1. Introduction to Metalevels

"A system is about its domain.
A reflective system is about itself"

P. Maes, ACM Proc. OOPSLA 1987

Example: Different Types of Program Semantics and their Metalanguages (Description Languages)

- **Syntactic structure**
  - Described by a context free grammar
  - Does not consider context

- **Static Semantics**
  - Described by context sensitive grammar
    - (or 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

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

[SC] 2.2.5 Metamodeling: [CS] 10.7, 14.4.1

Metamodeling and Metaprogramming

1. Introduction to metalevels
2. Metalevel architectures
3. Meta-object protocol (MOP)
4. Meta-object facility (MOF)
5. Component markup
Metalevels in Programming Languages

Level 4 - Meta-Concepts in the metamodel, the metalanguage (language description)

Level 3 - Language concepts (Meta-classes in the metamodel)

Level 2 - Software Classes (Meta-objects)

Class Method Attribute

Programming Language Concept

Level 1 - Software Objects

car 1

cart1

cart1.drive() car1.color

Level 0 - Real World

car

driving
car.color

Creating a Class from a Metaclass

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

class WorkPiece {
    Object belongsTo;
}

class RotaryTable {
    WorkPiece place1, place2;
}

class Robot {
    WorkPiece piece1, piece2;
}

class Press {
    WorkPiece place;
}

class ConveyorBelt {
    WorkPiece[] pieces;
}

Metaclass

Classes in a software system

Concepts of a metalevel can be represented at the base level. This is called reification.

Examples:
- Java Reflection API [Szyperski 14.4.1]
- UML metamodel (MOF)

Reflection (Self-Modification, Metaprogramming)

- Computation about the metamodel in the model is reflection.
- The application can look at its own skeleton (metadata) and may even change it
  - Allocating new classes, methods, fields
  - Removing classes, methods, fields

Remark: In the literature, "reflection" was originally introduced to denote "computing about the own program" [Mees'87] but has also been used in the sense of "computing about other programs" (e.g., components).
Introspection

- Read-only reflection is called **introspection**
  - The component can look up the metadata 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

**Metadata**
- Data
- Code
- Information

Introcession

- Read and Write reflection is called **introcession**
  - The component can look up the metadata of itself or another component and may change it
  - Typical application: dynamic adaptation of parts of own program
    - Classes, methods, attributes, types

Reflection Example

**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 that uses this information about itself in its normal course of execution.

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);
```
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
- Compiler run-time systems for parallel computers
- Component systems, such as CORBA
- Composition systems, such as Invasive Software Composition
- Modeling systems, such as UML or Modelica
- ... probably all systems ...

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
  - Reflection is thinking about oneself with the help of metadata
  - Metaprogramming is programming with metaobjects
In a **metalevel architecture**, the metamodel is used for computations, but the metaprograms execute either on the metalevel or on the base level.

- Special variants:
  - **Introspective architecture** (no self-modification)
    - Example: Java Reflection API
  - **Staged metalevel architecture** (metaprogram evaluation time is different from system runtime)
    - Example: C++ templates (generics), expanded at compile time
Static Metaprogramming Architecture

Compilers

Compilers Are Static Metaprograms

3. Metaobject Protocols (MOP)

From structural to behavioral reflection
Metaobject Protocol

A metaobject protocol (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 thereby changing the semantics of the language.

A Very Simple MOP

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

```java
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;
    }
}
```

```java
public class Statement {
    public void execute() {
    }
}
```

```java
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;
    }
}
```

An Adapted MOP

```java
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);
    }
}
```

```java
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
```

Open Languages

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

<table>
<thead>
<tr>
<th>Open Languages</th>
<th>Program with Language Extensions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Language Extensions</td>
</tr>
<tr>
<td></td>
<td>Open Compiler</td>
</tr>
<tr>
<td></td>
<td>Static Metaprograms</td>
</tr>
<tr>
<td></td>
<td>Matamodel</td>
</tr>
<tr>
<td></td>
<td>Metaprogrammed Protocol</td>
</tr>
<tr>
<td></td>
<td>Standard Language</td>
</tr>
<tr>
<td></td>
<td>Program in Standard Language</td>
</tr>
</tbody>
</table>

An Open Language

- ... offers its own metamodel for static metaprogramming
  - Its schema (e.g., structure of AST) is made accessible as an abstract data type
  - 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

Example: Generating IDL specifications

- 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

4. Metaobject Facilities (MOF)

- 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
- Some other problems:
  - How to generate code for exchange between C++ and Java?
  - How to bind other type systems than 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 (MOF) is a generative mapping (transformer, generator) from the metalanguage level (Level 4) to the language level (Level 3).

MOF: Example

- The MOF for the CORBA meta-metamodel contains a type system for type systems:
  - Entities
  - Relationships
  - Packages
  - Exceptions

- Can describe every type system of a programming or modeling language
- MOF concepts must be mapped to types of a specific type system
- From these mappings, code can be generated that provides services for that type system, e.g., code that navigates in object graphs.

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 (car 1)
Level 0 - Real World (car driving)

MOF: Example

- From different language descriptions, different (parts of) languages are generated
  - Type systems
  - Modeling 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 1- Software Objects
- object1
- object1.color
- object1.print()

Level 2- Software Classes
- Person
- void
- Color

Level 3- Software concepts
- IDL, UML, C++, C, Cobol

Level 4- Meta-Concepts
- Meta-object facility MOF

Metaobject Facility MOF in CORBA
- 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. 1997

- 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 metamodel
  - Exchange data between tools possible
Isomorphic Language Mappings

**Concept**

<table>
<thead>
<tr>
<th>Language Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDL Class Method Attribute</td>
</tr>
<tr>
<td>Class Method Attribute</td>
</tr>
<tr>
<td>Transformer</td>
</tr>
</tbody>
</table>

**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
- Language extensions, e.g., for extending UML
- Reflection/introspection supported
- Application to workflows, databases, groupware, business processes, data warehouses (Common Warehouse Model, CWM)

Reason: Similarities of Type Systems

- Metalevel hierarchies are similar for programming, specification, and modeling levels
- 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

5. Component Markup

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

- Convey more semantics for the artefact they markup
- HTML, 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 introspection

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

```java
<< ClassBox >>
class SimpleList {
    generic TType elem;
    SimpleList next;
    generic TType 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`

Metainformation for JavaBeans is identified by markup in the form of Hungarian Notation. This metainformation is needed, e.g., by the JavaBeans Assembly tools to find out which classes are beans and what properties and events they have.
### Markup by Comments

- **Javadoc tags**
  - `@author`
  - `@date`
  - `@obsolete`

- **Java 1.5 attributes**
  - `// @author`
  - `// @date`
  - `//selfDefinedData`

- **C# attributes**
  - `@author`
  - `@date`
  - `@obsolete`

- **C# / .NET attributes**
  - `[author(Uwe Assmann)]`
  - `[date Feb 24]`
  - `[selfDefinedData(...)]`

### Markup is Essential for Component Composition

- because it identifies metadata, which in turn 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!

### What Have We Learned?

- **Reflection** is reasoning and modification of oneself or others
- **Introspection** is thinking about oneself, but not modifying
- **Metaprogramming** is programming with meta-objects
- System has a **reflective architecture** if meta- and base level use the same specification or programming language
- System has a **metalevel architecture** if it only supports metaprogramming at meta-level (not at the base level)
- 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 (part of) a language
    - The CORBA MOF is a MOF for type systems mainly
- **Metamodeling**, e.g., MOF for UML / Corba IDL / ...
- Some well-known examples of metaprogramming:
  - Static metaprogramming at base level
    - C++ templates, AOP
  - Static metaprogramming at meta level
    - e.g., Compiler analysis / transformations
  - Dynamic metaprogramming at base level
    - e.g., Java Reflection
- Component and composition systems are **reflective architectures**
  - Markup marks the variation and extension points of components
  - Composition introspects the markup