11. Metadata, -modelling, and -programming

Prof. Dr. Uwe Aßmann
Technische Universität Dresden
Institut für Software- und Multimediatechnik
http://st.inf.tu-dresden.de/teaching/cbse
13.04.2017

Lecturer: Dr. Sebastian Götz

1. Searching and finding components
2. Metalevels and the metapyramid
3. Metaobject architectures
4. Metaobject protocols (MOP)
5. Metaobject facilities (MOF)
6. Metadata as component markup
Mandatory Literature

- ISC, 2.2.5 Metamodelling
- OMG MOF 2.0 Specification
  http://www.omg.org/spec/MOF/2.0/
  http://doi.acm.org/10.1145/643120.643124
11.1. Searching and Finding Components in Repositories

It should be as easy to find good quality reusable software assets as it is to find a book on the internet
Component Repositories

- Components must be stored in component repositories with metadata (markup, attributes) to find them again
- Descriptions (Metadata)
  - **Attributes**: Keywords, Author data
  - **Usage protocols** (behavioral specifications)
    - (Protocol) State machines record the sequence of calls to the component
    - Sequence diagrams record parallel interaction sequences of the component
    - Contracts (pre/post/invariants) specify conditions on the state before, after and during the calls
- Examples of Component Repositories
  - CORBA
    - implementation registry
    - interface registry
  - COM+ registry
  - Commercial Component Stores [www.componentsource.com](http://www.componentsource.com)
  - Debian Linux Component System (apt, dpkg)
  - CTAN TeX Archive
Why Searching Components?

- A public component repository is called a **market**, managed by a **trader (broker)**
  - Distributing or selling components
  - Companies can register components at the trader
  - Customers can search components in markets and buy or rent them
- Searching for functionality (interface, contract, protocol)
  - Reuse instead of build
  - Searching for components to replace own ones
  - Semantic substituability should be ensured
- Searching for quality features
  - Performance, energy consumption, reliability
11.2. An Introduction to Metalevels

“A system is about its domain. A reflective system is about itself.”

Pattie Maes, 1988
**Metadata**

- **Meta**: Greek for “describing”
- **Metadata**: Describing data (sometimes: self-describing data). The type system is called metamodel (i.e., a model describing a model).
- **Metalevel**: The elements of the meta-level (the meta-objects) describe the objects on the base level.
- **Metamodelling**: Description of the model elements/concepts in the metamodel.
- **Metalanguage**: A description language for languages.

---

**Diagram**

![Metadata Diagram]
Metalevels in Programming Languages (The Meta-Pyramid)

Component-Based Software Engineering (CBSE)

**M3**
Modeling Level 3: Conceptual level
A metametamodel is a metalanguage

**M2**
Modeling Level 2: Language
A metamodel is a language specification

**M1**
Modeling Level 1: Software Classes (meta-objects) (Model)
Class:
- Car
- void drive()
- Color

**M0**
Modeling Level 0: Software Objects
- car1
- car1.drive()
- car1.color

**M-1**
Real World
- car
- driving
- car color

Metalanguage concepts
Modelling concepts (Metametaclasses in the metametamodel)

Language concepts (Metaclasses in the metamodel)

Application concepts

World concepts
DSL and CL

- Domain-specific languages (DSL) form extensions on M2
- Composition languages (CL) also
- Language engineering means to develop M2 models (metamodels) using M3 language
We write metaclasses with dashed lines, metametaclasses with dotted lines.
Metaclasses are *schemata* for classes, i.e., describe what is in a class.

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

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

```java
public class Attribute {
    Object type;
    Object value;
}
```

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

```java
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

```java
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[]{ });
```

Diagram:
- Robot
- Class
- Press
- RotaryTable
- WorkPiece
- ConveyorBelt

<<instance-of>>
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 (MOP)
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
- Introspection is very important in component supermarkets (finding components)
Component-Based Software Engineering (CBSE)

Reading Reflection (Introspection)

- Used for generating something based on metadata information

```java
Component component = .. get from market ..
for all cl in component.classes do
    generate_for_class_start(cl);

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

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

    generate_for_class_end(cl);
done;
```
Full Reflection (Run-Time Code Generation)

Generating code, interpreting, or loading it

```python
for all c in self.classes do
    helperClass = makeClass(c.name+"Helper");
    for all a in c.attributes do
        helperClass.addAttribute(copyAttribute(a));
    end
    self.loadClass(helperClass);
    self.addClass(helperClass);
end
```

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

Patti Maes
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

- Modeling Level M-1: real-world objects
- Modeling Level M0: objects in the running program
- Modeling Level M1: programs, classes, types
- Modeling Level M2: language
- Modeling Level M3: metalanguage, language description language
11.3. 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

Repository with Concepts/Types/Descriptions as Artefacts

Metaobjects

Reflective Architecture

Meta-
program

Metalevel

Reflection

Base Level

Repository with Objects as Artefacts
Examples

► 24/7 systems with total availability
  ■ Dynamic update of new versions of classes
  ■ Telecommunication systems
  ■ 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
Introspective Architectures

Component-Based Software Engineering (CBSE)
Staged Metalevel Architecture (Static Metaprogramming Architecture)
Compilers

Component-Based Software Engineering (CBSE)

- Parsing, Analysing
- Programs in Source Form
- AST
- Intermediate Representation
- Programs in Target Form
- Code Generation, Pretty Printing
- ASG
Compilers Are Static Metaparograms

Component-Based Software Engineering (CBSE)
11.4 Metaobject Protocols (MOP)
Metaobject Protocol (MOP)

By changing the MOP (*MOP intercession*), the language semantic 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
- By changing the MOP of a component from a component market, the component can be adapted to the reuse context

A **meta-object protocol (MOP)** is a reflective implementation of the methods of the metaclasses (*interpreter* for the language) describing the semantics, i.e., the behavior of the language objects in terms of the language itself.
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 Meta-class in a MOP By Subclassing

```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);
    }
}
```

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
An Open Language has a Static MOP

- An **Open Language** 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 from a market at compile time
  - During reuse of the component in system generation
  - Static adaptation of components
- Metaprograms are removed during system generation, no runtime overhead
  - Avoids the overhead of dynamic metaprogramming
- Ex.: Open Java, Open C++
11.5 Metaobject Facility (MOF)

A structural metalanguage for graphs
A **metaobject facility (MOF)** is a language specification language (*metalanguage*) to describe the context-free structure and context-sensitive *structure* of a language and to check the wellformedness of models. Dynamic semantics (interpretation) is omitted.
Metaobject Facility (MOF)

- MOF (metaobject facility) of OMG is a metalanguage to describe the structure of modelling languages, and finally the structure of models as abstract syntax graphs (ASG)
  - MOF was first standardized Nov. 97, available now in version 2.0 since Jan 2006
- MOF is a minimal UML class diagram like language
  - MOF provides the modelling concepts: class, inheritance, relation, attribute, signature, package; but, e.g., method bodies are lacking
  - Constraints (in OCL) on the classes and their relations
- A MOF is not a MOP
  - The MOP is interpretative
  - A MOF specification does not describe an interpreter for the full-fledged language, but provides only a structural description
MOF Describes, Constrains, and Generates Structure of Languages on M2

M3: Meta-Concepts in the metametamodel (metalanguage language description)

M2: Language concepts (metaclasses in the metamodel)

M1: Software Classes (metaobjects) (Model)

M0: Software Objects

M-1: Real World

MOF Metalanguage

Language concepts (metaclasses in the metamodel)
A MOF specification (a MOF metamodel) is a typed attributed graph, containing:

- The concepts of a language as metaclasses
- Their relationships as associations between metaclasses
- Their constraints

With MOF, the context-sensitive structure of languages is 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**
Describing Type Systems with the MOF

Meta-concepts (Meta-meta model) (Meta-object facility MOF)

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

Software Classes (Types)

Software Objects

Meta-meta-models describe general type systems!
A Typical Application of MOF: Mapping Type Systems with a Language Mapping

The type system of CORBA-IDL 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 (by introspection):
  ```java
  for all c in classes_in_IDL_spec do
    generate_class_start_in_Java(c);
    for all a in c.attributes do
      generate_attribute_in_Java(a);
    done;
    generate_class_end_in_Java(c);
  done;
  ```

Other problems:
- How to generate code for exchange between C++ and Java?
- How to bind other type systems as IDL into Corba (UML, ...)?
Mapping Type Systems in CORBA

Meta-meta-models are used to describe general type systems

M3 Meta-Concepts (Meta-meta model) (Meta-object facility MOF)

M2 Software Concepts (Meta-classes) (Type Systems such as IDL, UML, C++, C, Cobol)

M1 Software Classes (Types)

M0 Software Objects
Automatic Data Transformation with the Metaobject Facility (MOF)

- From two MOF metamodels, **transformation bridges** are generated
  - And an isomorphic mapping between them
- Transformer functionality can be generated
  - 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
- Code looks like (similarly for all mapped languages):

```plaintext
for all c in classes in Java_spec do
    generate_class_mapper_from_Java_To_IDL(c);
    for all a in c.attributes do
        generate_attribute_mapper_from_Java_To_IDL(a);
    done;
    generate_class_end_mapper_from_Java_To_IDL(c);
done;

for all c in classes in IDL_spec do
    generate_class_mapper_from_IDL_to_C++(c);
    for all a in c.attributes do
        generate_attribute_mapper_from_IDL_to_C++ (a);
    done;
    generate_class_end_mapper_from_IDL_to_C++ (c);
done;
```
Comparing the MOF metamodels s1 and s2 with a language mapping l, transformers on classes and objects can be generated.
The MOF as Smallest Common Denominator and “Mediator” between Type Systems

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

- MOF is specified in itself (self-describing, lifted metamodel)
  - The structure, relations and constraints of the MOF language can be described with itself

- The MOF can be bootstrapped with the MOF

- IDL for the MOF can be generated
  - With this mechanism the MOF can be accessed as remote objects from other languages
  - MOF descriptions can 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](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
11.6 Asserting Embedded Metadata with Component Markup

.. A simple aid for introspection and reflection...
Example: Generic Types with XML Markup

Component-Based Software Engineering (CBSE)

<< ClassTemplate >>

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

<< ClassTemplate >>

class SimpleList {
    WorkPiece elem;
    SimpleList next;
    WorkPiece getNext() {
        return next.elem;
    }
}
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
Embedded Markup and Style Sheets

- Markup can be defined as *embedded* or by *style sheets*
  - Embedded markup marks (types) a part of a component in-line
    - The part may be required or provided
  - Style sheets mark (type) a part of a component off-line
    - with a matching language that filters the document contents
    - with addressing that points into the component
    - positions
    - implicit hook names
    - address expressions on compound components
- Some component languages allow for defining embedded markup
  - latex (new environments and commands)
  - languages with comments (comment markup)
- Style sheets can refer to embedded markup
- Both can be mixed
Markup with Hungarian Notation

- **Hungarian notation** is an embedded 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

- Hungarian notation has the advantage, that the syntactic tools of the base language work for the generic components, too.

```java
<< ClassTemplate >>

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

<< ClassTemplate >>

class SimpleList {
    WorkPiece elem;
    SimpleList next;
    WorkPiece getNext() {
        return next.elem;
    }
}
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 @entity @invoke-around

Java annotations and C# attributes are *metadata*

- Java 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