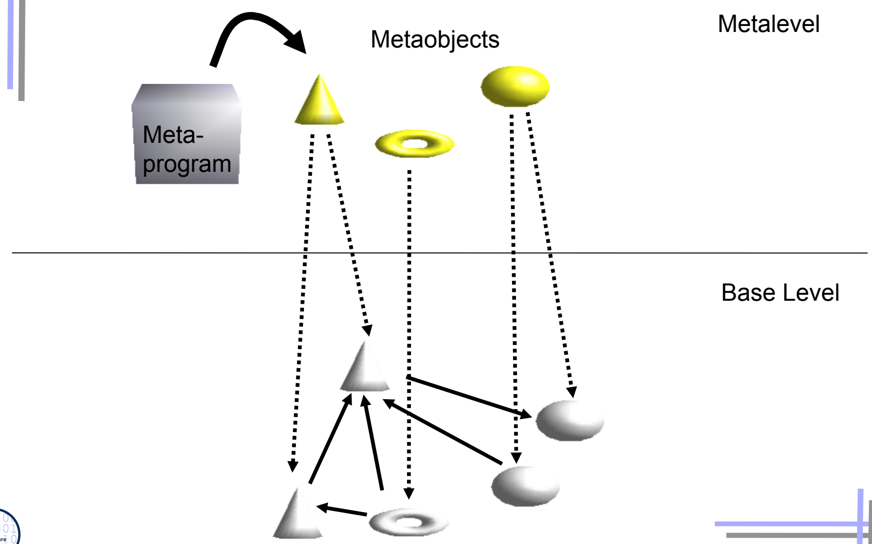
Examples

- 24/7 systems with total availability
  - Dynamic update of new versions of classes
  - Telecommunication systems
  - Power plant control software
  - Internet banking software
- Self-adaptive systems
  - Systems reflect about the context *and* themselves and, consequently, change themselves
- Reflection is used to think about versions of the systems
  - Keeping two versions at a time

Metalevel Architecture

- In a metalevel architecture, the metamodel is used for computations,
  - but the metaprograms execute either on the metalevel or on the base level.
  - supports metaprogramming, but not full reflection
- Special variants that *separate* the metaprogram from the base level programs
  - *Introspective architecture* (no self modification)
  - *Staged metalevel architecture* (metaprogram evaluation time is different from system runtime)

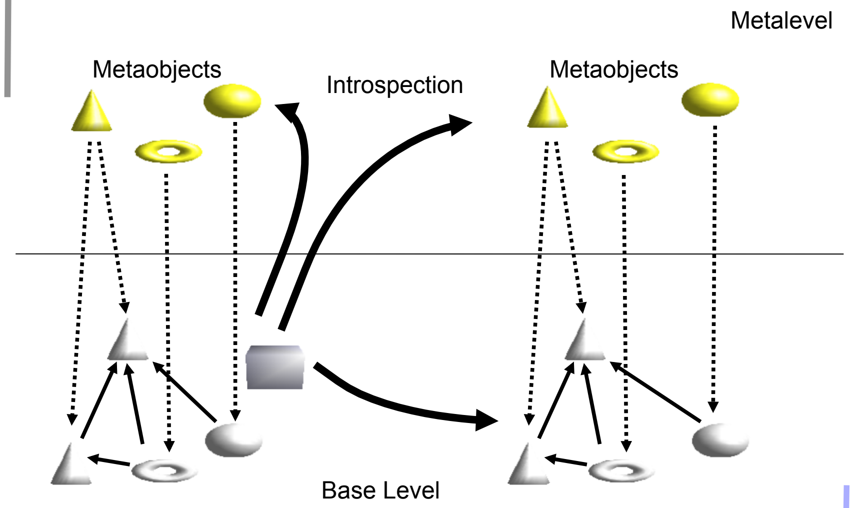
Metalevel Architecture



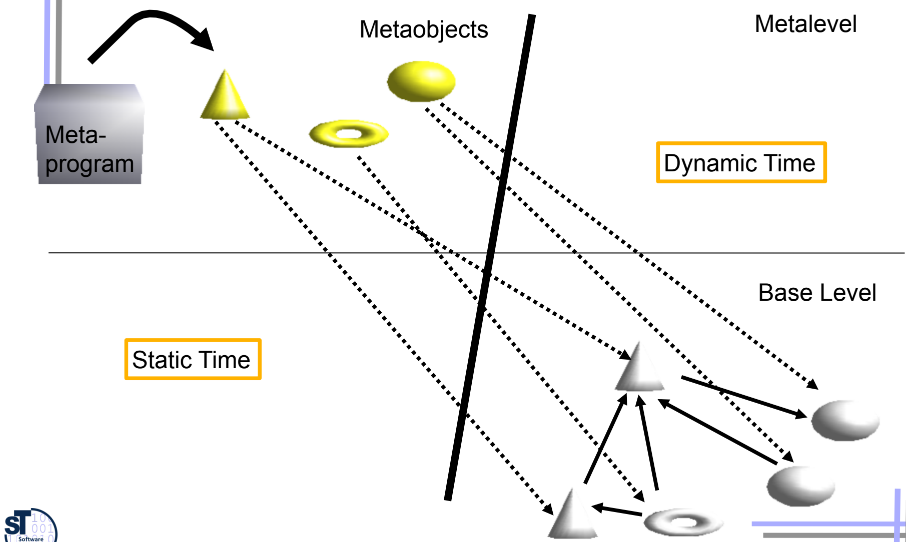
Examples

- Integrated development environment
  - Refactoring engine
  - Code generators
  - Metric analyzers (introspective)

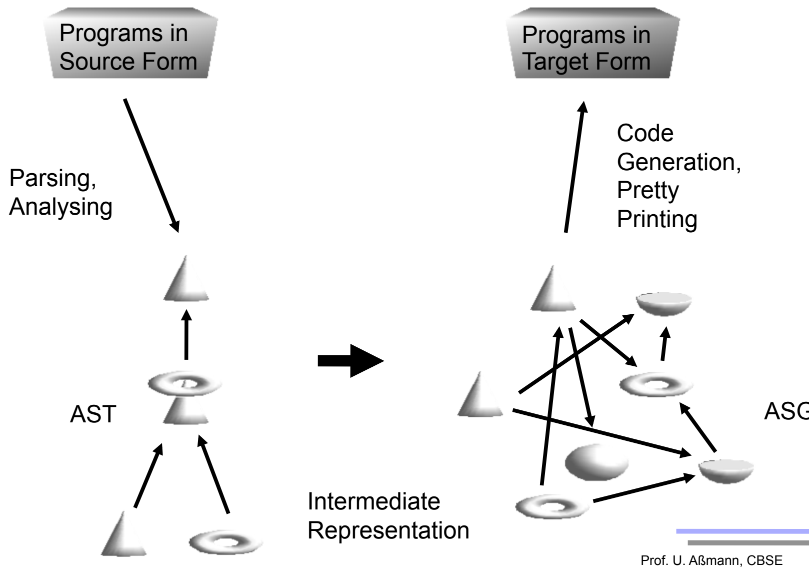
Introspective Architectures



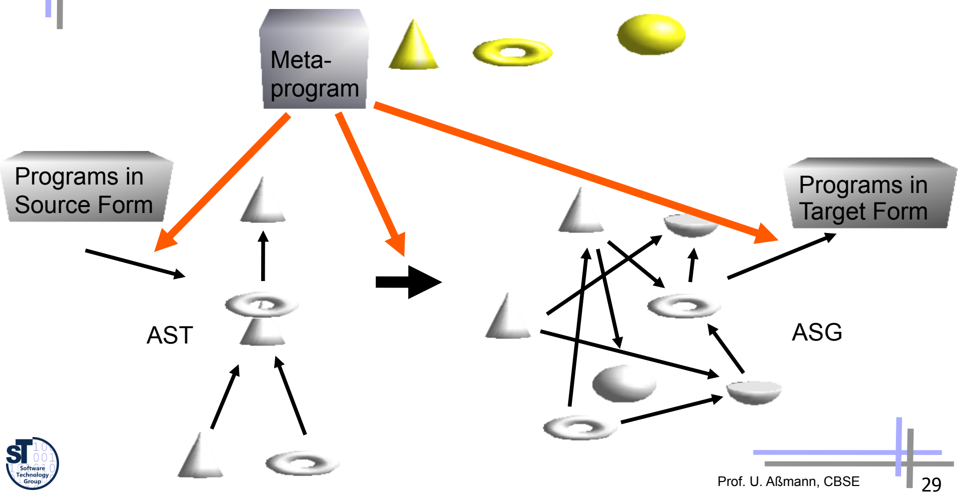
Staged Metalevel Architecture (Static Metaprogramming Architecture)



Compilers



Compilers Are Static Metaprograms



2.3 Metaobject Protocols (MOP)

Metaobject Protocol (MOP)

- A MOP is an reflective implementation of the methods of the metaclasses
  - It specifies an interpreter for the language, describing the semantics, i.e., the behavior of the language objects
  - in terms of the language itself.
- By changing the MOP *(MOP intercession),* the language semantics is changed
  - or adapted to a context.
  - If the MOP language is object-oriented, default implementations of metaclass methods can be overwritten by subclassing
  - and the semantics of the language is changed by subclassing

A Very Simple MOP

```
public class Class {
  Class(Attribute[] f, Method[] m) {
    fields = f; methods = m;
  }
  Attribute[] fields; Method[] methods;
}
public class Attribute {
  public String name; public Object value;
  Attribute (String n) { name = n; }
  public void enterAttribute() { }
  public void leaveAttribute() { }
  public void setAttribute(Object v) {
    enterAttribute();
    this.value = v;
    leaveAttribute();
  }
  public Object getAttribute() {
    Object returnValue;
    enterAttribute();
    returnValue = value;
    leaveAttribute();
    return returnValue;
  }
}
```

```
public class Method {
  public String name;
  public Statement[] statements;
  public Method(String n) { name = n; }
  public void enterMethod() { }
  public void leaveMethod() { }
  public Object execute {
    Object returnValue;
    enterMethod();
    for (int i = 0; i <= statements.length; i++) {
      statements[i].execute();
    }
    leaveMethod();
    return returnValue;
  }
}
public class Statement {
  public void execute() { ... }
}
```

Adapting a Metaclass in a MOP By Subclassing

```
public class TracingAttribute extends Attribute {
  public void enterAttribute() {
    System.out.println("Here I am, accessing attribute " + name);
  }
  public void leaveAttribute() {
    System.out.println("I am leaving attribute " + name +
                  ": value is " + value);
  }
}
```

```
Class Robot = new Class(new Attribute[]{ "WorkPiece piece1",   "WorkPiece piece2" },
  new Method[]{ "takeUp() { WorkPiece a = rotaryTable.place1; } "});
Class RotaryTable =  new Class(new TracingAttribute[]{ "WorkPiece place1",
     "WorkPiece place2" },  new Method[]{});
```

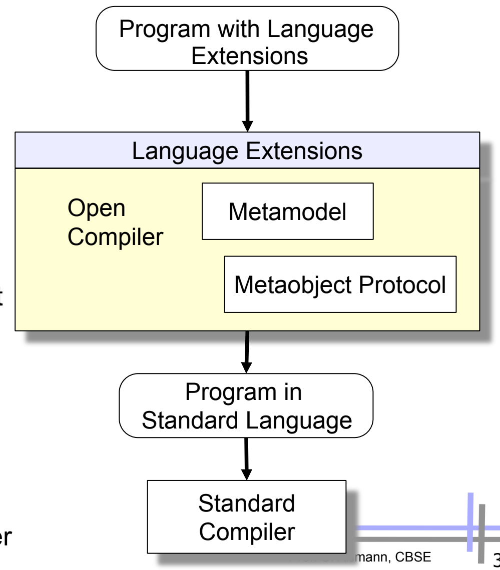
```
Here I am, accessing attribute place1
I am leaving attribute place1: value is WorkPiece #5
```


Adaptation of Components by MOP Adaptation

```
// Adapter is hidden in enterMethod
Method EventAdapterMethod extends Method {
  Object piece;

  public Object execute() {
    // event communication
    notifyRotaryTable();
    piece = listenToRotaryTable();

    super.execute();
    return piece;
  }
}
// Create a class Robot with the new semantics for takeUp()
Class Robot = new Class(new Attribute[]{ },
  new Method[]{ new EventAdapterMethod("takeUp") });
```


An Open Language with Static MOP

- .. has a static metalevel architecture (static metaprogramming architecture), with a *static MOP*
- ... offers its AST as metamodel for static metaprogramming
  - Users can write static metaprograms to adapt the language
  - Users can override default methods in the metamodel, changing the static language semantics or the behavior of the compiler



An Open Language

- ... can be used to adapt components at compile time
  - During system generation
  - Static adaptation of components
- Metaprograms are removed during system generation, no runtime overhead
  - Avoids the overhead of dynamic metaprogramming
- Open Java, Open C++

2.4 Metaobject Facilities (MOF)

Metaobject Facility (MOF)

- Rpt: A metalanguage is used to describe languages
  - Context-free structure (model trees or abstract syntax trees, AST)
  - Context-sensitive structure and constraints (model graphs or abstract syntax graphs, ASG)
  - Dynamic semantics (behavior)

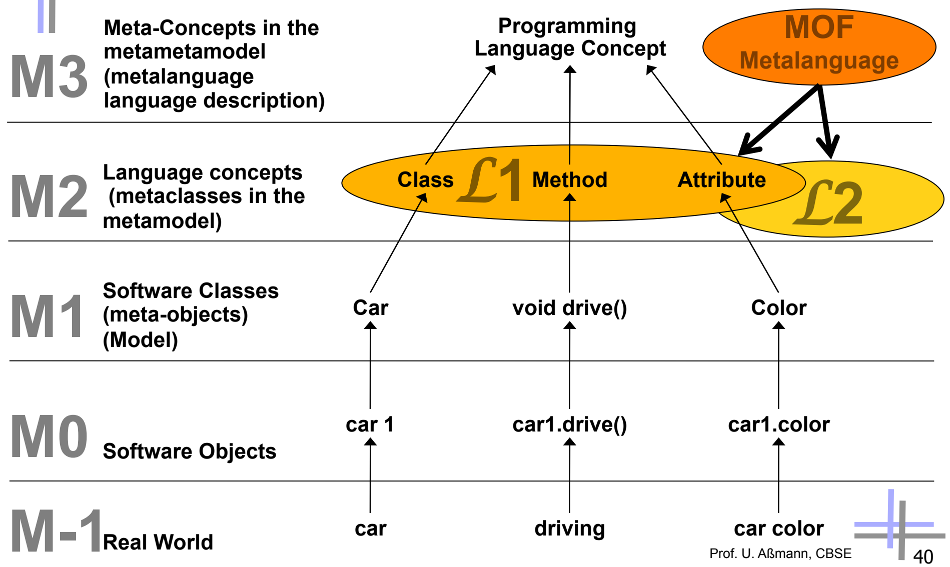
A **metaobject facility (MOF)** is a language specification language (metalanguage) to describe the *structure* of a language (context-free, context-sensitive).

- MOF is a metalanguage to to describe model graphs / ASG
- MOF provides the modeling concepts
  - Classes, relations, attributes; methods are lacking
  - Logic constraints (OCL) on the classes and their relations
  - Usually, a MOF does not describe an interpreter for the full-fledged language, but provides only a *structural description*

Metaobject Facility (MOF)

- A MOF is not a MOP
  - The MOF is generative
  - The MOP is interpretative
- The OMG-MOF (metaobject facility) was first standardized Nov. 97, available now in version 2.0 since Jan 2006

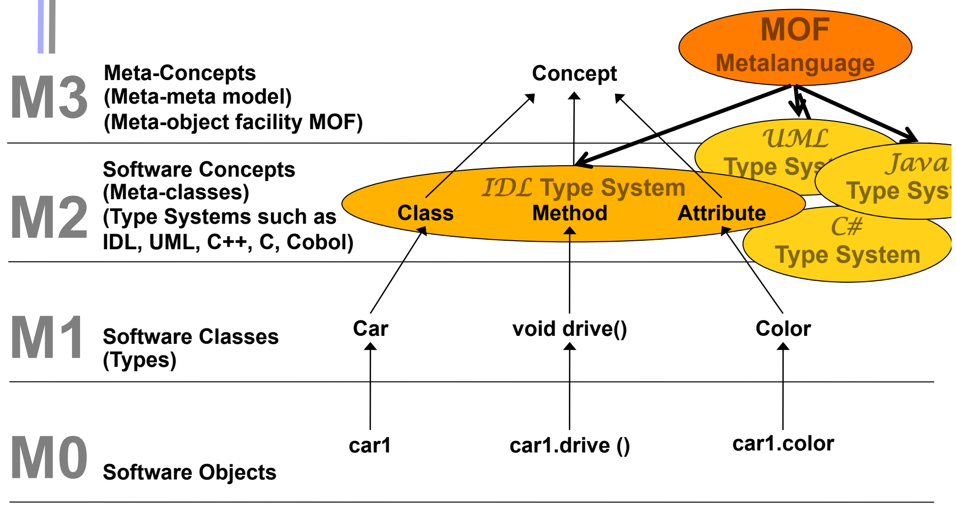
MOF Describes, Constrains, and Generates Structure of Languages on M2



MOF

- With MOF, context-sensitive structure of languages are described, constrained, and generated
  - Type systems
    - to navigate in data with unknown types
    - to generate data with unknown types
    - Describing IDL, the CORBA type system
    - Describing XML schema
  - Modelling languages (such as UML)
  - Relational schema language (common warehouse model, CWM)
  - Component models
  - Workflow languages
- From a language description in MOF,
  - Generative mappings (transformer, generator) from the metalanguage level (M3) to the language level (M2) can be generated
  - Also mappings from different languages on M2

Describing Type Systems with the MOF



Meta-meta-models describe general type systems!

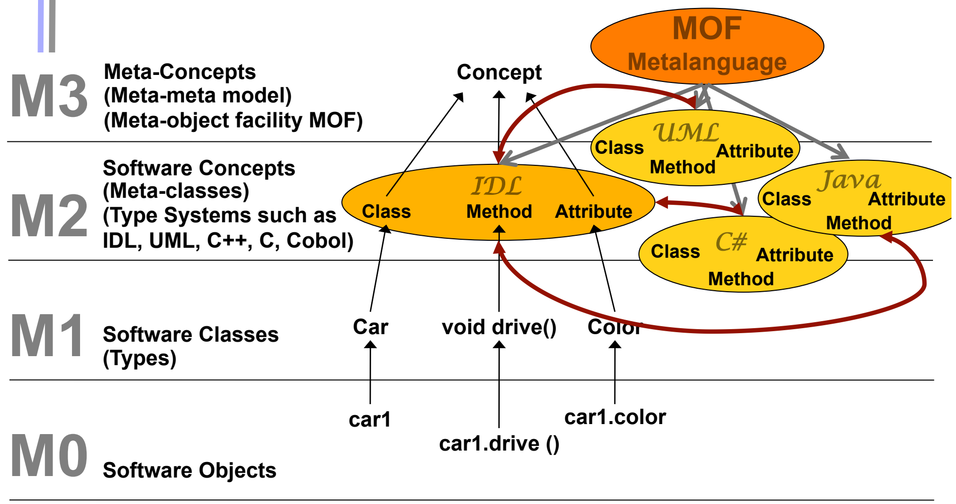
A Typical Application of MOF: Mapping Type Systems

- The type system of CORBA is a kind of "mediating type system" (least common denominator)
  - Maps to other language type systems (Java, C++, C#, etc)
  - For interoperability to components written in other languages, an interface description in IDL is required
- Problem: How to generate Java from IDL?
  - You would like to say (here comes the introspection):

```
for all c in classes do
        generate_class_start(c);
        for all a in c.attributes do
            generate_attribute(a);
        done;
        generate_class_end(c);
done;
```

- Other problems:
  - How to generate code for exchange between C++ and Java?
  - How to exchange data of OMT and UML-based CASE-tools?
  - How to bind other type systems as IDL into Corba (UML, ...)?

Mapping Type Systems in CORBA



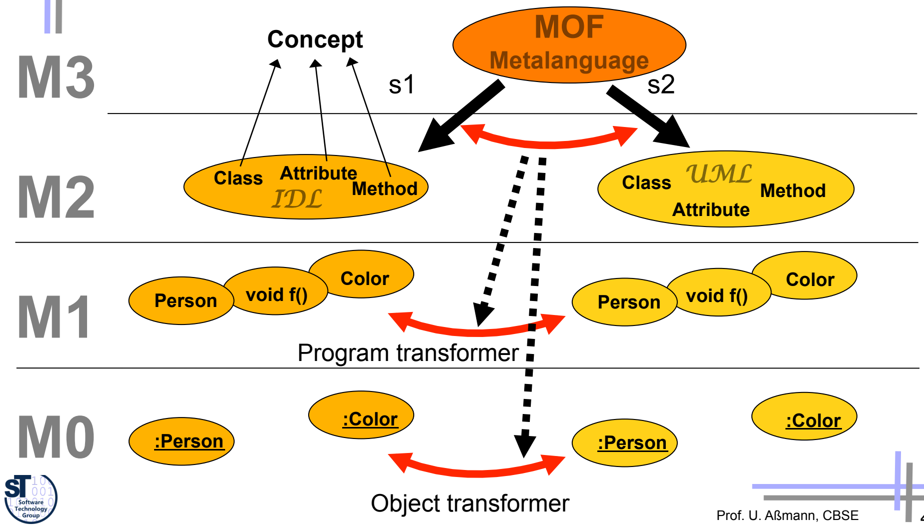
Meta-meta-models describe general type systems!

Automatic Data Transformation with the Metaobject Facility (MOF)

- Given:
  - 2 different language descriptions
  - An isomorphic mapping between them
- Produced helper functionality:
  - A transformer that transforms data in the languages
- Data fitting to MOF-described type systems can automatically be transformed into each other
  - The mapping is only an isomorphic function in the metametamodel
  - Exchange data between tools possible

Language Mappings for Program and Object Mappings

- Comparing the MOF language descriptions s1 and s2, transformers on classes and objects can be generated

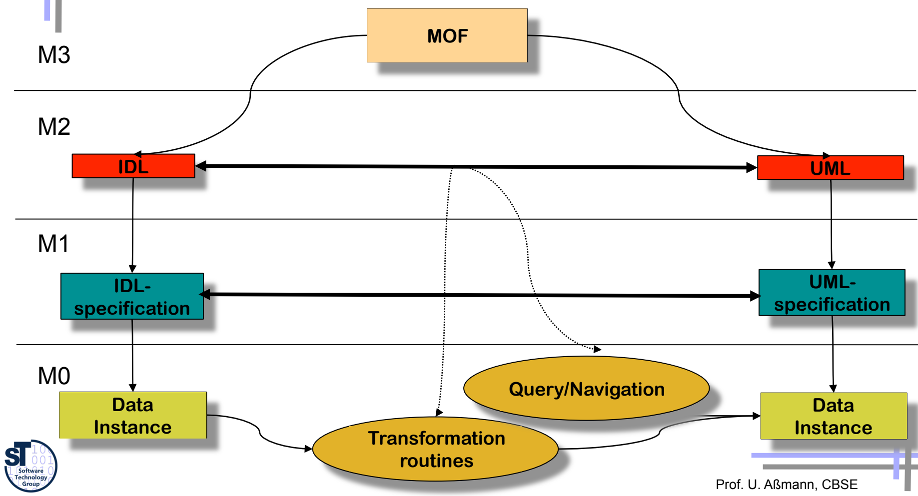


Reason: Similarities of Type Systems

- Metalevel hierarchies are similar for programming, specification, and modeling level
  - Since the MOF can be used to describe type systems there is hope to describe them all in a similar way
- These descriptions can be used to generate
  - Conversions
  - Mappings (transformations) of interfaces and data

The MOF as Smallest Common Denominator and "Mediator"

- From the mappings of the language-specific metamodels to the IDL metamodel, transformation, query, navigation routines can be generated



Bootstrap of MOF

- The MOF can be bootstrapped with the MOF
  - The structure and constraints of the MOF language can be described with itself
- IDL for the MOF can be generated
  - With this mechanism the MOF can be accessed as remote objects
  - MOF descriptions be exchanged
  - Code for foreign tools be generated from the MOF specifications
  - The MOF-IDL forms the interface for *metadata repositories (MDR)* http://mdr.netbeans.org
  - Engines in any IDL-mapped language can access an MDR, by using the IDL-generated glue code
  - Example: OCL Toolkit Dresden (which also supports EMF/Ecore besides of MDR)

Summary MOF

- The MOF describes the structure of a language
  - Type systems
  - Languages
  - itself
- Relations between type systems are supported
  - For interoperability between type systems and -repositories
  - Automatic generation of mappings on M2 and M1
- Reflection/introspection supported
- Application to workflows, data bases, groupware, business processes, data warehouses

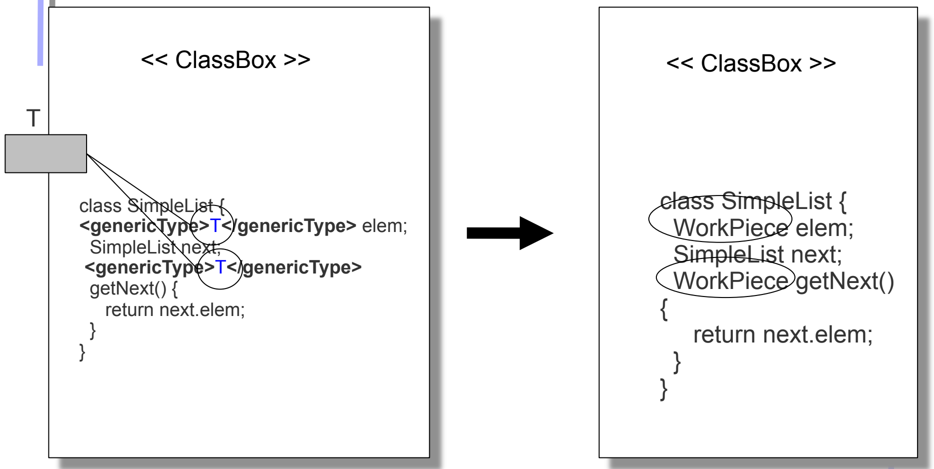
2.5 Asserting Embedded Metadata with Component Markup

.. A simple aid for introspection and reflection...

Markup Languages

- Markup languages convey more semantics for the artifact they markup
  - For a component, they describe metadata
  - XML, SGML are markup languages
- A markup can offer contents of the component for the external world, i.e., for composition
  - Remember: a component is a container
  - It can offer the content for introspection
  - Or even introcession
- A markup is stored together with the components, not separated

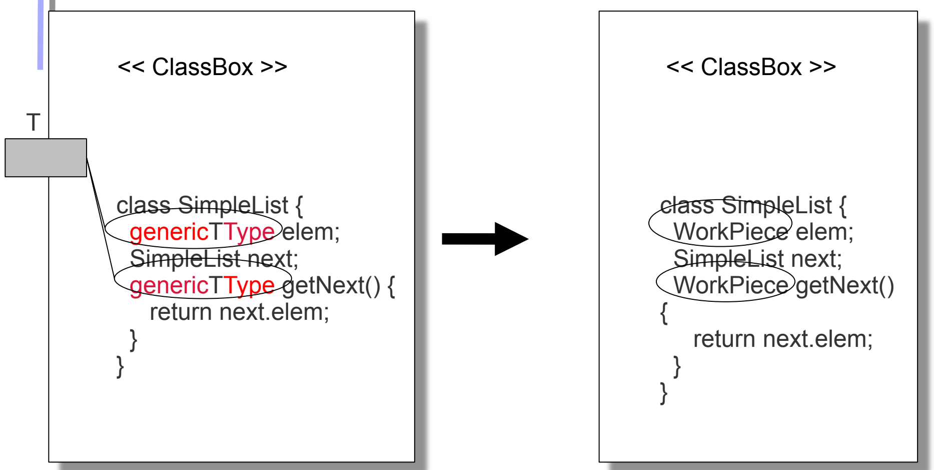
Example: Generic Types



Markup with Hungarian Notation

- **Hungarian notation** is a markup method that defines naming conventions for identifiers in languages
  - to convey more semantics for composition in a component system
  - but still, to be compatible with the syntax of the component language
  - so that standard tools can be used
- The composition environment can ask about the names in the interfaces of a component (introspection)
  - and can deduce more semantics

Generic Types with Hungarian Notation



Java Beans Naming Schemes use Hungarian Notation

- Property access
  - setField(Object value);
  - Object getField();
- Event firing
  - fire<Event>
  - register<Event>Listener
  - unregister<Event>Listener

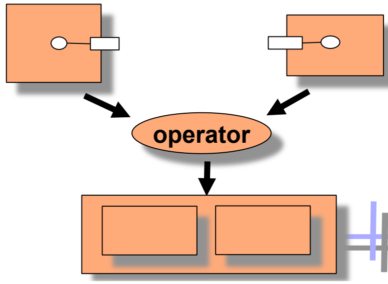
Markup and Metadata Attributes

Many languages support *metadata attributes*

- by Structured Comments
  - Javadoc tags
    - `@author @date @deprecated`
- Java 1.5 annotations and C# attributes are *metadata*
  - Java 1.5 annotations:
    - `@Override @Deprecated @SuppressWarnings`
  - C# /.NET attributes
    - `[author(Uwe Assmann)]`
    - `[date Feb 24]`
    - `[selfDefinedData(...)]`
  - User can define their own metadata attributes themselves
  - Metadata attributes are compiled to byte code and can be inspected by tools of an IDE, e.g., linkers, refactorers, loaders
- UML stereotypes and tagged values
  - <<Account>> { author="Uwe Assmann" }

Markup is Essential for Component Composition

- because it supports introspection and intercession
  - Components that are not marked-up cannot be composed
- Every component model has to introduce a strategy for component markup
- Insight: a component system that supports composition techniques must have some form of reflective architecture!
- Composition operators need to know where to compose
- Markup marks the variation points and extension points of components
- The composition operators introspect the components
- And compose



What Have We Learned?

- Metalanguages are important (M3 level)
  - Reflection is modification of oneself
  - Introspection is thinking about oneself, but not modifying
  - Metaprogramming is programming with metaobjects
  - There are several general types of reflective architectures
- A MOP can describe an interpreter for a language; the language is modified if the MOP is changed
  - A MOF specification describes the structure of a language
  - The CORBA MOF is a MOF for type systems mainly
- Component and composition systems are reflective architectures
  - Markup marks the variation and extension points of components
  - Composition introspects the markup
  - Composition can also use static metaprogramming or open languages

The End