Metamodeling and Metaprogramming

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

[ISC] 2.2.5 Metamodeling; [CS] 10.7, 14.4.1

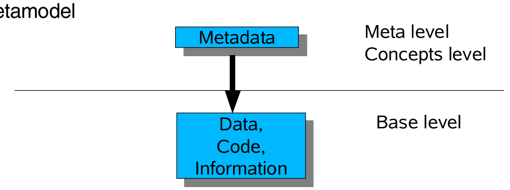
1. Introduction to Metalevels

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

P. Maes, ACM Proc. OOPSLA 1987

Metadata

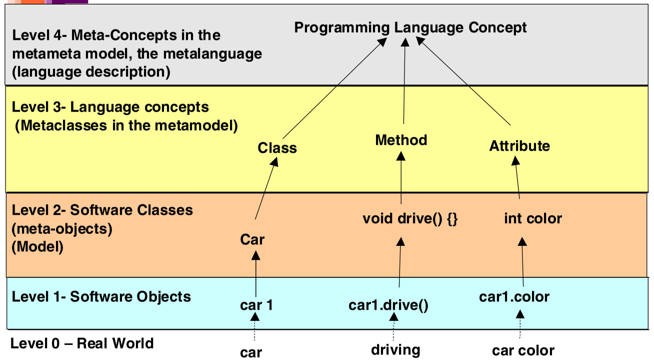
- *Meta*: means "describing"
- *Metadata*: describing data (sometimes: self-describing data). The type system is called *metamodel*
- *Metalevel*: the elements of the meta-level (the *meta-objects*) describe the objects on the *base level*
- *Metamodeling*: description of the model elements/concepts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



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

Metalevels in Programming Languages



Classes and Metaclasses

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

Classes in a software system

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)

Creating a Class from a Metaclass

```
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 {..}
public class Method {..}
```

- Example: Create a class by instantiating the metaclass:

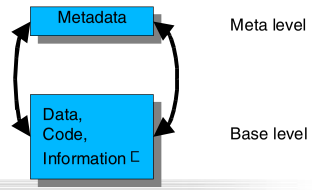
```
Class WorkPiece = new Class( new Attribute[]{ "Object belongsTo" }, new Method[]{});
Class RotaryTable = new Class( new Attribute[]{ "WorkPiece place1", "WorkPiece place2" },
                               new Method[]{});
Class Robot    = new Class( new Attribute[]{ "WorkPiece piece1", "WorkPiece piece2" },
                            new Method[]{});
Class Press    = new Class( new Attribute[]{ "WorkPiece place" }, new Method[]{});
Class ConveyorBelt = new Class( new Attribute[]{ "WorkPiece[] pieces" }, new Method[]{});
```

Metaprogram at base level

Reflection (Self-Modification, Metaprogramming)

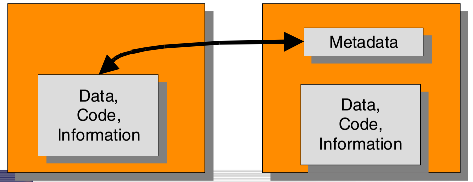
- Computation about the metamodel in the model is *reflection*.
- The application can look at its own skeleton (metadata) and may even change it
  - Allocating new classes, methods, fields
  - Removing classes, methods, fields

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



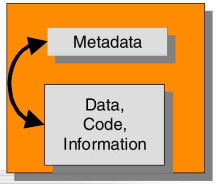
Introspection

- 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



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)

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 1: objects
- Level 2: classes, types
- Level 3: language
- Level 4: metalanguage, language description language

Use of Metamodels and Metaprogramming

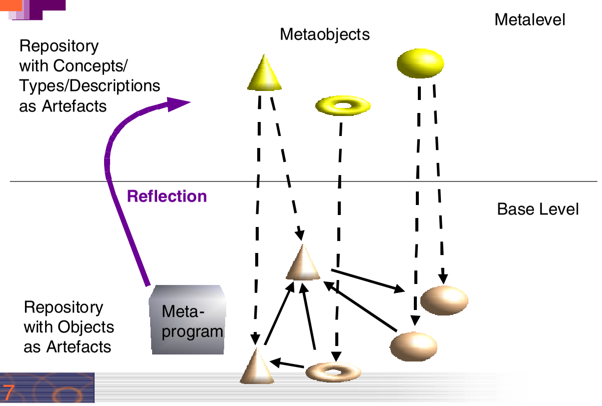
- To model, describe, introspect, and manipulate
- Workflow systems
- Databases
- Programming languages
- Compiler run-time systems for parallel computers
- Component systems, such as CORBA
- Composition systems, such as Invasive Software Composition
- Modeling systems, such as UML or Modelica
- ... 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

Reflective Architecture



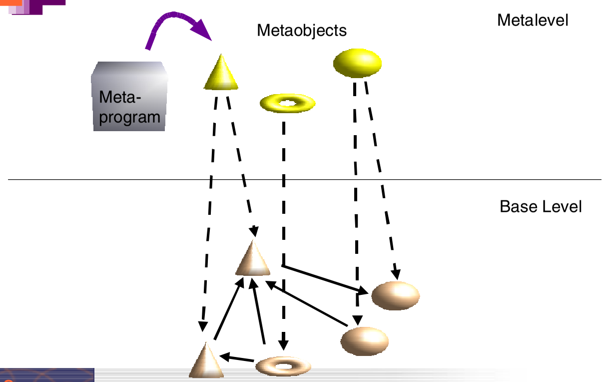
TDDC18 Component-based software. IDA, Linköpings universitet. Slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2005.

Metalevel Architecture

- 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

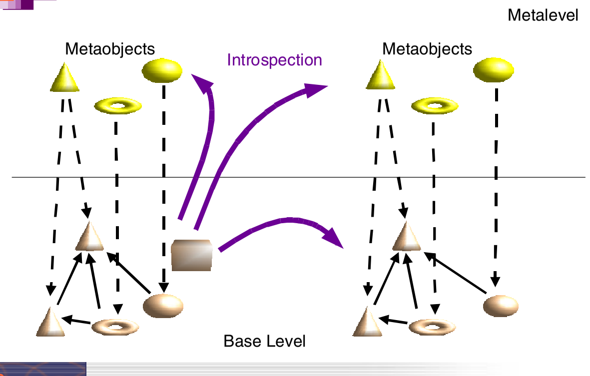
TDDC18 Component-based software. IDA, Linköpings universitet. Slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2005.

Metalevel Architecture

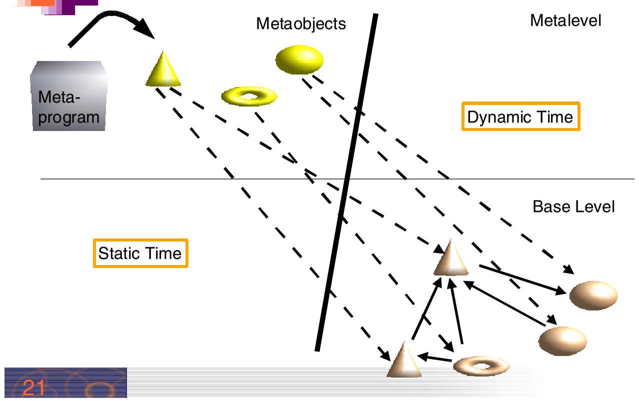


TDDC18 Component-based software. IDA, Linköpings universitet. Slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2005.

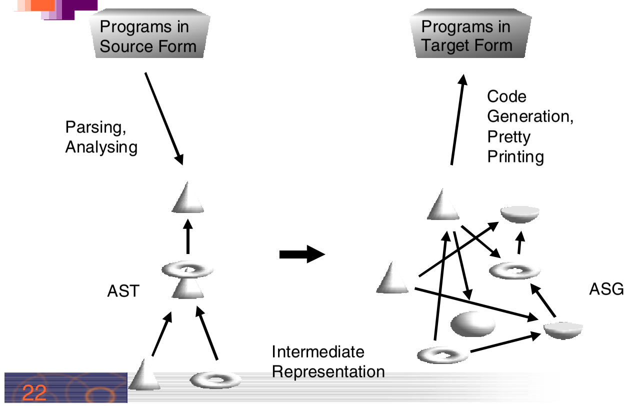
Introspective Architectures



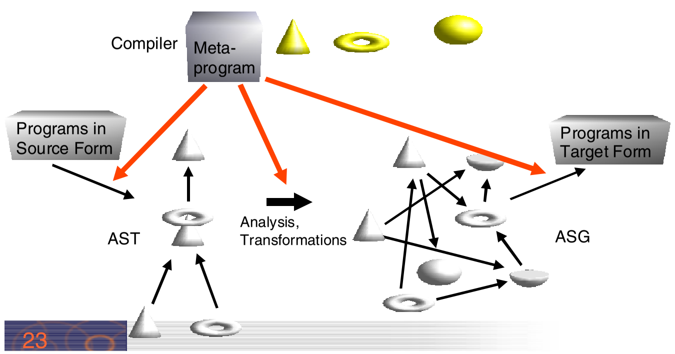
Static Metaprogramming Architecture



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

[ISC] p.52

```
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() { ... }
}
```


An Adapted MOP

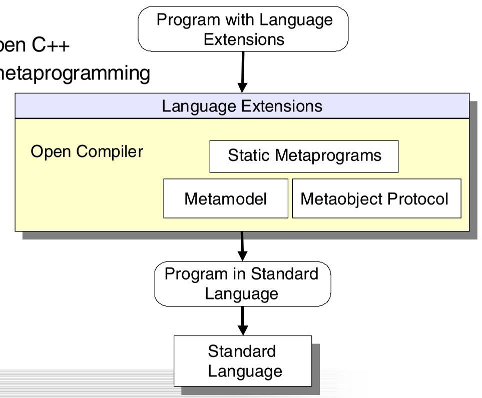
```
public class TracingAttribute extends Attribute {
  public void enterAttribute() {
    System.out.println("Here I am, accessing attribute " + name);
  }
  public void leaveAttribute() {
    System.out.println("I am leaving attribute " + name +
         ": value is " + value);
  }
}
```

```
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
```


Open Languages

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



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

4. Metaobject Facilities (MOF)

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

Metaobject Facility (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, ..)?

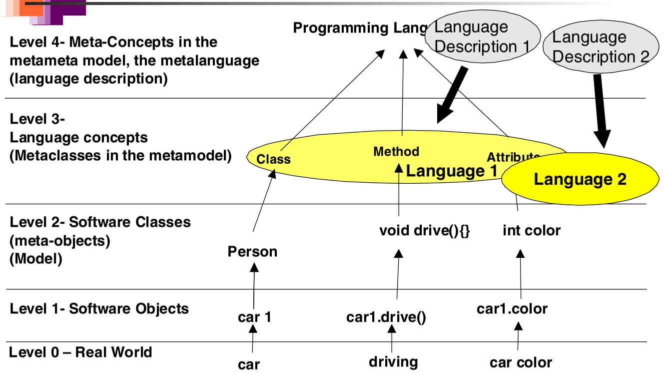
Metaobject Facility (MOF)

- 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)

TDDC18 Component-based software. IDA, Linköpings universitet. Slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2005.

The MOF Generator



TDDC18 Component-based software. IDA, Linköpings universitet. Slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2005.

MOF: Example

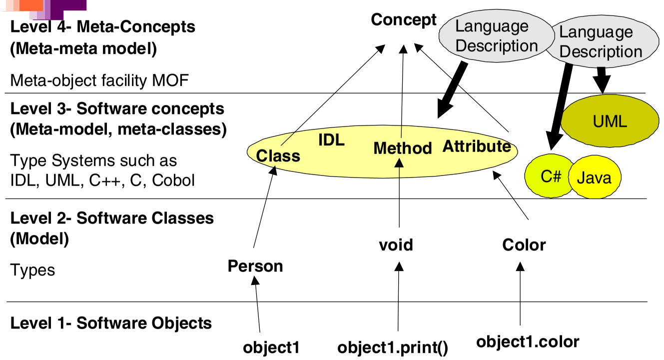
- The MOF for the CORBA meta-metamodel contains a *type system for type systems*:
  - Entities
  - Relationships
  - Packages
  - Exceptions
- Can describe every type system of a programming or modeling language
- MOF concepts must be mapped to types of a specific type system
- From these mappings, code can be generated that provides services for that type system, e.g. code that navigates in object graphs.

TDDC18 Component-based software. IDA, Linköpings universitet. Slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2005.

Metaobject Facility (MOF)

- From different language descriptions, different (parts of) 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



Meta-meta-models describe general type systems!

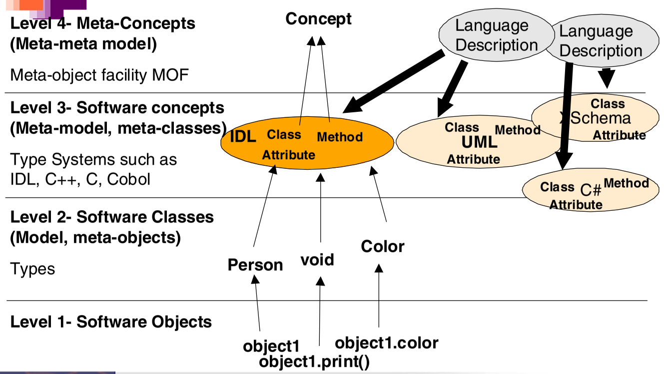
TDDC18 Component-based software. IDA, Linköpings universitet. Slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2005.

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)

TDDC18 Component-based software. IDA, Linköpings universitet. Slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2005.

Meta Levels in Corba Type Systems



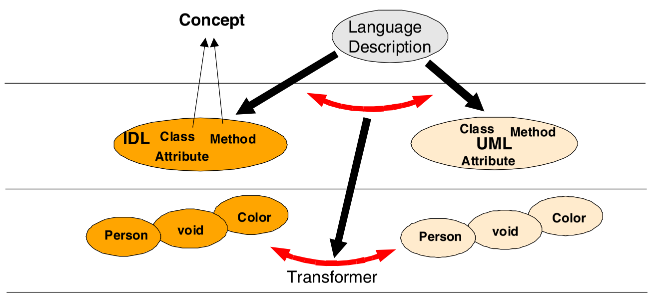
Meta-meta-models describe general type systems!

TDDC18 Component-based software. IDA, Linköpings universitet. Slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2005.

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

Isomorphic Language Mappings



TDDC18 Component-based software. IDA, Linköpings universitet. Slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2005.

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

TDDC18 Component-based software. IDA, Linköpings universitet. Slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2005.

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 data bases, groupware, business processes, data warehouses (Common Warehouse Model, CWM)

TDDC18 Component-based software. IDA, Linköpings universitet. Slides by courtesy of Uwe Assmann, IDA / TU Dresden. Revised by C. Kessler, 2005.

5. Component Markup

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

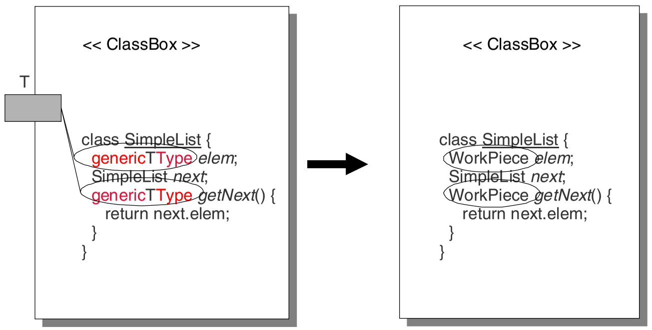
Markup Languages

- 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 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 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 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
- 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?

- *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 *reflective architecture* if meta- and base level use the same specification or programming language
- System has *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

What Have We Learned?

- Metamodeling, e.g. MOF for UML / Corba IDL / ...
- Some well-known examples of metaprogramming:
  - 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
- Component and composition systems are **reflective architectures**
  - Markup marks the variation and extension points of components
  - Composition introspects the markup