1. Introduction to metalevels
2. Different Ways of Metaprogramming
3. UML Metamodel and MOF
4. Component markup

Metadata

- **Meta**: means "describing"
- **Metadata**: describing data (sometimes: self-describing data).
  - The language (esp., type system) for specifying metadata 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

Classes and Metaclasses

```java
public class WorkPiece { Object belongsTo; }
class RotaryTable { WorkPiece place1, place2; }
class Robot { WorkPiece piece1, piece2; }
class Conveyor { WorkPiece pieces; }

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

public class Attribute {
}
public class Method {
}
```

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)

- **Reflection** is computation about the metamodel in the base model.
  - The application can look at its own skeleton (metadata) and may even change it
    - Allocating new classes, methods, fields
    - Removing classes, methods, fields
  - Enabled by reification of meta-objects at base level (e.g., as API)

Remark: In the literature, "reflection" was originally introduced to denote "computation about the own program" [Maes'87], but has also been used in the sense of "computing about other programs" (e.g., components).
Example: Creating a Class from a Metaclass

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

```java
public class Attribute {
}
```

```java
public class Method {
}
```

```java
class WorkPiece {
    Object belongsTo;
}
```

```java
class RotaryTable {
    WorkPiece place1, place2;
}
```

```java
class Robot {
    WorkPiece piece1, piece2;
}
```

```java
class ConveyorBelt {
    WorkPiece[] pieces;
}
```

---

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 for late (run-time) binding

---

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

```java
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;
```

---

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;
    }
}
```

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

---

Made It Simple

- Level 0: objects
- Level 1: classes, types
- Level 2: language elements
- Level 3: metalanguage, language description language
2. Different Ways of Metaprogramming

- meta-level vs. base level
- static vs. dynamic

Metaprograms are programs that compute about programs.

### Metaprograms can run at base level or at meta level

**Metaprogram execution at the meta-level:**
- Metaprogram is separate from base-level program
- Direct control of the metadata as metaprogram data structures
- Expression operators are defined directly on the metabjects
- Example: Compiler, program analyzer, program transformer
  - Program metadata = the internal program representation
    - has classes to create objects describing base program classes, functions, statements, variables, constants, types etc.

**Metaprogram execution at the base level:**
- Metaprogram/-code embedded into the base-level program
- All expressions etc. evaluated at base level
- Access to metadata only via special API, e.g. Java Reflection

### Static vs. Dynamic Metaprogramming

Recall: Metaprograms are programs that compute about programs.

- **Static metaprograms**
  - Execute before runtime
  - Metainformation removed before execution – no runtime overhead
  - Examples: Program generators, compilers, static analyzers

- **Dynamic metaprograms**
  - Execute at runtime
  - Metadata stored and accessible during runtime
  - Examples:
    - Programs using reflection (Introspection, Introsession);
    - Interpreters, debuggers
### Static Metaprogramming

- **Metaprogram** and metaobjects exist only at compile time. No run-time overhead.

### Example: Static Metaprogramming (1)

- **Example**: Static Metaprogramming (1)
  - **C++ templates**
  - **Example**: generic type definition
  - *(Meta)Information about generic type removed after compiling!*

### Example: Static Metaprogramming (2)

- **C++ templates**
  - Example: generic type definition
  - *(Meta)Information about generic type removed after compiling!*

### Compilers Are Static Metaprograms

- **Compilers** are Static Metaprograms
  - **Meta-**
  - **program** (Level 2)

### Compilers are Static Metaprograms

- **Compilers** are Static Metaprograms
  - **Example**: Static Metaprogramming (2)
  - **C++ templates**
    - **Example**: generic type definition
    - *(Meta)Information about generic type removed after compiling!*

### Dynamic Metaprogramming

- **Repository** with Concepts/Types/Descriptions as Artefacts
  - **Metaobjects**
  - **Reflection**

---

---
Summary: Ways of Metaprogramming

<table>
<thead>
<tr>
<th>Metaprogram runs at:</th>
<th>Base level</th>
<th>Meta level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compile/Deployment time</td>
<td>C++ template programs</td>
<td>Compiler transformations; COMPOST</td>
</tr>
<tr>
<td>(static metaprogramming)</td>
<td>C sizeof(...) operator</td>
<td></td>
</tr>
<tr>
<td>Run time</td>
<td>Java Reflection</td>
<td>Debugger</td>
</tr>
<tr>
<td>(dynamic metaprogramming)</td>
<td>JavaBeans introspection</td>
<td></td>
</tr>
</tbody>
</table>

Reflective Architecture

- A system with a reflective architecture maintains metadata and a causal connection between meta- and base level.
  - The metobjects describe structure, features, semantics of domain objects
  - This connection is kept consistent
- Reflection is thinking about oneself (or others) at the base level with the help of metadata
- Metaprogramming is programming with metaobjects, either at base level or meta level

Reflective Architecture

- A system with a reflective architecture maintains metadata and a causal connection between meta- and base level.
  - The metobjects describe structure, features, semantics of domain objects
  - This connection is kept consistent
- Reflection is thinking about oneself (or others) at the base level with the help of metadata
- Metaprogramming is programming with metaobjects, either at base level or meta level

UML Metamodel and MOF

- UML metamodel specifies UML semantics
- In the form of a (UML) class model (= reification)
- Specified in UML Superstructure document (OMG 2006)
  - Using only elements provided in MOF
- UML metamodel: MOF (“Meta-Object Facility”)
  - Self-describing
  - Subset of UML (= reification)
  - For bootstrapping the UML specification
- UML Extension possibility 1: Stereotypes
  - e.g., `<metaclass>` is a stereotype (specialization) of a class
  - By subclassing metaclass “Class” of the UML metamodel

UML Metamodel (Simplified Excerpt)
Example: Reading the UML Metamodel

Some semantics rules expressed in the UML metamodel above:
- Each model element must have a name.
- A class can be a root, leaf, or abstract (inherited from GenerizableElement)
- A class can have many subclasses and many superclasses (1:N relations to class "Generalization")
- A class can have many features, e.g. attributes, operations (via Classifier)
- Each attribute has a type e.g. classes, interfaces, datatypes

UML vs. programming language metamodel hierarchies

Metamodel (L3)

UML metamodel: MOF

- ModelElement
  - name: Name

Metalevel (L2)

Class

- isActive: Boolean

Base level (L1)

Cat

- name: String
- color: Color
- age: Integer

Object level (L0)

tom : Cat

- name: "Tom"
- color: White
- age: 7

Caution

- A metamodel is not a model of a model but a model of a modeling language of models.
- A model (e.g. in UML) describes a language-specific software item at the same level of the metalevel hierarchy.
  - In contrast, metadata describes it from the next higher level, from which it can be instantiated.
- MOF is a subset of UML able to describe itself – no higher metalevels required for UML.

Markup Languages

- Convey more semantics for the artifact 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 introcession

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 still be used
- The composition environment can ask about the names in the interfaces of a component (introspection)
  - and can deduce more semantics from naming conventions
**Generic Types in COMPOST**

```java
<< ClassBox >>

class SimpleList {
    generic TType elem;
    SimpleList next;

    generic TType getNext() {
        return next.elem;
    }
}
```

**Java Beans Naming Schemes**

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

**Markup by Comments**

- **JavaDoc tags, XDoclet**
  - `@author`
  - `@date`
  - `@deprecated`
- **Java 1.5 attributes**
  - Can annotate any declaration e.g. class, method, interface, field, enum, parameter, ...
    - predefined and user-defined
  - ```java
class C extends B {
    @Overrides
    public int foo() { ... }
    ...
}
```
- **C# attributes**
  - ```csharp
@author
@date
@deprecated
```
- **C#/.NET attributes**
  - ```csharp
[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? (1)**

- **Reflection** is a program’s ability to reason about and possibly modify itself or other programs with the help of metadata.
  - Reflection is enabled by reflection of the metamodel.
  - Introspection is thinking about a program, but not modifying.

- A metaprogram is a program that computes about programs
  - Metaprograms can execute at the base level or at the meta-level.
  - Metacode can execute statically or at run time.
    - Static metaprogramming at base level
      - e.g. C++ templates, AOP
    - Static metaprogramming at meta level
      - e.g. Compiler analysis / transformations
    - Dynamic metaprogramming at base level
      - e.g. Java Reflection

**What Have We Learned? (2)**

- The UML metamodel is a description of UML specified in terms of the UML metamodel, MOF
  - UML models describe program objects on the same level of the meta-hierarchy level.
- Component and composition systems are reflective architectures
  - Markup marks the variation and extension points of components e.g., using Hungarian notation
  - Composition introspects the markup
  - Look up type information, interface information, property information
  - or full reflection