1. Introduction to Metalevels

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

Patti Maes, ACM OOPSLA 1987

---

Metamodeling and Metaprogramming

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

---

Metadata

- Meta: means "describing"
- Metadata: describing data (sometimes: self-describing data)
- Metamodeling: description of the model elements/concepts in the metamodel
- Metalevel: elements of the meta-level (the meta-objects) describe the objects on the base level
- Metaclasses: elements of the meta-level (the meta-objects) describe the objects on the base level

---

1. Level 0 – Real World Entities
   - car
   - driving
   - car color

2. Level 1 – Software Objects
   - car
   - drive() method
   - car.color attribute

3. Level 0 – Software Classes
   - Class
   - Method
   - Attribute

4. Level 3 – Language Concepts
   - Programming Language Concept

5. Level 2 – Metamodeling
   - Metaclasses

---

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 insensitive grammar (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

---

Classes and Metaclasses

- **Classes in a Software System**
  - Class: WorkPiece
    - Attributef1, f2
    - Method: piece1.place(), place2()

- **Metaclasses**
  - Public class Class
    - Attribute: f1
    - Method: piece1.place()

---

Examples:

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

---
Creating a Class from a Metaclass

- Example: Create a class by instantiating the metaclass:

```java
class RotaryTable

class Press

class WorkPiece

class Robot

class ConveyorBelt
```

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 its own
- Allocate and remove classes, methods, fields
- Allocating new classes, methods, fields
- Removing classes, methods, fields

Introression

- 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

Reflection Example

```
Reading Reflection (Introgression):
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 (Introgression):
for all c in self.classes do
helpClass = makeClass(c.name + "help");
for all a in c.attributes do
generate_class_start(c);
generate_attribute(a);
done;
helpClass.endClass();
done;
```

A reflective system is a system that uses this information about itself in its normal course of execution.
Made It Simple

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

Use of Metamodels and Metaprogramming

- To model, describe, introspect, and manipulate
- Workflow systems, such as ARIS [Scheer'98]
- Databases
- Debuggers
- Programming languages, such as Java Reflection API
- Component systems, such as JavaBeans or CORBA DII
- Composition systems, such as Invasive Software Composition
- Modeling systems, such as UML or Modelica
- ... probably all systems ...

Reflective Architecture

- A system with a reflexive architecture maintains metadata and a causal connection between meta- and base level.
  - The metatables 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 metatables, either at base level or meta level

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 at system build time, different from system runtime)
    - Example: C++ templates (generic), expanded at compile time
    - Example: COMPOST composition programs configure programs...
Metalevel Architecture

Metalevel architecture only allows metaprogramming at the meta level. Special case: Static metalevel architecture.

Static Metaprogramming Architecture

Static metaprogramming architecture: Metaprogram and metaobjects exist only at compile time. No run-time overhead.

Compilers

Programs in Source Form

Programs in Target Form

AST

Analysis, Transformations

Compilation, Pretty Printing

AST

Analysis, Transformations

Programs in Target Form

ASG

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

- Adaptation by subclassing the metaclass Attribute

An Adapted MOP

- 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

Open Languages

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

4. Metaobject Facilities (MOF)
Example: Generating IDL specifications

IDL = Interface Description Language
- The type system of CORBA
- Maps to many programming 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
- (See also lecture on CORBA)

Metamodel 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 metamodel specifications

A metamodel facility (MOF) is a generative mapping (transformer, generator) from the metamodel level (Level 4) to the language level (Level 3)

The MOF Generator

- 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

- Level 1: Software Objects
  - object1
  - object1.color
  - object1.print()

- Level 2: Software Classes
  - Person
  - void
  - Color

- Level 3: Software Concepts
  - (Meta-model, meta-classes)
  - Types
  - Type Systems such as IDL, UML, C++, C, Cobol

- Level 4: Meta-Concepts
  - (Meta-meta model)
  - Meta-object facility 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 metametamodel
  - Exchange data between tools possible

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
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)

5. Component Markup

- A simple aid for introspection and reflection...

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 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 still be used
- The composition environment can ask about the names in the interfaces of a component (introspection)
  - and can deduce more semantics

Java Beans Naming Schemes

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

Metainformation for Java Beans 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
  - // @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 reasoning and modification of oneself or others with the help of metadata.
- Reflection is enabled by reification of the metamodel.
- Introspection is thinking about oneself, but not modifying.
- Metaprogramming is programming with meta-objects.
- System has reflective architecture if metaprogram executes at base level and the base-model and metamodel are kept consistent
- System has metalevel architecture if it only supports metaprogramming at meta-level (not at the base level)
- 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
- UML MOF is a MOF for type systems mainly

### What Have We Learned? (2)

- 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
- Look up type information, interface information, property information or full reflection