An Example of the MOF: XMI for UML

- XMI - The Value of Interchange. Dr. Stephen A. Brodsky, IBM, Talk am 5.2. 1999, OMG XMI Briefing
- Overview of XMI. Sridhar Iyengar, Unisys Corporation, 5.2.99, OMG XMI Briefing
XMI - The Exchange Format

- XML: eXtensible Markup Language
- Contains metalanguages for DTD-definitions (Document Type Definition)
- or Xschema definitions
  - HTML, MathML, SpeechML, MusicML, VectorML, SVG (simple vector graphics), DocBook
- Standard for CORBA: XMI, a UML in XMI, MOF-based
  - eXtended Markup language Interchange format
- XMI = UML in XML, described by MOF
XMI

- **Purpose**
  - Neutral, open exchange format for UML in distributed environments

- **Advantage**
  - XMI generates a DTD for a type system
  - Based on the MOF ensures that the DTDs can be transformed to each other (interoperability)
  - Compatible with web standards and standard modeling languages
  - Supports transmission of difference-documents (diffs) and extensions of meta-models
XMI as Intermediate Representation

Design

Development Tools

Database Schema

XMI

Software Assets

Reports

Repository

6 Bridges of 6 suppliers

App1

App2

App3

App4

App5

App6

\( N \times N - N = 30 \text{ bridges} \)
S. Brodsky, OMG XMI Briefing, Feb 5, 1999
Overview XMI

XML exchange streams (modele)

- MOF: Metameta-model, Metadata Definitionen & Management
- UML: meta-model, Analysis & Design
- XML: Syntax

XML DTD (meta-models) (spezific per type system)

UML Instance

CWM Instance

MOF Instance

UML 1.1 DTD

CWM DTD

MOF 1.1 DTD
XMI as Intermediate Repräsentation

- CaseTool
  - CaseTool-Spez. in XML
  - Transformation by XML-Reader/Writer
- MOF
- UML
  - UML-Spec. in XML (page)
  - UML-Spec. (in UML)
  - Transformation by XML-Reader/Writer
- UML-DTD
  - Representation with Style Sheets in Browsers
- CaseTool-DTD
  - Query/Navigation by WebQL, the XML-QL
  - Transformation by XML-Reader/Writer

MOF

- CaseTool-Spez.
  - in XML

XML-QL Representation with Style Sheets in Browsers

Transformation by XML-Reader/Writer

Query/Navigation by WebQL, the XML-QL
For every type system a DTD is generated
For the MOF a DTD is generated

Additionally, XML is generated
<!-- Document prologue, etc.   -->
<Model xmi.id="a1">
  <name>Business</name>
  <visibility xmi.value="public"/>
  <ownedElement>
    <Class xmi.id="a7">
      <name>Customer</name>
      <feature>
        <Attribute>
          <name>id</name>
          <multiplicity>
            <XMI.field>1</XMI.field>
          </multiplicity>
          <type><DataType href="|a247"/></type> <!-- Custid -->
        </Attribute>
      </feature>
      <Operation>
        <name>update</name>
        <scope xmi.value="instance"/>
      </Operation>
    </Class>
  </ownedElement>
</Model>
XMI example for trader-type system with trader-DTD

```xml
<!DOCTYPE trader SYSTEM "trader.dtd">
<trader XMI.Id="trader_12">
    <ServiceType XMI.Id="Service_1">
        <BaseType.name> "Savings_Account" </BaseType.name>
        <propertyType XMI.Id="property_1">
            ...
        </propertyType>
    </ServiceType>
</trader>
```

**trader-DTD:**

```xml
<!ELEMENT trader ((ServiceType|Inherits)*)>
<!ATTLis trader XMI.Id ID #IMPLIED >
<!ELEMENT ServiceType (BaseType.name, ServiceType.interface_type, (propertyType)*)>
<!ATTLis ServiceType XMI.Id ID #IMPLIED> 
...```
XMI Summary

- XMI will revolutionize the exchange of data and meta data between distributed tools
- Closed systems bye bye!
- DCOM will be wiring standard, but probably Microsoft will invent the same under a different name and sell it as its idea (see UDDI, DCOM, etc.)
- Corba contains with MOF and XMI the more flexible concepts
- MOF and XMI support general extensible languages
  - Application specific languages, which are nevertheless interoperable
    - Design with application concepts; style I interoperability
History

- 1.1 March 14
Metamodelling and Metaprogramming

Part I
Basics
Mandatory Literature

- ISC, 2.2.5 Metamodelling
Literature

- R. Orfali, D. Harkey: Client/Server programming with Java and Corba. Wiley&Sons. easy to read.
An Introduction into Metalevels

“A system is about its domain. A reflective system is about itself”
Maes, 1988
Metadata

- **Meta**: means “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**
Different Types of Semantics and their Metalanguages (Description Languages)

- **Structure**
  - Described by a *context free grammar*
  - Does not regard context

- **Static Semantics**
  - Described by *context sensitive grammar (attribute grammar, denotational semantics, logic constraints)*
  - 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)*
  - Sets of runtime states or terms
Classes and Metaclasses

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

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

Classes in a software system

Metaclasses
Creating a Class from a Metaclass

- Using the constructor of the metaclass

```csharp
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, Introcession, Metaprogramming)

- Computation about the metamodel in the model is *reflection*
- 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
Reading 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)

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

Full Reflection: (Introcession)

for all c in self.classes do
    helpClass = makeClass(c.name+"help");
    for all a in c.attributes do
        helpClass.addAttribute(copyAttribute(a));
done;
    self.addClass(helpClass);
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

```java
enum { Singleton, Parameterizable } BaseFeature;
public class LanguageConcept {
    String name;
    BaseFeature singularity;
    LanguageConcept(String n, BaseFeature s) {
        name = n;
        singularity = s;
    }
    LanguageConcept Class = new LanguageConcept("Class", Singleton);
    LanguageConcept Attribute =
        new LanguageConcept("Attribute", Singleton);
    LanguageConcept Method =
        new LanguageConcept("Method", Parameterizable);
}
```

Language concepts (Metamodel)

Metalanguage concepts

Language description concepts (Metametamodel)
Made It Simple

- Level 1: objects
- Level 2: classes, types
- Level 3: language
- Level 4: metalanguage, language description language
Use of Metamodels and Metaprogramming

- To model, describe, introspect, and manipulate
- Workflow systems
- Databases
- Programming languages
- Component systems, such as CORBA
- Composition systems, such as Invasive Software Composition
- ... probably all systems...
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
- Reflection is thinking about oneself with the help of metadata
- Metaprogramming is programming with metaobjects
Reflective Architecture

Repository with Concepts/Types/Descriptions as Artefacts

Metaobjects

Meta-program

Reflection

Base Level

Metalevel

Repository with Objects as Artefacts
In a metalevel architecture, the metamodel is used for computations,

- but the metaprotocols execute either on the metalevel or on the base level.
- supports metaprogramming, but not full reflection

**Special variants:**
- Introspective architecture (no self modification)
- Staged metalevel architecture (metaprogram evaluation time is different from system runtime)
Metalevel Architecture

Metaobject

Meta-program

Base Level

Metalevel
Introspective Architectures

Metaobjects

Introspection

Metaobjects

Base Level

Metalevel
Static Metaprogramming Architecture

Base Level

Metaobjects

Metalevel

Static Time

Dynamic Time

Base Level

Static Time

Metaobjects

Meta-program
Compilers

Programs in Source Form

Parsing, Analysing

AST

Intermediate Representation

Programs in Target Form

Code Generation, Pretty Printing

ASG
Compilers Are Static Metaprograms

- Programs in Source Form
- AST
- Meta-program
- Programs in Target Form
- ASG

Compilers are static metaprograms, transforming programs from source form to target form.
Metaobject Protocols (MOP)
Metaobject Protocol

- A MOP is an 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, the language semantics is changed,
  - or adapted to a context.
- If the language is object-oriented, default implementations of
  - metaclass methods can be overwritten by subclassing,
  - and the semantics of the language is changed by subclassing.
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() { ... }
}

// More code...
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[]{});
Adaptation of Components

// Adapter is hidden in enterMethod
Method EventAdapterMethod extends Method {
    Object piece;
    public void enterMethod() {
        // event communication
        notifyRotaryTable();
        piece = listenToRotaryTable();
    }
    public Object execute() {
        enterMethod();
        return piece;
    }
}

// Create a class RotaryTable with the new semantics for takeUp()
Class RotaryTable = new Class(new Attribute[]{},
    new Method[]{ new EventAdapterMethod("takeUp") });
Open Languages

- Open Java, Open C++
- Employ static metaprogramming

Program with Language Extensions

Language Extensions

Open Compiler

Static Metaprograms

Metamodel

Metaobject Protocol

Program in Standard Language

Standard Language
Open Languages

- 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
- 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
Metaobject Facilities (MOF)
What is IDL (Interface Description Language)?

- The type system of CORBA
- Maps to many other language type systems (Java, C++, C#, etc)
- Is a kind of “mediating type system”, least common denominator...
- For interoperability to components written in other languages, an interface description in IDL is required
Problem: How to generate IDL from a Java application?
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;

Need a type system, that describes the Java type system
  With classes and attributes, methods

Other problems:
  How to generate Code for 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, ..)?
Metaobject Facility (MOF)

- Metadata can be used to
  - Get knowledge about unknown data formats, types
  - Navigate in unknown data
  - Generate unknown data
    - Generate type systems (e.g., IDL from programming languages)
    - Generate languages from metalanguage specifications

A Metaobject facility is a generative mapping (transformer, generator) from the metalanguage level (M3) to the language level (M2)
The MOF Generator

Level 4 - Meta-Concepts in the metameta model, the metalanguage language description

Level 3 - Language concepts (Metaclasses in the metamodel)

Level 2 - Software Classes (meta-objects) (Model)

Level 1 - Software Objects

Level 0 – Real World

The MOF Generator
Metaobject Facility (MOF)

- From different language descriptions, different languages are generated
  - Type systems
  - Modelling languages (such as UML)
  - Component models
  - Workflow languages
- A MOF cannot generate a full-fledged language
- A MOF is not a MOP
  - The MOF is generative
  - The MOP is interpretative
Meta Levels in Corba Type Systems

Level 4 - Meta-Concepts
(Meta-meta model)
(Meta-object facility MOF)

Level 3 - Software Concepts (Meta-classes)
(Type Systems such as IDL, UML, C++, C, Cobol)

Level 2 - Software Classes (Types)

Level 1 - Software Objects

Meta-meta-models describe general type systems!
Web Services and Workflows are Subsumed under the MOF

- It is possible to specify workflow languages with the MOF.
- BPEL and the underlying XML dialects (WSDL, SOAP) are hot candidates
- The MOF will provide good interoperability
Meta Levels in Workflow Systems
And Web Services (BPEL)

Level 4 - Meta-Concepts
(Metameta classes)
(Metameta model)

Level 3 - Workflow
Concepts (Meta-classes)
(Meta-model)

Level 2 - Workflow
Software Classes
(Meta-objects)
(Model)

Level 1 - Workflow
Software Objects

- Client
- Order
- Data
- Function (Web Service)
- Ressource
- Materia
- Item
- fred
- nail
- orderForGoods
The OMG-MOF (metaobject facility) is a MOF, i.e., a metalanguage, describing type systems:
- Describing IDL, the CORBA type system
- Describing the UML metamodel
- Describing XML schema
- Standardized Nov. 97

It is not a full metalanguage, but only contains:
- Classes, relations, attributes
- OCL specifications to express constraints on the classes and their relations
- A MOP cannot be specified in the MOF (methods are lacking in the MOF)
Automatic Data Transformation with the Metaobject Facility (MOF)

- **Given:**
  - 2 different language descriptions
  - An isomorphic mapping between them

- **Produced:**
  - 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
Meta Levels in Corba Type Systems

Level 4 - Meta-Concepts
(Meta-meta model)
(Meta-object facility MOF)

Level 3 - Software Concepts (Meta-classes)
(Type Systems such as IDL, C++, C, Cobol)

Level 2 - Software Classes
(meta-objects)
(Types)

Level 1 - Software Objects

- object1
- object1.print()
- object1.color

Meta-meta-models describe general type systems!
Isomorphic Language Mappings

- **Concept**: Person
- **Language Description**: Class, Method, Attribute
  - **IDL**: Class, Method, Attribute
  - **UML**: Class, Method, Attribute

Transformer
Example Trader in MOF MODL

Package trader {
    class property_type {
        attribute string name;
        attribute TypeCode value_type;
    }
    class service_type {
        attribute string name;
        attribute string interface_type;
    }
    association has {
        role single service_type service;
        role set [0..*] of property_type property;
    }
    association inherits {
        role set [0..*] of service_type base;
        role set [0..*] of service_type derived;
    }
}

- Different MOF languages can be realized
  - Here MODL, an initial language
- Mapping of MODL (MOF Definition language) to IDL:
  - attributes to set/get-functions
  - associations to Link-classes and Link-Sequence-classes
  - MODL generates a class, containing special methods, by with to get all classes (method all_of_type)
The MOF can be also described in UML

Mapping of MOF to IDL:
- attributes to set/get-functions
- associations to Link-classes and Link-Sequence-classes
- MODL generates a class, containing special methods, by with to get all classes (method all_of_type)
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”

- MOF
  - IDL
  - UML
  - Query/Navigation
  - Transformation routines
  - IDL-specification
  - UML-specification
  - data-Instance

(IDL to MOF to UML)
(IDL to MOF to Query/Navigation)
(MOF to UML to Query/Navigation)
(MOF to Transformation routines)
_Transformation routines to data-Instance_
Bootstrap

- The MOF can be bootstrapped with the MOF
  - It 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
Summary MOF

- The MOF describes general type systems
- New type systems can be added, composed and extended from old ones
- Relations between type systems are supported
- For interoperability between type systems and repositories
- Automatic generation of IDL
- Reflection/introspection supported
- Application to workflows data bases, groupware, business processes, data warehouses (Common Warehouse Model, CWM)
Component Markup

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

- Convey more semantics for the artifact they markup
- XML, SGML are markup languages
- Remember: a component is a container
- A markup can offer contents of the component for the external world, i.e., for composition
  - It can offer the content for introspection
  - Or even introcession
Example: Generic Types

```
<< ClassBox >>

class SimpleList {
    <genericType>T</genericType> elem;
    SimpleList next;
    <genericType>T</genericType> getNext() {
        return next.elem;
    }
}

<< ClassBox >>

class SimpleList {
    WorkPiece elem;
    SimpleList next;
    WorkPiece getNext() {
        return next.elem;
    }
}
```
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

class SimpleList {
    generic T Type elem;
    SimpleList next;
    generic T Type getNext() {
        return next.elem;
    }
}

class SimpleList {
    WorkPiece elem;
    SimpleList next;
    WorkPiece getNext() {
        return next.elem;
    }
}
Java Beans Naming Schemes

- Property access
  - setField(Object value);
  - Object getField();

- Event firing
  - fire<Event>
  - register<Event>Listener
  - unregister<Event>Listener
Markup by Comments

- Javadoc tags
  - @author
  - @date
  - @obsolete
- Java 1.5 attributes
- C# attributes
  - //@author
  - //@date
  - //selfDefinedData
- C# / .NET attributes
  - [author(Uwe Assmann)]
  - [date Feb 24]
  - [selfDefinedData(...)]
Markup is Essential for Component Composition

- because it supports introspection and introcession
- 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 be a 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

operator
What Have We Learned?

- Metalanguages are important (M3 level)
- Reflection is modification of oneself
- Introspection is thinking about oneself, but not modifying
- 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 is a generator for 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