TDDD05 / DF14900
Component-Based Software

Metamodeling and Metaprogramming

Christoph Kessler, IDA, Linköpings universitet, 2010
Some slides by courtesy of U. Assmann, IDA / TU Dresden

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

U. Assmann: *Invasive Software Composition*, Sect. 2.2.5 Metamodeling;
C. Szyperski: *Component Software*, Sect. 10.7, 14.4.1 Java Reflection

Metadata

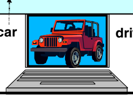
- **Meta**: means "describing"
- **Metadata**: describing data (sometimes: self-describing data).
  - The language (esp., type system) for specifying metadata is called **metamodel**.
- **Metalevel**: the elements of the meta-level (the meta-objects) describe the objects on the base level
- **Metamodeling**: description of the model elements/concepts in the metamodel

Metadata — Meta level, Concepts level

Data, Code, Information — Base level

Metalevels in Programming Languages

Level			
Level 3 - Meta-Concepts in the metameta model, the metalanguage (language description)	Programming Language Concept		
Level 2 - Language concepts (Metaclasses in the metamodel)	Class	Method	Attribute
Level 1 - Software Classes (meta-objects) (Model)	Car	void drive() {}	int color
Level 0 - Software Objects	car 1	car1.drive()	car1.color
"Real World" Entities	car	driving	car color



Metalevels in Programming Languages

Level			
Level 3 - Meta-Concepts in the metameta model, the metalanguage (language description)	Programming Language Concept		
Level 2 - Language concepts (Metaclasses in the metamodel)	Class	Method	Attribute
Level 1 - Software Classes (meta-objects) (Model)	Car	void drive() {}	int color
Level 0 - Software Objects	car 1	car1.drive()	car1.color
"Real World" Entities	car	driving	car color

Classes and Metaclasses

Classes in a software system

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

Metaclasses

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

Metadata — Meta level

Data, Code, Information — Base level

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

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

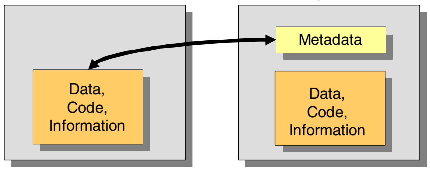
```
public class Class {
   Attribute[] fields;
   Method[] methods;
   Class ( Attribute[] f, Method[] m) {
     fields = f;
     methods = m;
   }
}
public class Attribute {..}
public class Method {..}
```

- Create a new class at runtime by instantiating the metaclass:

```
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 ConveyorBelt = new Class( new Attribute[]{ "WorkPiece[] pieces" },  new Method[]{});
```

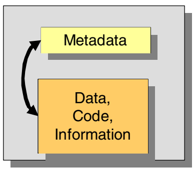

Introspection

- *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

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

```
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)

Good for language extension / customization, e.g. with UML MOF, or for compiler generation

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 0: objects
- Level 1: classes, types
- Level 2: language elements
- Level 3: metalanguage, language description language

Use of Metamodels and Metaprogramming

To model, describe, introspect, and manipulate

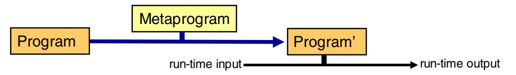
- Programming languages, such as Java Reflection API
- Modeling languages, such as UML or Modelica
- XML
- Compilers
- Debuggers
- Component systems, such as JavaBeans or CORBA DII
- Composition systems, such as Invasive Software Composition
- Databases
- ... many other systems ...

TDDD05 / DF14900
Component-Based Software

2. Different Ways of Metaprogramming

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

Metaprograms are programs that compute about programs



Christoph Kessler, IDA, Linköpings universitet, 2009.

Metaprograms can run at base level or at meta level

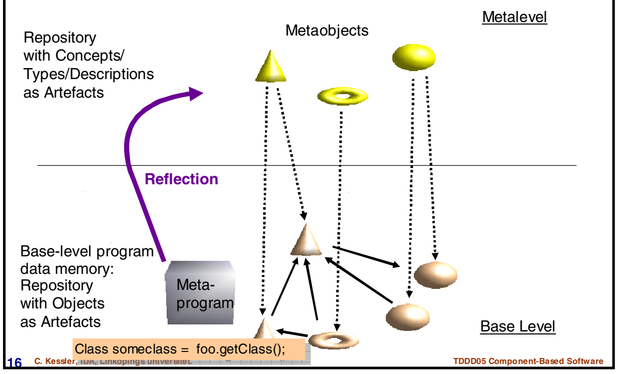
Metaprogram execution at the metalevel:

- Metaprogram is separate from base-level program
- Direct control of the metadata as metaprogram data structures
- Expression operators are defined directly on the metaobjects
- 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.

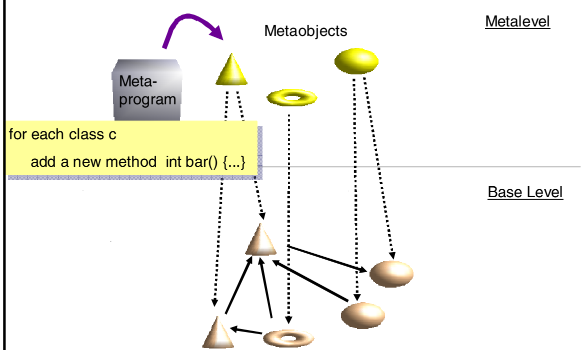
Metaprogram execution at the base level:

- 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

Base-Level Metaprogram



Meta-level Metaprogram

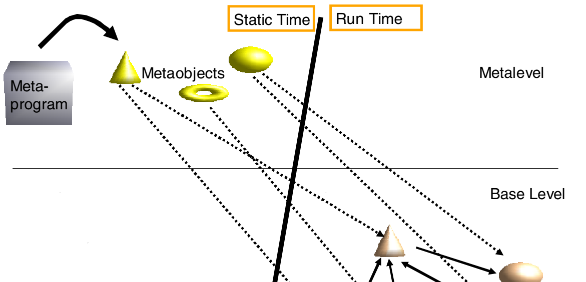


Static vs. Dynamic Metaprogramming

Recall: Metaprograms are programs that compute about programs.

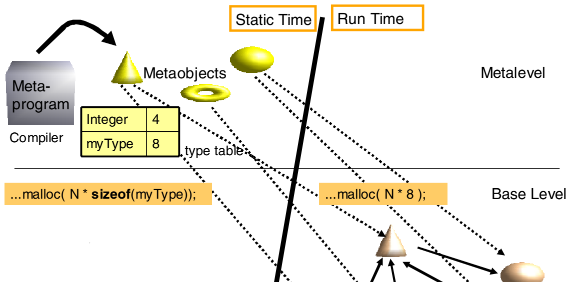
- **Static metaprograms**
  - Execute before runtime
  - Metainformation removed before execution – no runtime overhead
  - Examples: Program generators, compilers, static analyzers
- **Dynamic metaprograms**
  - Execute at runtime
  - Metadata stored and accessible during runtime
  - Examples:
    - Programs using reflection (Introspection, Intercession);
    - Interpreters, debuggers

Static Metaprogramming



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

Example: Static Metaprogramming (1)



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

Example: Static Metaprogramming (2)

C++ templates

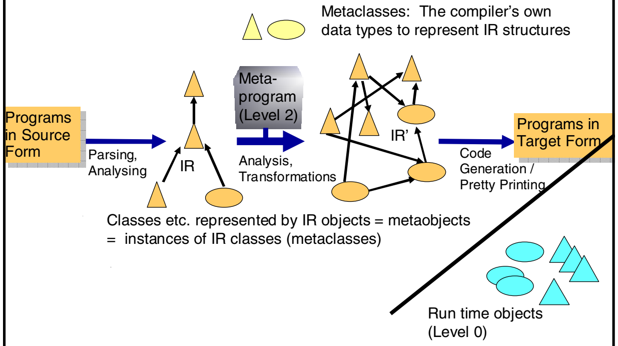
- Example: generic type definition
  - (Meta)Information about generic type removed after compiling!

```
template <class E>
class Vector {
  E *pelem;
  int size;
  E get( int index ) {...}
  ...
}
...
Vector<int> v1;
Vector<float> v2;
```

expanded at compile time to equivalent of:

```
class Vector_int {
  int *pelem;
  int size;
  int get( int index ) {...}
  ...
}
class Vector_float {
  float *pelem;
  int size;
  float get( int index ) {...}
  ...
}
...
Vector_int v1;
Vector_float v2;
```


Compilers Are Static Metaprograms



Compilers are Static Metaprograms

```
char x[7];
int a[13];
...
```

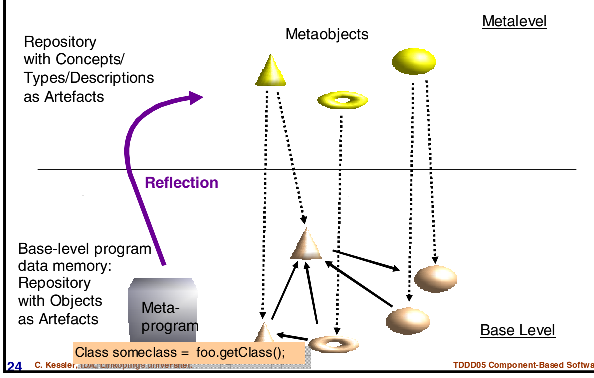
```
/* array - construct the type `array 0..n-1 of ty' with alignment a or ty's */
Type array( Type ty,  int n,  int a )
{
   if (ty && isfunc(ty)) {
      error( "illegal type `array of %t'\n", ty );
      return array ( inttype, n, 0 );
   }
   if (a == 0)
      a = ty->align;
   if (level > GLOBAL && isarray(ty) && ty->size == 0)
      error( "missing array size\n" );
   if (ty->size == 0) {
      if (unqual(ty) == voidtype)
         error( "illegal type `array of %t'\n", ty );
      else if (Aflag >= 2)
         warning( "declaring type `array of %t' is undefined\n", ty );
   } else if (n > INT_MAX / ty->size) {
      error( "size of `array of %t' exceeds %d bytes\n", ty, INT_MAX );
      n = 1;
   }
   return tynode ( ARRAY, ty, n * ty->size, a, (Generic)0 );
}
```

chartype	1
inttype	4
voidtype	0
ARRAY(7,chartype)	7
ARRAY(13,inttype)	52

type table excerpt

Source: lcc C compiler,
excerpt of file "types.c"
(type table management)

Dynamic Metaprogramming



Summary: Ways of Metaprogramming

Metaprogram runs at:	Base level	Meta level
Compile/Deployment time (static metaprogramming)	C++ template programs C sizeof(...) operator C preprocessor	Compiler transformations; COMPOST
Run time (dynamic metaprogramming)	Java Reflection JavaBeans introspection	Debugger

Reflection

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 (or others) at the base level with the help of metadata
- Metaprogramming is programming with metaobjects, either at base level or meta level

3. UML Metamodel and MOF

Christoph Kessler, IDA,
Linköpings universitet, 2009.

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

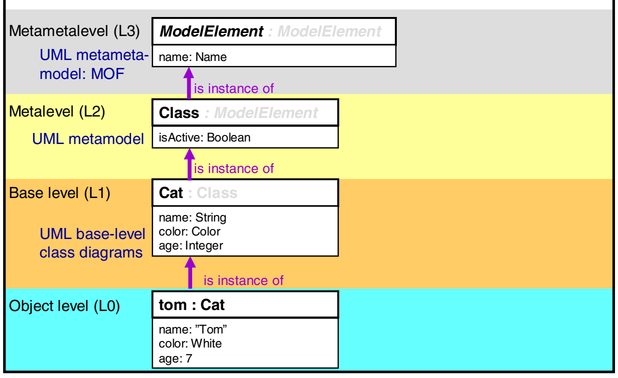
UML metametamodel: MOF ("Meta-Object Facility")

- self-describing
- subset of UML (= reification)
- for bootstrapping the UML specification

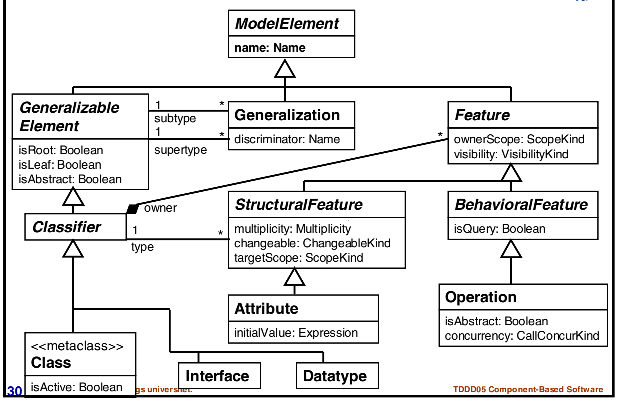
UML Extension possibility 1: Stereotypes

- e.g., <<metaclass>> is a stereotype (specialization) of a class
  - by subclassing metaclass "Class" of the UML metamodel

UML metamodel hierarchy



UML Metamodel (Simplified Excerpt)



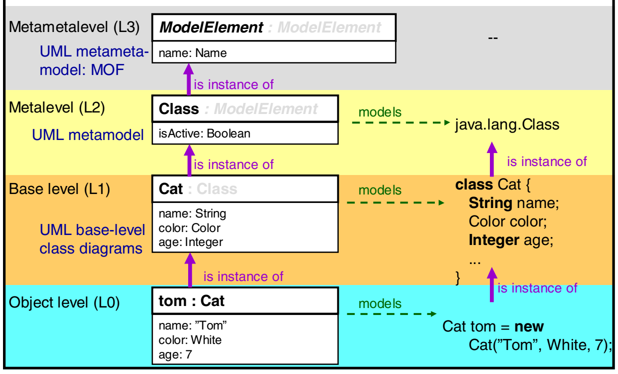
Example: Reading the UML Metamodel

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
  - (1:N relation to Classifier),

    e.g. classes, interfaces, datatypes

UML vs. programming language metamodel hierarchies



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.

4. Component Markup

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

Christoph Kessler, IDA,
Linköpings universitet, 2009.

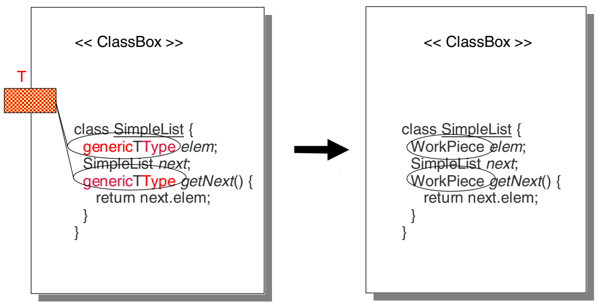
Markup Languages

- Convey more semantics for the artifact they markup
- HTML, XML, SGML are markup languages
- Remember: a component is a container
- Markup can make contents of the component accessible 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 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
  - **class** C **extends** B {
    @Overrides
    public int foo() { ... }
    ...
    }
- C# attributes
  - //@author
  - //@date
  - //selfDefinedData
- 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? (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 *reification* 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 metalevel.
  - 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 metametamodel, 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, Comments/Annotations, external markup (separate files referencing the contents)
  - Composition introspects the markup
  - Look up type information, interface information, property information
  - or full reflection