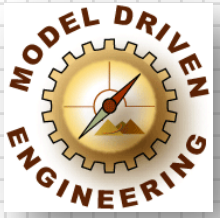


Model-Driven Engineering

From code to models

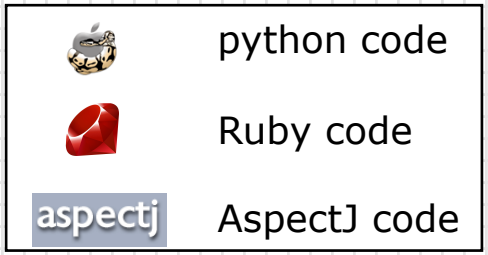
Jean-Michel Bruel

LatinAmerican School on SE, Rio de Janeiro, July 2013

> "To persuade me of the merit of your language, you must show me how to construct *models* in it."
>
> 1978 Turing Award Lecture

About this document…

- Target audience
  - Master's students
  - Basic knowledge in UML, Java, (Ruby)
- Author
  - Professor at IUT de Blagnac (Technical Institute)
  - For all comments: bruel@irit.fr
- More available here:
  - http://jmb.c.la



- Legend-

Content

- Why learning MDE from code ?
- Model or code
- Metaprogramming
- Aspect Oriented Programming
- Model-Driven Engineering
- Self-Adaptive Systems

What is it all about?

- Understand metaprogramming
- Apply it to understand MDE

```ruby
class BoutDeCode
        # "constructeur"
        def initialize(nom)
                @nom = nom
        end

        def bonjour
                puts "salut moi c'est " << @nom
        end
end
```

What is it all about? (ctd.)

```ruby
class BoutDeCode
    # "constructeur"
    def initialize(nom)
        @nom = nom
    end

    def bonjour
        puts "salut moi c'est " << @nom
    end
end

moi = BoutDeCode.new('JMB')
moi.bonjour # salut moi c'est JMB

# pb de la variable privée :
puts moi.nom # undefined method `nom' for ...

# solution, on ajoute un accesseur :
class BoutDeCode
    def get_nom
        @nom
    end
end
puts moi.get_nom # JMB

# mais par meta-programmation :
class BoutDeCode
    attr_reader :nom
end
puts moi.nom # JMB (c'est quand même plus élégant)
```

What is it all about? (ctd.)

- Several possible usage:
  - Code Evolution
  - Dynamic Adaptation
    - example: self-healing
    - Notion of context or « mode »
  - Deployment on heterogeneous targets
  - etc.

What is it all about? (ctd.)

- Another Python example

exemple.py – /Users/JMB/Documents/Enseignement/TI...

```
class BoutDeCode(object):
        # "constructeur"
        def __init__(self,nom="John Doe"):
                self.nom = nom

        def bonjour(self):
                print "salut moi c'est " + self.nom

moi = BoutDeCode('JMB')
print "Salut c'est moi " + moi.nom # Salut c'est moi JM
```

Ln: 11 C

exemple-property.py – /Users/JMB/Documents/Enseigne...

```
class BoutDeCode(object):
        # "constructeur"
        def __init__(self,nom="John Doe"):
                self.nom = nom

        def bonjour(self):
                print "salut moi c'est " + self.nom

        def getNom(self):
                return self.__nom

        def setNom(self,str):
                self.__nom = str

        nom = property(getNom,setNom)

moi = BoutDeCode('JMB')
print "Salut c'est moi " + moi.nom # Salut c'est moi JMB
```

Ln: 19 Col: 0

Model or code?

- Possible collaboration Model/Code:
  - code as "*first citizen*"
    - XP
  - model as "*first citizen*"
    - MDE
  - model => code
    - Code generation
  - code => model
    - Reverse engineering
  - sharing at same time code / model
    - Recent tools
  - models@runtime

Content

- Why learning MDE from code ?
- Model or code
- Metaprogramming
- Aspect Oriented Programming
- Model-Driven Engineering
- Self-Adaptive Systems

Model or code (ctd.)?

- Code generation
  - Advantages
    - productivity
    - quality
    - Maintainability
  - How
    - Files (e.g., XML)
    - annotations / markup
  - Limitations
    - *"code is more read than written"*

Model or code (ctd.)?

- models@runtime
  - Advantages
    - Focus on design
    - More users in the loop
    - Knowledge acquisition and capitalization
  - How
    - Specific tools (e.g. Kermeta)
    - Dedicated libraries (e.g. PauWare)
  - Limitations
    - Too precise too early

Model or code (ctd.)?

- profiles / DSL
  - profiles
    - adapt/refine the semantic (e.g., WPCP)
    - validation rules
    - Numerous tools
  - *Domain Specific Languages* (DSL)
    - Close to users
    - Initial work is important

Integrated approach

- Apply programming techniques to models
  - languages to manipulate artefacts
  - languages to transform models
- Study meta-programming
  - definitions
  - examples

Content

- Why learning MDE from code ?
- Model or code
- Metaprogramming
- Aspect Oriented Programming
- Model-Driven Engineering
- Self-Adaptive Systems

WhyMeta?

- Some thoughts
  - Why should we manipulate data and functions only (what about code itself)?
  - Some functionality "*crosscut*"
  - Extra-functional properties
- // with metadata
  - "About …"
  - Data about data
  - e.g., HTML META tag

Definitions

- Abstraction
  - *"one thing that represents several real things equally well"* (E.W. Dijkstra, 1967)
- Model
  - System description
    - Built for a precise purpose
    - Usable to predict or analyze its behavior
- Aspect (in modeling)
  - System view

Definitions (ctd.)

- Metaprogramming:
  - Writing programs using programs
  - Several possible ways:
    - Code generation
    - Templates programming (e.g. C++)
    - Powerful languages (Java, Ruby, python)
    - Macros
    - etc.

Definitions (ctd.)

- Reflection :
  - A language is called reflective if it allows:
    - introspection: possibility to analyze the language itself
    - intercession: possibility to modify the objects' behavior at run-time
    - Examples
      - Smalltalk
      - Java
      - Ruby

Reflection

- Different types
  - structural (object->class->metaclass->…)
  - operational (memory, resources, etc.)
- Different timing
  - compilation time
  - load time
  - run time

Limitations

- Low performance
- Encapsulation broken

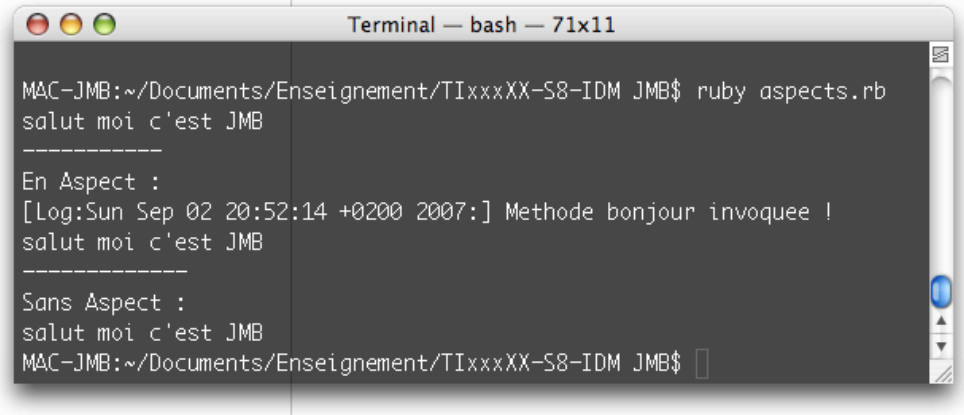
Content

- Why learning MDE from code ?
- Model or code
- Metaprogramming
- Aspect Oriented Programming
- Model-Driven Engineering
- Self-Adaptive Systems

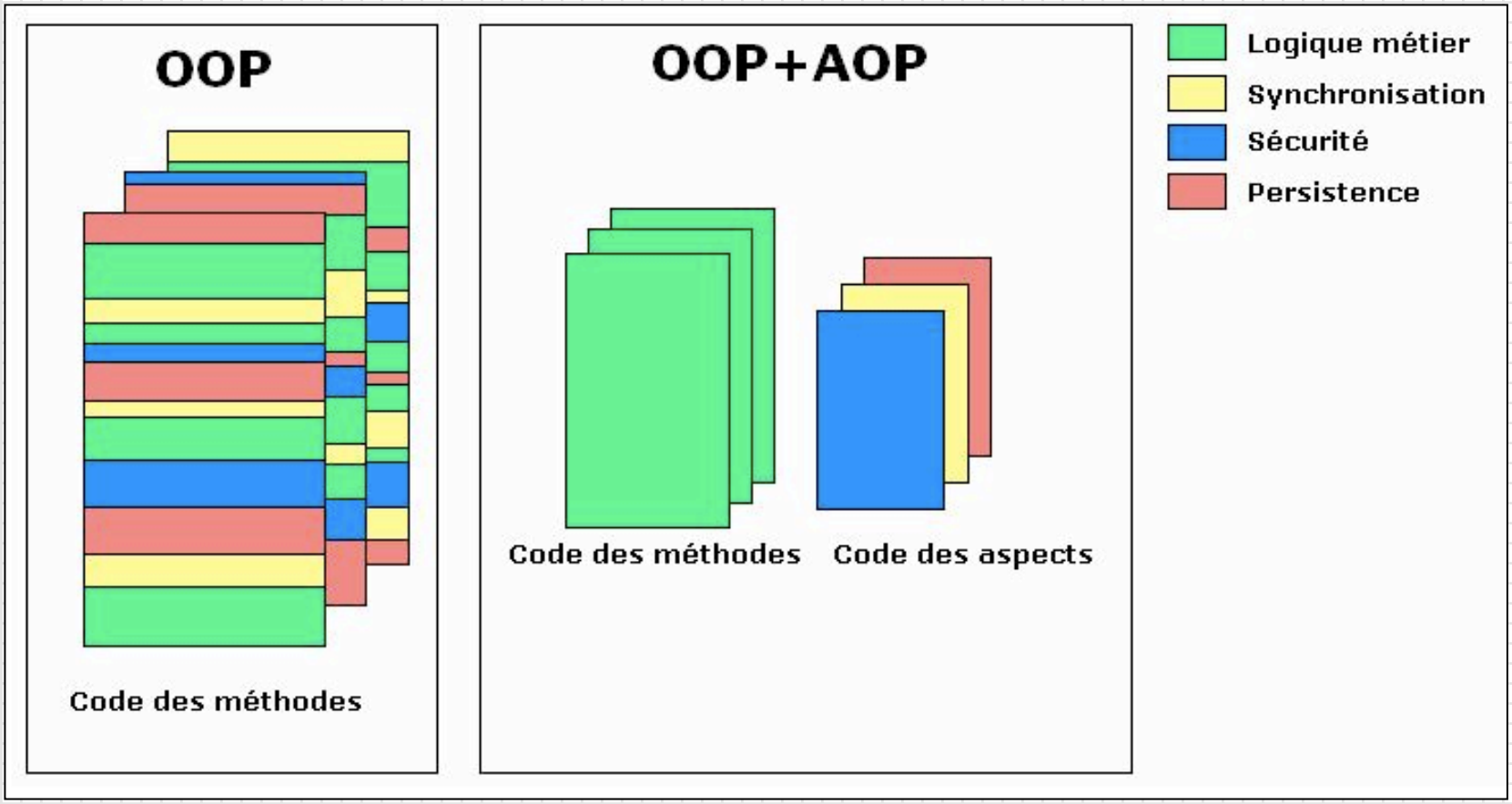
Aspect-Oriented Programming (AOP)

- Generalization of metaprogramming
- Modularity, reusability, etc.
- Cf. Ruby example

```ruby
# Solution 2 : Aspects
require 'rubygems'
require 'aspectr'
class VerbeuxAspect < AspectR::Aspect
    def avant(sym_methode, obj, val_ret, *param)
        today = Time.now
        puts "[Log:#{today}:] Methode #{sym_methode} invoquee !"
    end
end
# weaving
verbeux = VerbeuxAspect.new
verbeux.wrap(BoutDeCode, :avant, nil, :bonjour)

# le même simple code
puts "-----------"
puts "En Aspect :"
moi.bonjour

# avantage on peut enlever :-)
verbeux.unwrap(BoutDeCode, :avant, nil, :bonjour)
puts "-------------"
puts "Sans Aspect :"
moi.bonjour
```



AOP (ctd.)



http://www.dotnetguru.org

AOP (ctd.)

- Classical usage:
  - Logging/Tracing
  - Exceptions management
  - Debugging/Tests
  - Assertions/Contracts/Constraints
  - Performances optimization
  - Change detection
  - Access Synchronization
  - Persistence
  - Security/login
  - Design patterns

AOP (ctd.)

- Limitations
  - Aspect = code to be inserted at certain points in the application
  - Resulting code complicated
  - Difficult to test

Classes as objects

- Class Manipulation
  - Avoid repetitive code
  - Find an object' class
  - Modify a class dynamically

Avoid repetitive code

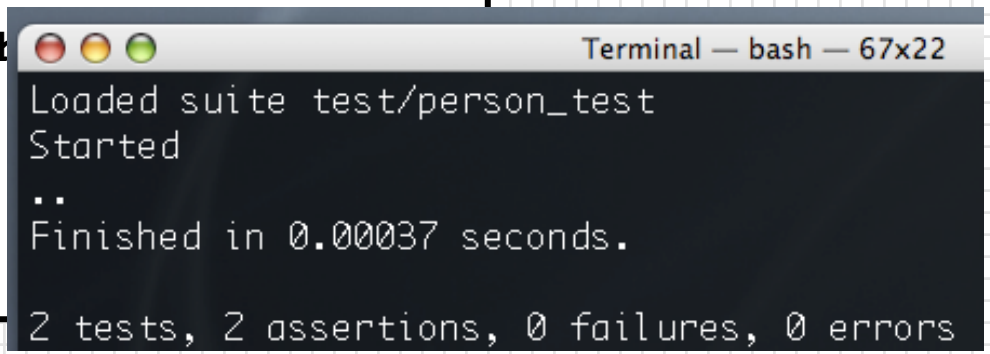
- Example of accessors

```
class BoutDeCode
        # "constructeur"
        def initialize(nom)
                @nom = nom
        end
…
end
…
# oups, j'ai oublié les accesseurs
class BoutDeCode
        attr_accessor :nom
end
```

Avoid repetitive code (ctd.)

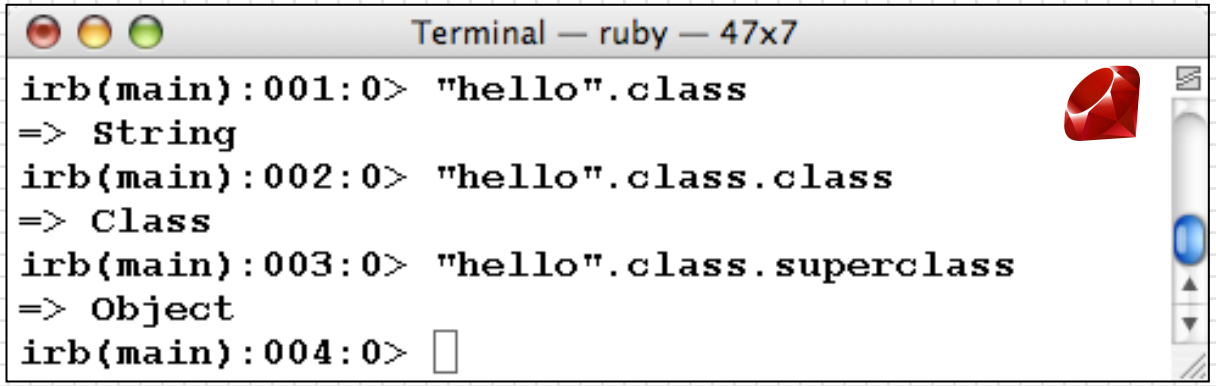
- Example of tests

```
#test/person_test.rb
require File.join(… 'person')
require 'test/unit'
class PersonTest < Test::Unit::TestCase
        def test_first_name
          p = Person.new('John','Doe',25)
          assert_equal 'John' p.first_name
        end
        def test_last_name
          p = Person.new('Joh
          assert_equal 'Doe'
        end
        …
end
```



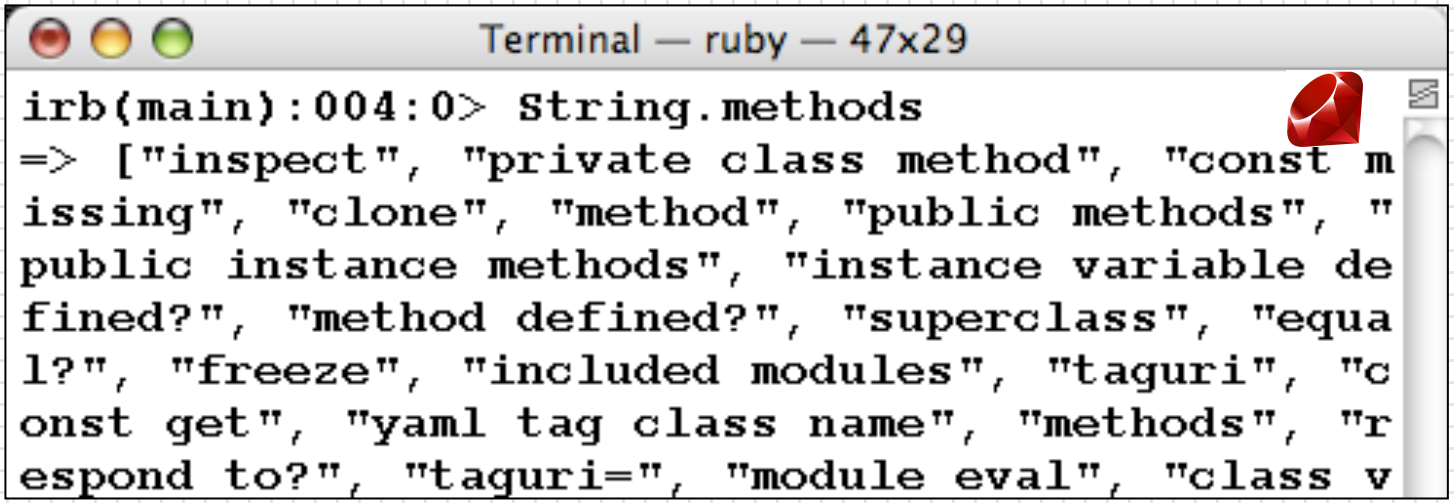
Find an object' class

- trivial for interpreted languages



- dedicated libraries
  - Java : java.lang.reflect
  - .Net : System.Reflection

Find an object' class



Modify a class

- Dynamicity and adaptation
  - Example: method redefinition

```
class Foo # déjà définie ailleur
        # la méthode foom est buggée
        # on la renomme au cas où
        alias :foom_BUG :foom
        # et on la redéfinit
        def foom
          …
        end
end
```

AOP

- Basic Concepts
  - *Pointcut*
    - Manipulable code element (e.g., **`function f`**)
  - *Jointpoint*
    - Precise activity (method call, constructors, etc.)
  - *Advice*
    - code to be activated (e.g., **`log`**)
    - Moment to activate (e.g., **`before`**)
- Aspect weaving
  - Available for almost all languages
  - Can be seen as a particular model transformation

AspectJ

- Pointcuts
  - call/execution
  - get/set
  - handler
  - initialization()
  - adviceexecution()

```aspectj
// exemple de pointcut
call(void Point.SetX(int))
```

AspectJ (pointcuts)

- Method signature

```aspectj
call(void Point.SetX(int))
```

- Pointcuts combination
  - ||, &&, !

```aspectj
call(void Point.SetX(int)) ||
call(void Point.SetY(int))
```

- Naming possibility

```aspectj
pointcut modified() :
        call(void Point.SetX(int)) ||
        …
```

AspectJ (pointcuts, ctd.)

- Genericity

```aspectj
call(void Point.Set*())
```

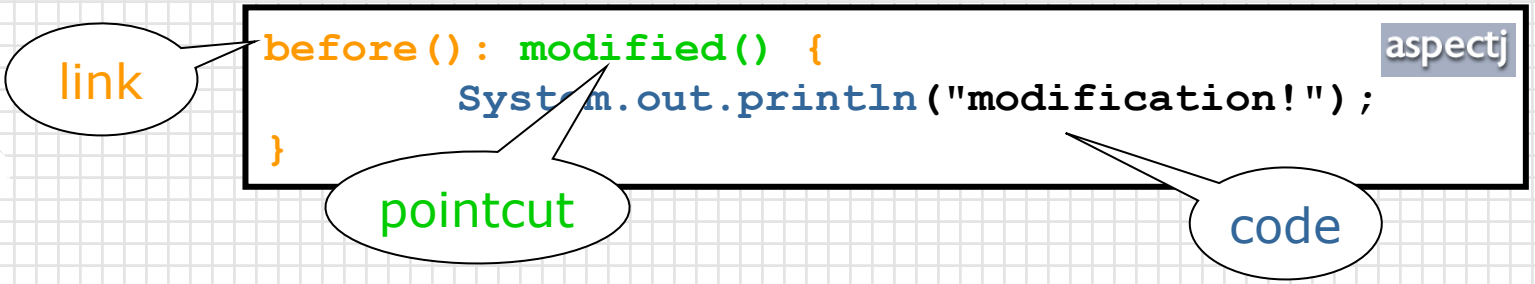
- Even on pointcuts themselves

```aspectj
cflow(modified())
```

- Existing primitive pointcuts
  - this
  - target
  - args

AspectJ (advices)

- Link between pointcut & code



- Possible Links:
  - before
  - after
  - around

AspectJ (advices, ctd.)

- parameters

```aspectj
pointcut setXY(FigureElement fe, int x, int y):
    call(void FigureElement.setXY(int, int))
    && target(fe)
    && args(x, y);

after(FigureElement fe, int x, int y) returning:
setXY(fe, x, y) {
    System.out.println(fe + " moved to (");
    System.out.println(x + ", " + y + ").");
}
```

AOP (ctd.)

- Ruby: **`require 'aspectr'`**

```ruby
# Solution 2 : Aspects
require 'rubygems'
require 'aspectr'
class VerbeuxAspect < AspectR::Aspect
    def avant(sym_methode, obj, val_ret, *param)
        today = Time.now
        puts "[Log:#{today}:] Methode #{sym_methode} invoquee !"
    end
end
# weaving
verbeux = VerbeuxAspect.new
verbeux.wrap(BoutDeCode, :avant, nil, :bonjour)

# le même simple code
puts "-----------"
puts "En Aspect :"
moi.bonjour

# avantage on peut enlever :-)
verbeux.unwrap(BoutDeCode, :avant, nil, :bonjour)
puts "-------------"
puts "Sans Aspect :"
moi.bonjour
```

```
Terminal — bash — 71x11

MAC-JMB:~/Documents/Enseignement/TIxxxXX-S8-IDM JMB$ ruby aspects.rb
salut moi c'est JMB
-----------
En Aspect :
[Log:Sun Sep 02 20:52:14 +0200 2007:] Methode bonjour invoquee !
salut moi c'est JMB
-------------
Sans Aspect :
salut moi c'est JMB
MAC-JMB:~/Documents/Enseignement/TIxxxXX-S8-IDM JMB$
```

AOP (ctd.)

```ruby
# Solution 2 : Aspects
require 'rubygems'
require 'aspectr'
class VerbeuxAspect < AspectR::Aspect
    def avant(sym_methode, obj, val_ret, *param)
        today = Time.now
        puts "[Log:#{today}:] Methode #{sym_methode} invoquee !"
    end
end
# weaving
verbeux = VerbeuxAspect.new
verbeux.wrap(BoutDeCode, :avant, nil, :bonjour)

# le même simple code
puts "-----------"
puts "En Aspect :"
moi.bonjour

# avantage on peut enlever :-)
verbeux.unwrap(BoutDeCode, :avant, nil, :bonjour)
puts "-------------"
puts "Sans Aspect :"
moi.bonjour
```

```
Terminal — bash — 71x11

MAC-JMB:~/Documents/Enseignement/TIxxxXX-S8-IDM JMB$ ruby aspects.rb
salut moi c'est JMB
-----------
En Aspect :
[Log:Sun Sep 02 20:52:14 +0200 2007:] Methode bonjour invoquee !
salut moi c'est JMB
-------------
Sans Aspect :
salut moi c'est JMB
MAC-JMB:~/Documents/Enseignement/TIxxxXX-S8-IDM JMB$
```

MDE Application

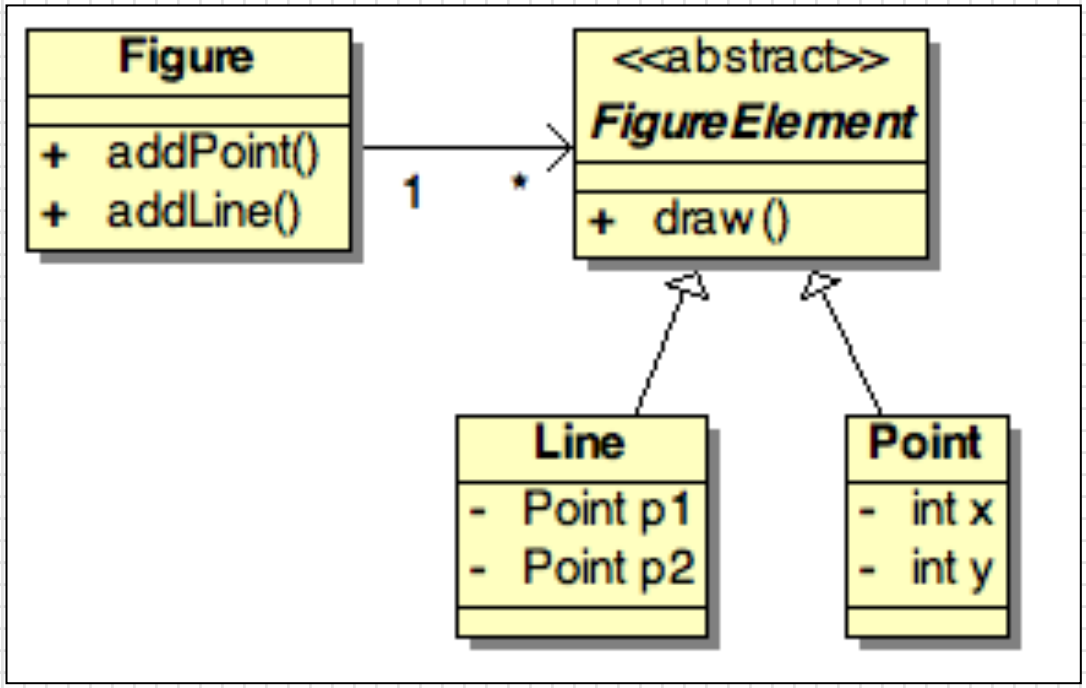
- Same manipulation
- No real dedicated support yet
  - A-QVT / A-ATL ?
  - Aspect code generation -> QVT/ATL?

MDE Application (ctd.)

- Design patterns
  - *"Aspectization"* of design patterns
- Use for model transformation

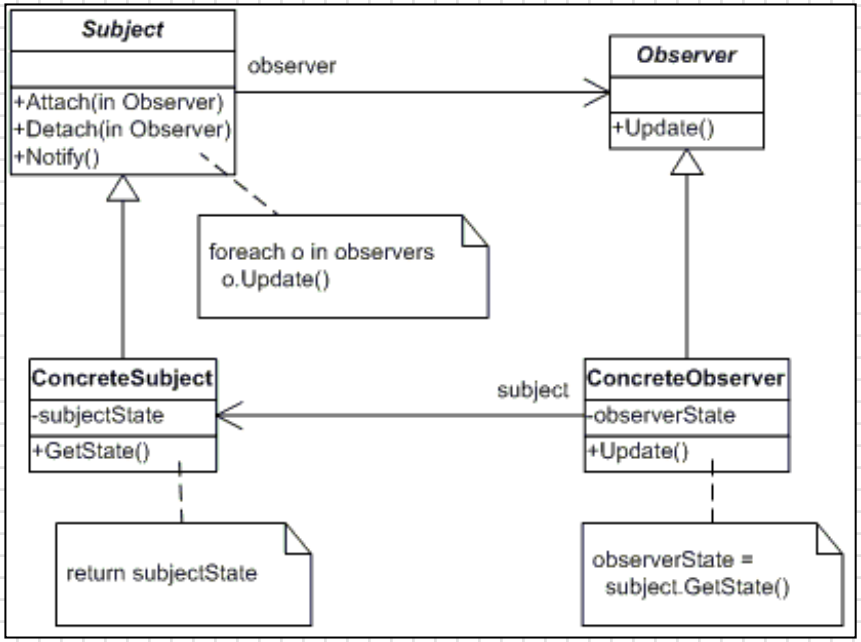
Patron Observer (java)

- Figures example

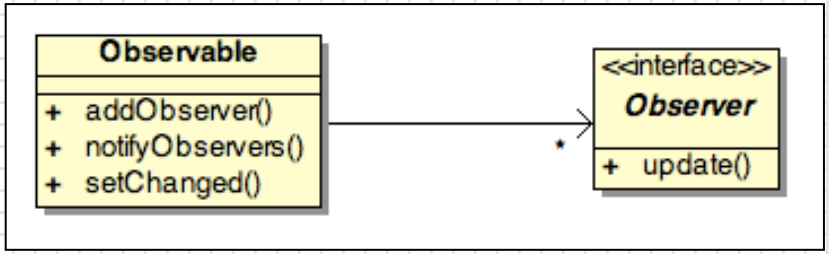


Patron Observer (java, ctd.)

- Pattern



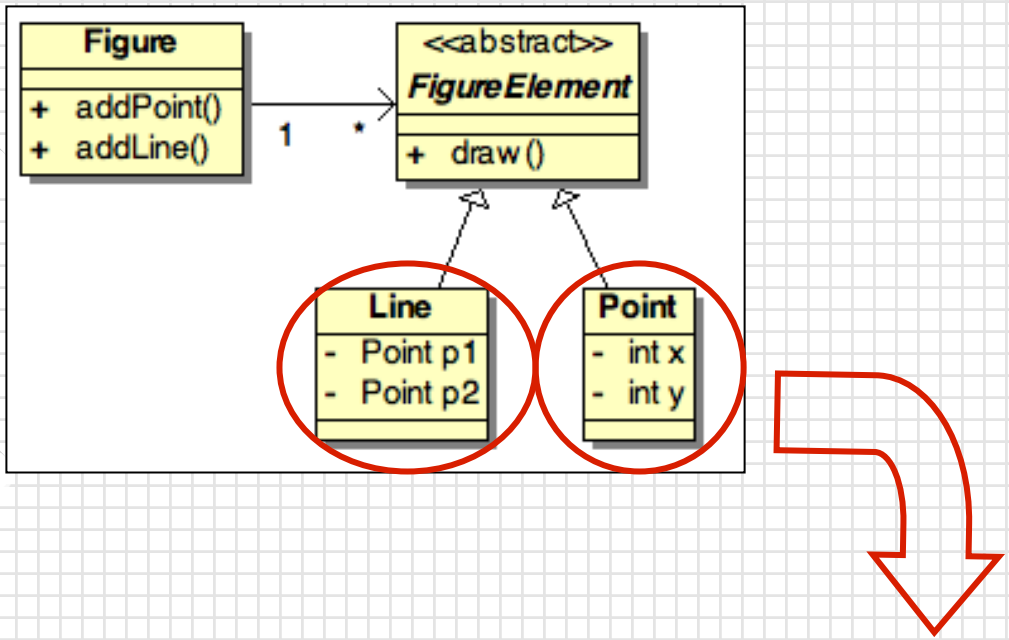
[Gamma, 1995]



[java.util]

Patron Observer (java, ctd.)

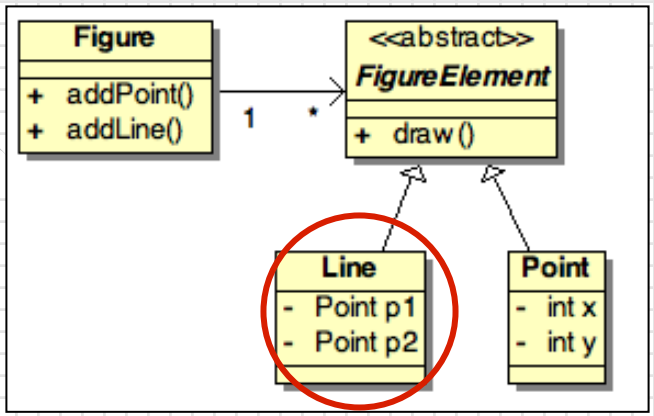
- What is observable?




```
public abstract class FigureElement extends Observable{…}
```

Patron Observer (java, ctd.)

- Who observes what?

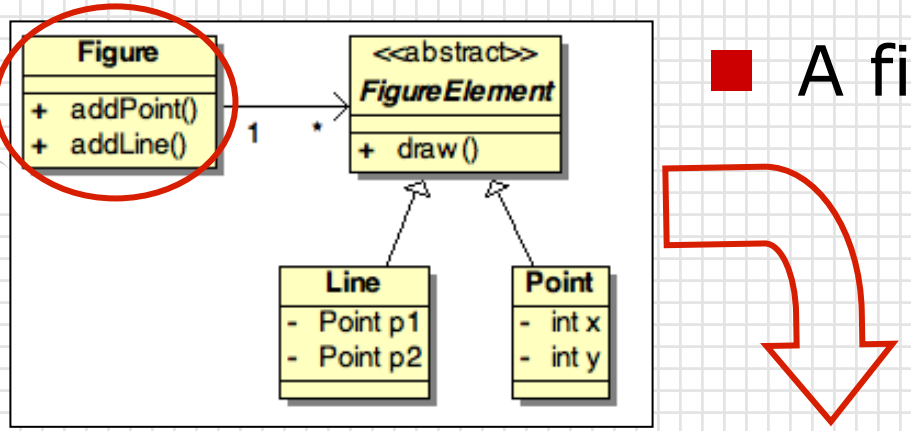


- A line observes "its" points

```
public class Line extends FigureElement implements Observer{
        Point p1,p2;

        public Line(Point origine, Point destination) {
                p1 = origine;
                p2 = destination;
                p1.addObserver(this);
                p2.addObserver(this);
        }
```

Patron Observer (java, ctd.)

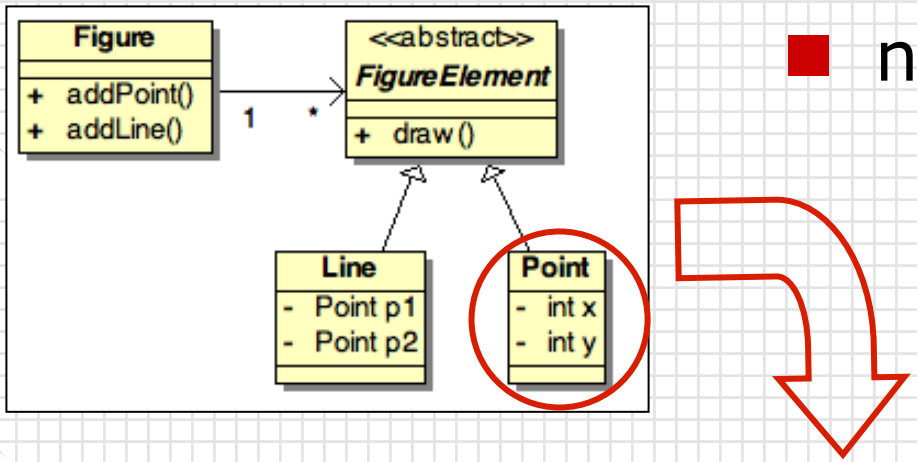
- Who observes what?
  - A figure observes everything




```
public class Figure implements Observer {
…
public void addLine(line l){
          this.ens.add(l);
          l.addObserver(this);
}
…
```

Patron Observer (java, suite)

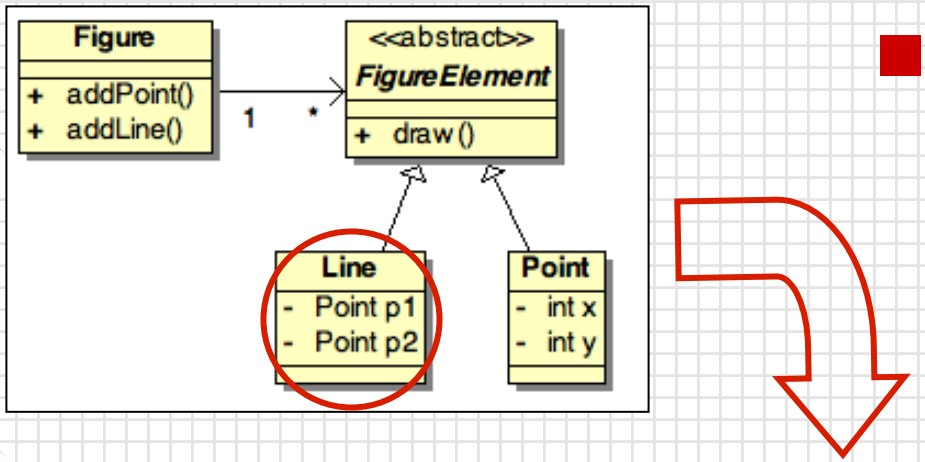
- What happens when a change occurs?
  - notifying




```
public void setXY(int x, int y) {
        …
        setChanged();
        this.notifyObservers();
}
```

Patron Observer (java, suite)

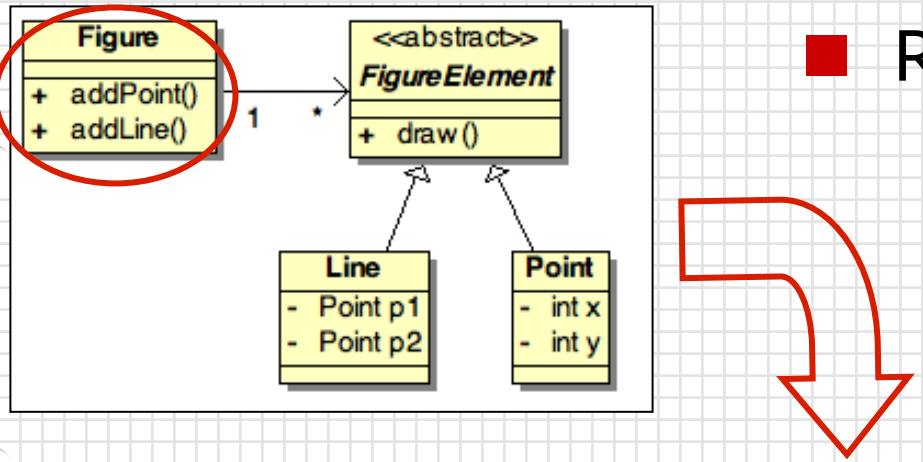
- What happens when a change occurs?
  - Reacting (line example)




```
public void update(Observable o, Object arg) {
        if (o==p1)
                System.out.print("modification de mon p1");
        else System.out.print("modification de mon p2");
        draw();
}
```

Patron Observer (java, suite)

- What happens when a change occurs?
  - Reacting (figure example)




```
public void update(Observable o, Object arg) {
        if (o instanceof Point)
                System.out.println("modif. d'un point");
        else
                System.out.println("modif. d'une ligne");
        ((FigureElement)o).draw();
}
```

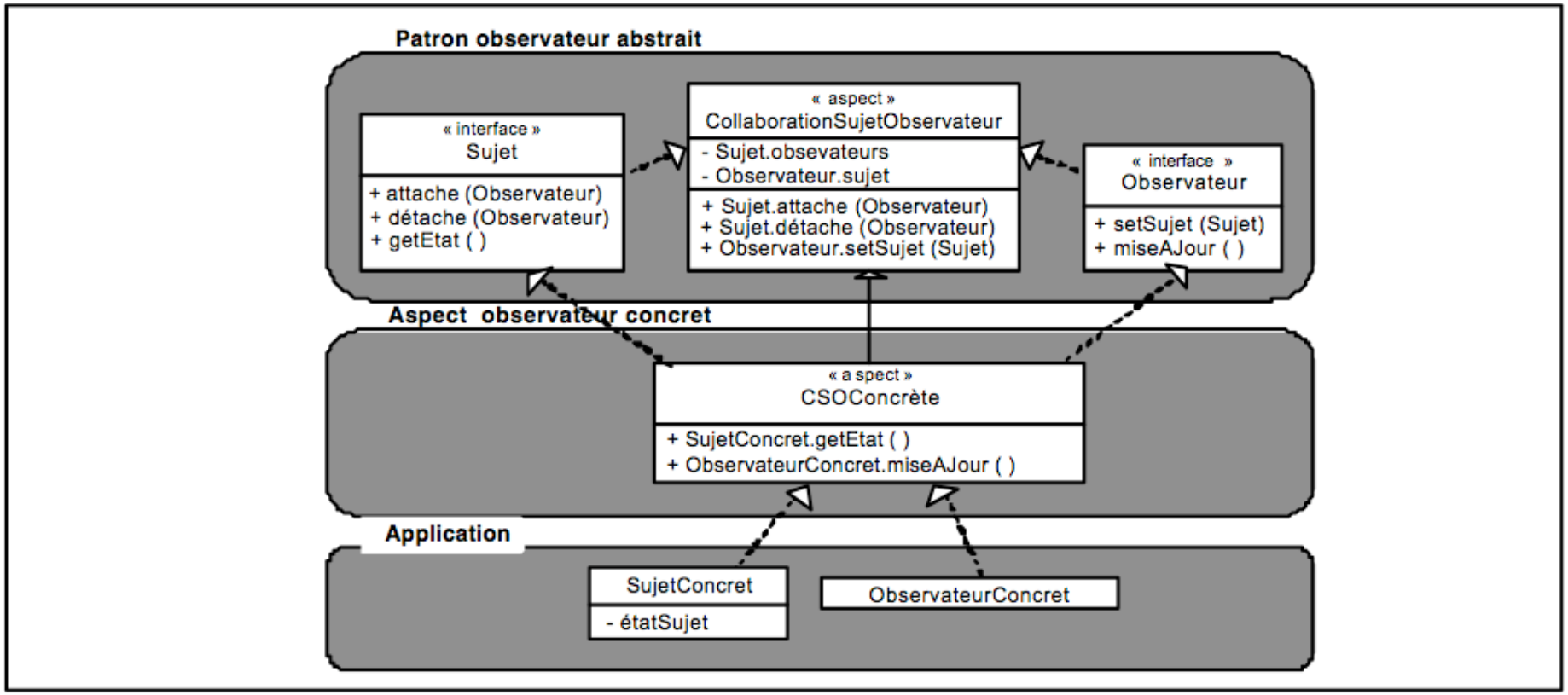
Observer (AspectJ)

- Either we stay close to the code
  - Modifying classes consequently
    - `extends Observable`
    - `implements Observer`
  - List all pointcuts
    - Members modification (e.g., `setXY`)
  - Coding advices
    - `setChanged();`
    - `this.notifyObservers();`
    - `public void update() {}`

Observer (AspectJ)

- Either we "simulate"
  - List of pointcuts
    - Members modification (e.g., setXY)
  - Coding advices
    - observers management (internally)
  - Concrete and Abstract Aspect (generic)

Observer (AspectJ)



[Inforsid2003]

MDE application

- Design patterns

- Use for model transformation
  - See Research papers

Xtext

- Framework for DSL
  - Domain Specific Language
  - Integrated with Eclipse EMF
- Allows automated generation of:
  - Parser/lexer (for code generation)
  - Ecore MM
  - Moels Import/export into EMF

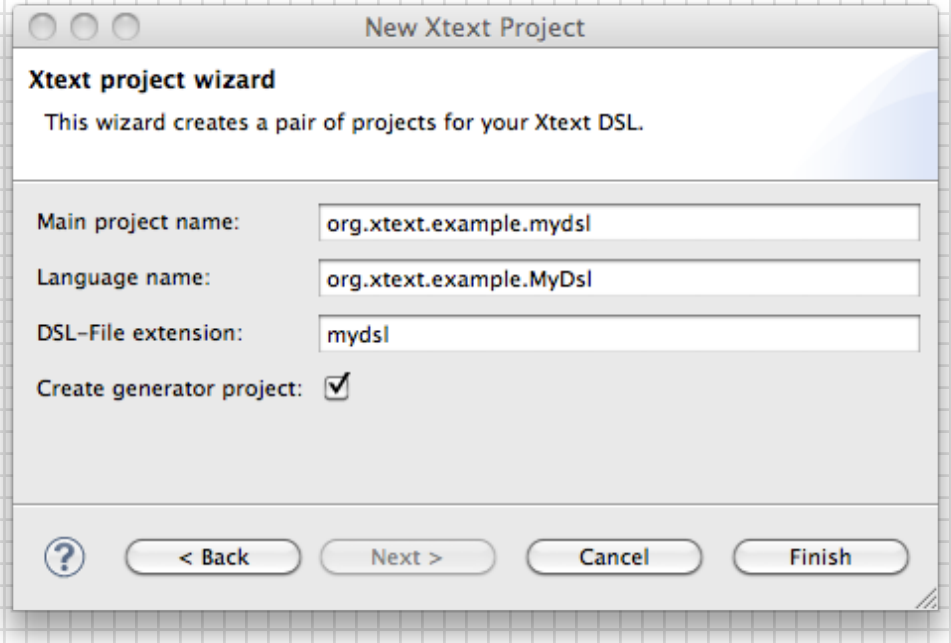
General principles

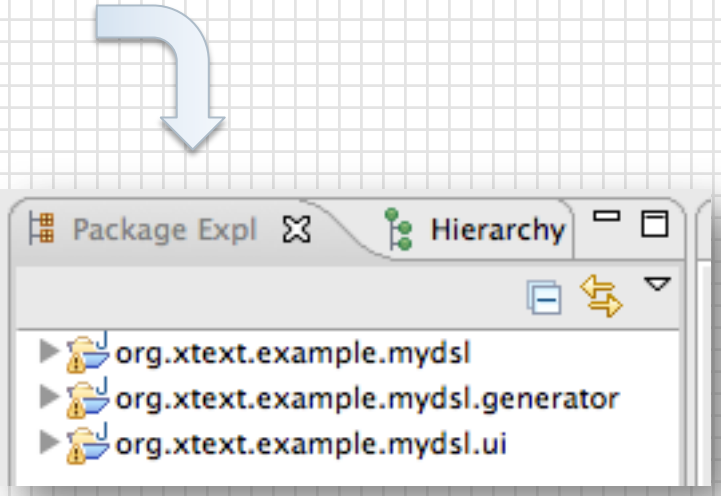
- Example of target syntax

```
type String
type Bool

entity Session {
  property Title: String
  property IsTutorial : Bool
}

entity Conference {
  property Name : String
  property Attendees : Person[]
  property Speakers : Speaker[]
}

entity Person {
  property Name : String
}

entity Speaker extends Person {
  property Sessions : Session[]
}
```

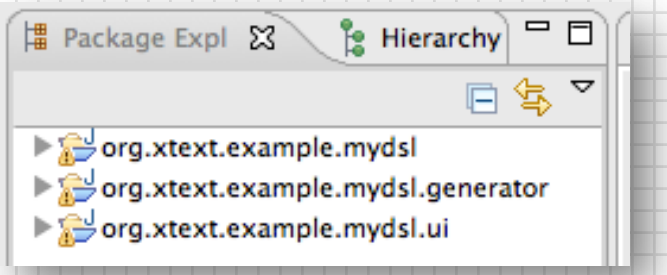
General principles (ctd.)

- Xtext project under Eclipse
  - Language name (for the DSL)
  - File Extension





General principles (ctd.)



- Files generated
  - The project itself
    - Grammar description
    - Constraints, behavior, …
  - User Interface `.ui`
    - Editor, « outline », code completion, …
  - Code generator `.generator`

Grammar Description

- Close to EBNF

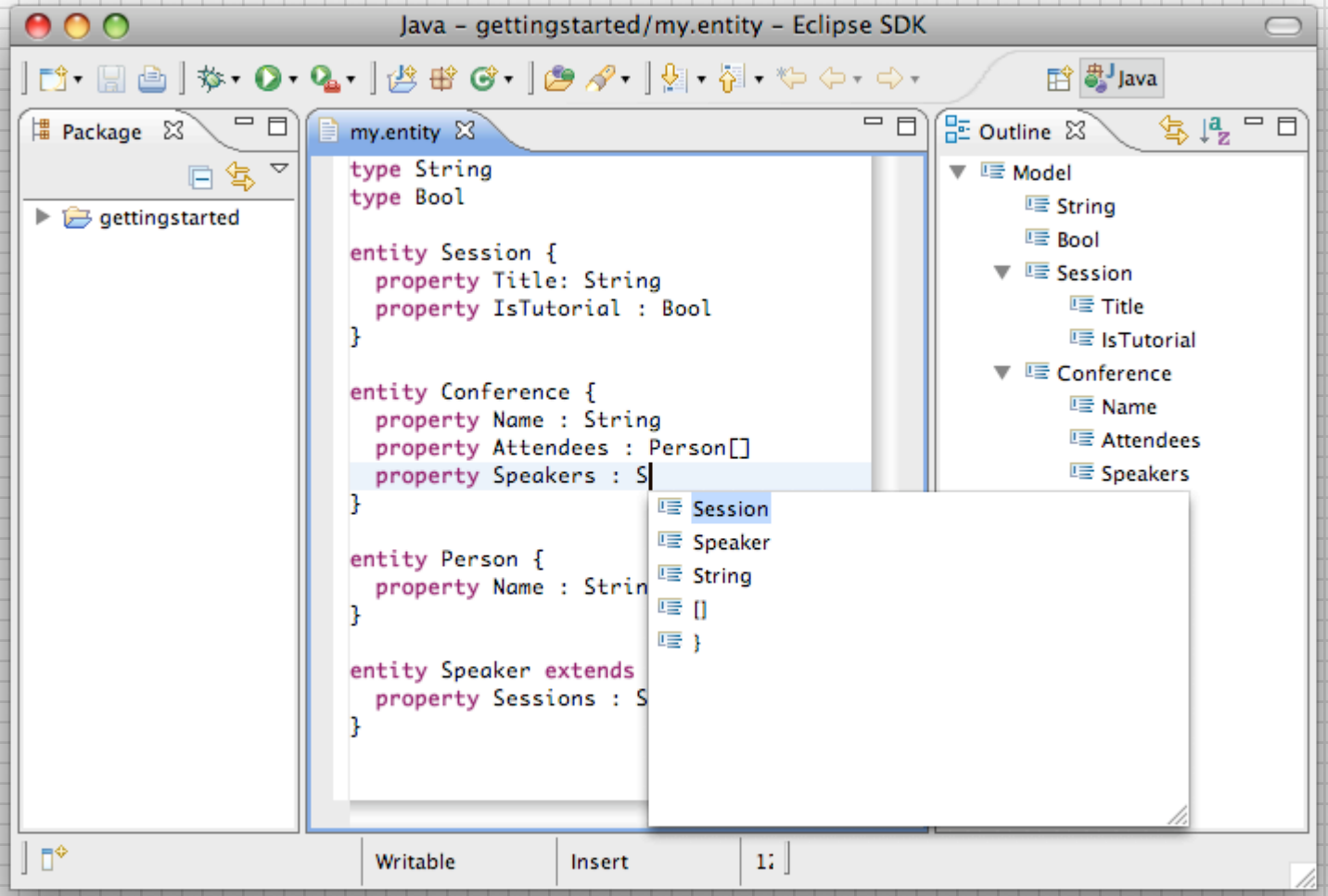
MyDsl.xtext

```
grammar org.xtext.example.MyDsl with org.eclipse.xtext.common.Terminals

generate myDsl "http://www.xtext.org/example/MyDsl"

Model :
    (imports+=Import)*
    (elements+=Type)*;

Import :
    'import' importURI=STRING;

Type:
    SimpleType | Entity;

SimpleType:
    'type' name=ID;

Entity :
    'entity' name=ID ('extends' extends=[Entity])? '{'
        properties+=Property*
    '}';

Property:
    'property' name=ID ':' type=[Type] (many?='[]')?;
```

Framework generation

- Using the generator
  - `GenerateXXX.mwe`
- Project launching
  - Either as an eclipse plug-in
  - Either as an eclipse application

Framework usage



Model Manipulation

- Writing code generator
  - Templates `.xpt`

```
«IMPORT entities»

«DEFINE main FOR Model»
  «EXPAND DAO::dao FOREACH this.types.typeSelect(Entity)»
  «EXPAND Entity::entity FOREACH this.types.typeSelect(Entity)»
«ENDDEFINE»

«ENDFILE»
«ENDDEFINE»

«DEFINE property FOR Property»
  private «this.type.name» «this.name»;

  public void set«this.name.toFirstUpper()»(«this.type.name» «this.name») {
    this.«this.name» = «this.name»;
  }

  public «this.type.name» get«this.name.toFirstUpper()»() {
    return this.«this.name»;
  }
«ENDDEFINE»
```

Model Manipulation (ctd.)

```
«IMPORT entities»

«DEFINE entity FOR Entity»
  «FILE this.name + ".java"»
    public class «this.name» {
      «EXPAND property FOREACH this.properties»
    }
  «ENDFILE»
«ENDDEFINE»

«DEFINE property FOR Property»
  private «this.type.name» «this.name»;

  public void set«this.name.toFirstUpper()»(«this.type.name» «this.name») {
    this.«this.name» = «this.name»;
  }

  public «this.type.name» get«this.name.toFirstUpper()»() {
    return this.«this.name»;
  }
«ENDDEFINE»
```

Content

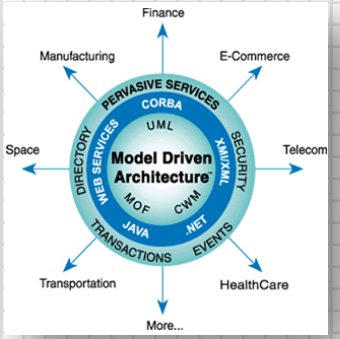
- Why learning MDE from code ?
- Model or code
- Metaprogramming
- Aspect Oriented Programming
- Model-Driven Engineering
- Self-Adaptive Systems

MDE Motivation

- Evolution of new technologies
- Separation between domain and architecture
- Algorithm stability
- Know-how Capitalization
  - Design pattern
  - Domain Objects
- => necessity to abstract from code

MDA

- *Model-Driven Architecture*
  - OMG
  - Notions of:
    - PIM: *Platform Independent Model*
    - PSM: *Platform Specific Model*
    - CIM: *Computation Independent Model*



MDE

- *Model-Driven Engineering*
  - More general than MDA
  - Less focus on architectures
  - Models as *first-class citizen*

Basic Concepts

- Models
- Metamodels and levels
- Language, syntax, semantic
- Transformations

Model

- Model
  - Abstraction of a system
    - Designed for a defined purpose
    - Usable to predict/simulate/validate
  - Description vs. Specification
    - Description => existing system
    - Specification => expected system

Model (ctd.)

- Objectives
  - Deal with systems complexity
  - SoC
  - Formalize intuitive elements
  - Communication (inside/outside the team)

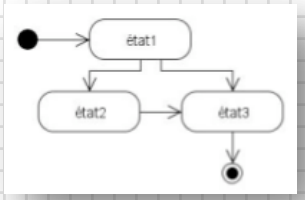
Model (ctd.)

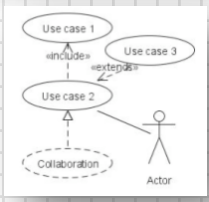
- Diverse and numerous
  - Different life cycle
  - Level of abstraction
  - Viewpoints

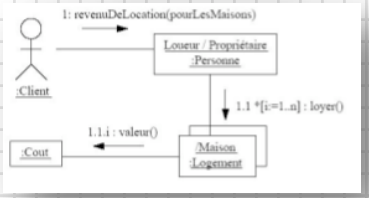
  => multiplicity of models

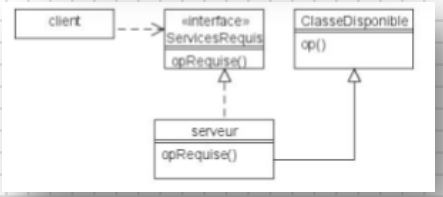
Model (ctd.)

- Example of UML
  - Behavioral models
  - Analysis models
  - Data models
  - Communications models
  - …





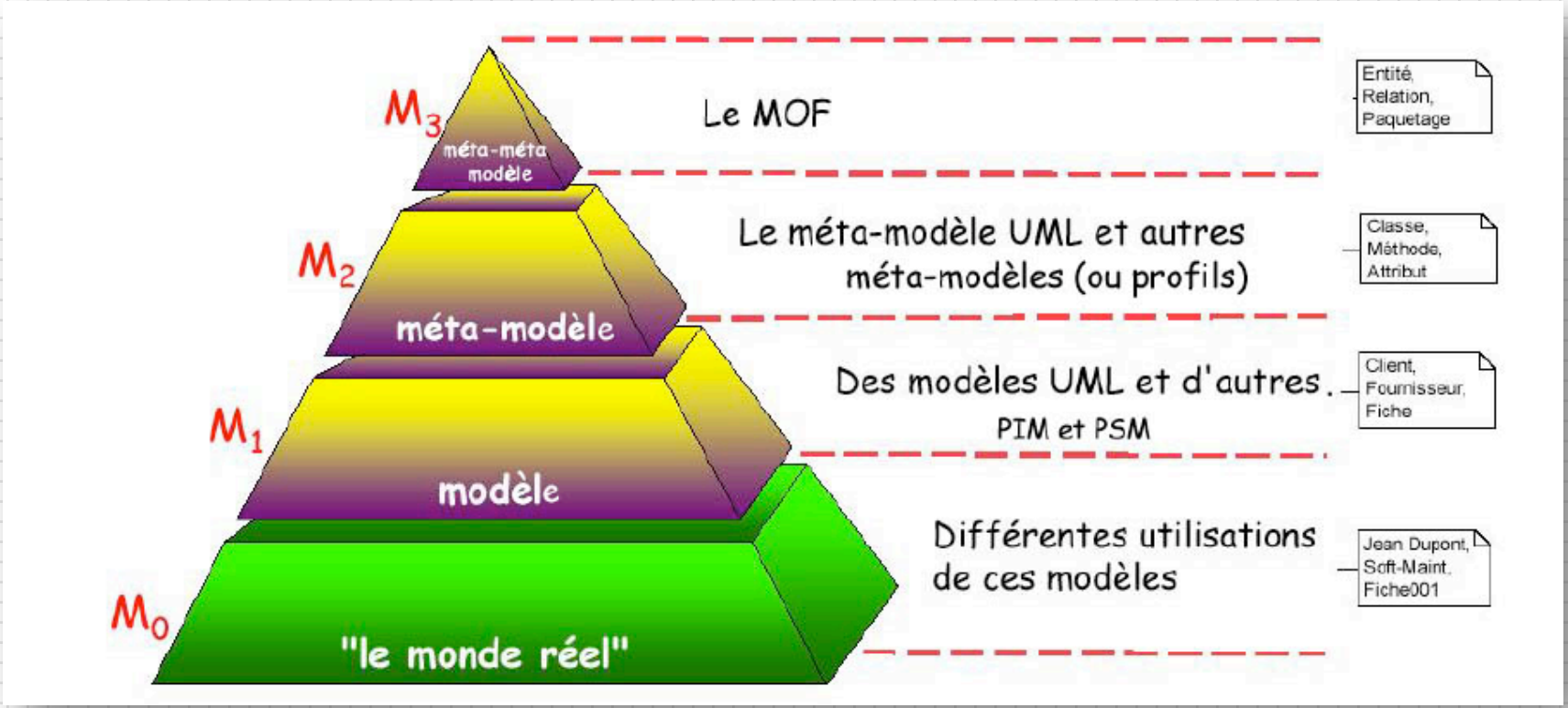




Model (ctd.)

- Benefits of a « formal » representation
  - => notion of conformity
  - => notion of metamodel

Metamodel and levels



Source : Jean Bézivin

Metamodel (ctd.)

- Examples
  - Programs
    - Process/Program/Grammar/EBNF
  - XML
    - Data/XML file/DTD/??
  - UML
    - Class/Class Diagram/UML/UML!!

Metamodel (ctd.)

- Instanciation / conformity
  - Class <-> Object
    - => instanciation
    - An object is a class instance
  - Model (M) <-> Metamodel (MM)
    - => conformity
    - A model is not an instance of a MM
    - A MM is not a model of the model

Metamodel (ctd.)

- OMG Vocabulary
  - MOF: Meta-Object Facility
  - Common MM
    - UML 2.0
    - CWM
    - QVT

Metamodel (ctd.)

- Comparison MOF/EBNF
  - BNF : Bakcus-Naur Form

```
-- Exemple des adresses US :
<postal-address> ::= <name-part> <street-address> <zip-part>
<name-part> ::= <personal-part> <last-name> <opt-jr-part> <EOL>
                | <personal-part> <name-part>
<personal-part> ::= <first-name> | <initial> "."
<street-address> ::= <opt-apt-num> <house-num> <street-name> <EOL>
<zip-part> ::= <town-name> "," <state-code> <ZIP-code> <EOL>
<opt-jr-part> ::= "Sr." | "Jr." | <roman-numeral> | ""
```

Metamodel (ctd.)

- Comparison MOF/EBNF (ctd.)
  - EBNF : Extended Bakcus-Naur Form

```
(* a simple program in EBNF - Wikipedia *)
program = 'PROGRAM' , white space , identifier , white space ,
          'BEGIN' , white space ,
          { assignment , ";" , white space } ,
          'END.' ;
identifier = alphabetic character , { alphabetic character | digit } ;
number = [ "-" ] , digit , { digit } ;
string = '"' , { all characters - '"' } , '"' ;
assignment = identifier , ":=" , ( number | identifier | string ) ;
alphabetic character = "A" | "B" | "C" | "D" | "E" | "F" | "G"
                     | "H" | "I" | "J" | "K" | "L" | "M" | "N"
                     | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
                     | "V" | "W" | "X" | "Y" | "Z" ;
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
white space = ? white space characters ? ;
all characters = ? all visible characters ? ;
```

« conforms to »

```
PROGRAM DEMO1
BEGIN
  A0:=3;
  B:=45;
  H:=-100023;
  C:=A;
  D123:=B34A;
  BABOON:=GIRAFFE;
  TEXT:="Hello world!";
END.
```

Metamodel (ctd.)

- Comparison MOF/EBNF (ctd.)

MOF	EBNF
DSL for models	DSL for syntax
Metamodel	Meta-syntax
To write models	To describe programming languages
« Initiated » by OMG and then normalized by ISO	Normalised by ISO

Language, syntax, semantic

- To start with…
  - Difference between data & information?

Data	Information
Representation of the information	Element of knowledge
Used to communicate	Interpretation of the data
e.g.: - June, 21 2013 - 2013 Summer day	Same information
e.g.: yersteday	Context dependency

Language, syntax, semantic

- General concepts
  - To communicate we need one (or more) language(s)
  - A model (M) is not a language
    - But can be the description of a language
  - A metamodel (MM) neither

Language, syntax, semantic

- A language consists in
  - A syntactic notation
  - A set of legal elements (combination of terms)
  - An interpretation for these termes

Language, syntax, semantic

- Syntax
  - Set of rules for the basic notation
- Semantic Domain
  - Set of interpretations for the notation
- Semantic
  - « mapping » from syntax to semantic domain

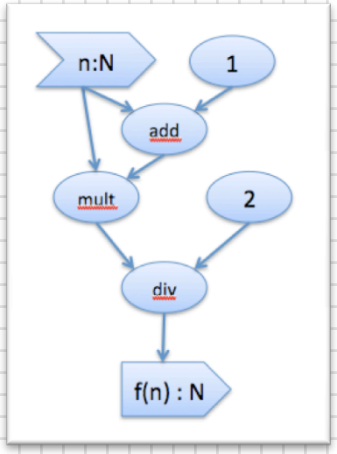
Examples

- Syntax

  <Exp>::=<Number>|
  <Variable>|
  <Exp>+<Exp>|…

- Semantic domain
  - Set of natural numbers
- Semantic
  - Associating a natural number to each expression

- Syntax



- Semantic domain
  - Set of traces
- Semantic
  - Trace log

Syntax

- Examples
  - Words, sentences, declarations, boxes, schemas, terms, modules, …
- Different kinds
  - Graphical
  - Textual
  - Abstract
  - Concrete
- Language for its representation
  - Textual => grammar
  - Graphical => graphs

Semantic

- Operational
  - Set of states and transitions between those states
- Denotational
  - Math. function that transfer from one state to the other
- Axiomatic
  - Logical properties on states

Illustrations

- Example: signification of « X = X+Y »
  - « at this point of the program X must contain the same value as X+Y »
  - X receive X+Y
  - What X+Y means?
  - => it depends on the semantic

Example of the MIU system

- Taken from [GEB,1989]
  - Problem: "can we produce MU?"
  - Rules
    - Considering an initial ***chain*** (MI)
    - And some ***production rules*** (or ***inference***)
      - The ***alphabet*** has only 3 lettres (M,I,U)
      - r1: if a chain finishes by I we can add U
      - r2: if a chain is of form Mx we can create Mxx
      - r3: III can be replaced by U
      - r4: UU can be eliminated
  - Implicit rule: we can only do what is authorized!

MIU (ctd.)

- Illustration of the rules
  - r1: I ending => we can add U
    - MI -> MIU
  - r2: Mx => Mxx
    - MU -> MUU
    - MUI -> MUIUI
  - r3: III can be replaced by U
    - UMIIIU -> UMUU
  - r4: UU can be removed
    - MUUUI -> MUI

MIU (ctd.)

- Problem: can we get MU from MI?
  - Go ahead and try for a while
  - chains can be seen as ***theorem***
  - Hence MU is a theorem that has to be proven
  - MI is an ***axiom***
  - Notion of ***derivation*** (example for MUIIU :)

MI	(axiom)
MII	(r2)
MIIII	(r2)
MIIIIU	(r1)
MUIU	(r3)
MUIUUIU	(r2)
MUIIU	(r4)

MIU (ctd.)

- MUIIU can be "***proven***"
- Some properties are "obvious": all theorems start with M, but no program can say it => difference between man and machine!
- *"it is possible for a machine to act unobservant, not for a human"*
- Difference between being *in* and *out* of the formal system
  - Playing with the rules => in
  - Asking why MU is not provable => out

Formal systems

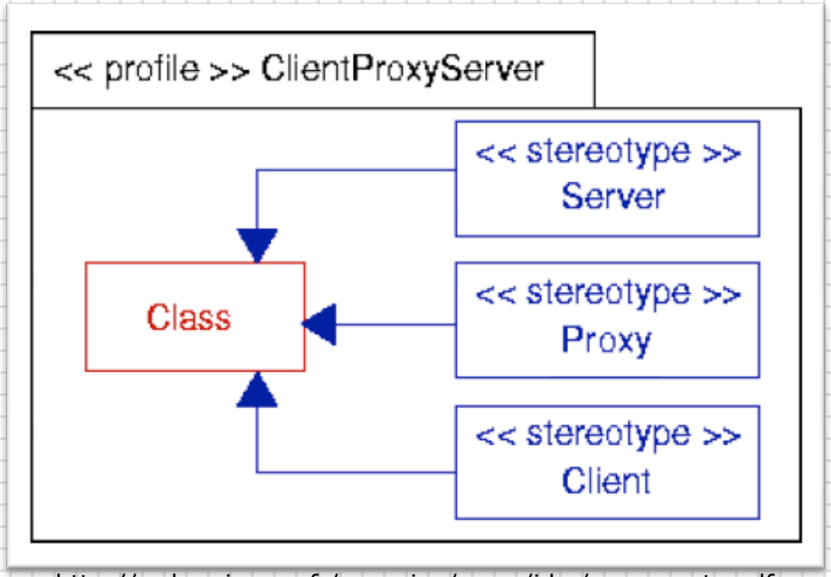
- Being able to determine whether an expression is correct (*well-formed*)
- Being complete (*completeness*)
  - Any correct expression should have a meaning
- Being *consistent*
  - An expression cannot have different meanings
- Both properties are complementary
- BUT you cannot have both! (Gödel theorem)

Formal verification

- Undecidability of the correctness proof
- Partial solution = test
  - Checking for errors
  - Not finding one doesn’t mean there is no one
- State machines
  - *model-checking*
  - Problem of combinatory explosion
- Formal proofs
  - *theorem proving*
  - More an helping system that a proof system

Example of UML profiles

- Possibility of language extension
  - Stereotypes
  - Tagged Values
  - OCL constraints



http://web.univ-pau.fr/~ecariou/cours/idm/cours-meta.pdf

Example of UML profiles (ctd.)

- profile for EJB
- SysML
- MARTE

Transformations

- Objectives
  - Models as « first level citizen »
  - Bridges between models
- Basic principles
  - Going from a source model (M_{in}) to a target model (M_{out})

Transformations (ctd.)

- Basic mechanisms
  - Take informations from M_{in}
  - Generate/modify informations into M_{out}
- Link with maths
  - Bijection, morphisms, …
  - Formal description

Transformations (ctd.)

- MDA vocabulary
  - CIM : *Computation Independent Model*
    - *E.g., UML Use Cases*
  - PIM : *Platform Independent Model*
    - *E.g., UML Class Diagram*
  - PSM : *Platform Specific Model*
    - *E.g., EJB model*
  - PDM : *Platform Description Model*
    - *E.g., AADL*

Transformations (ctd.)

- General vocabulary
  - M2M : *Model To Model*
    - *E.g., Class Diag. <-> E/R diag.*
  - M2T : *Model To Texte*
    - *E.g., Class Diag. -> Java code*

Transformations (ctd.)

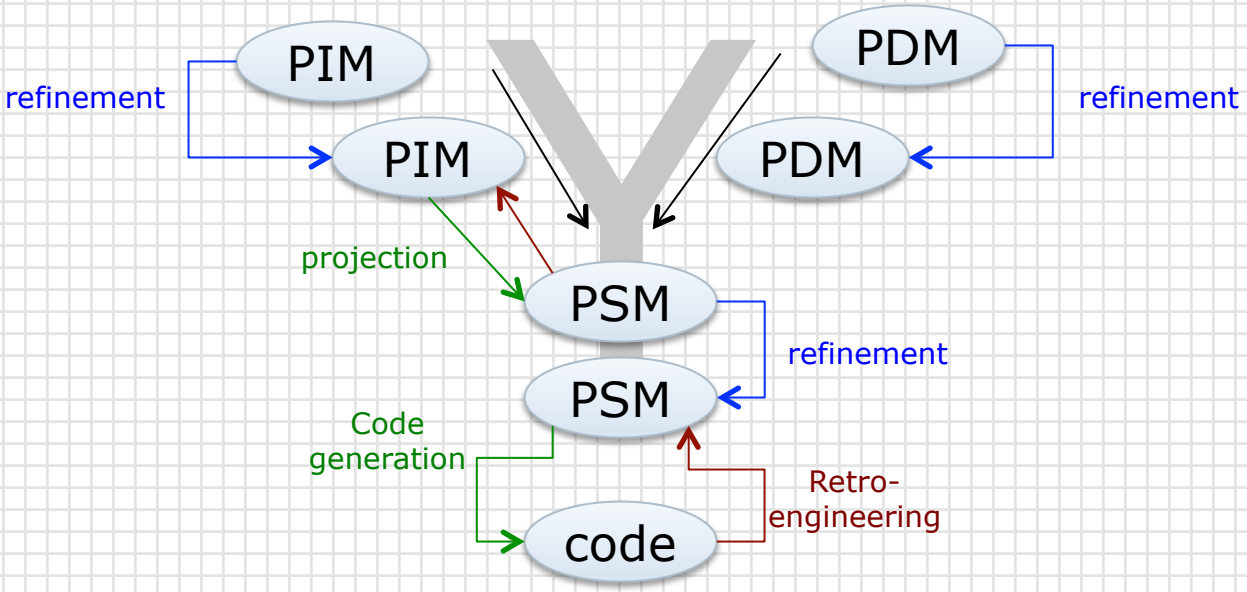
- What language for transformations?
  - M2T
    - Code generation
    - Mainly based on templates
    - E.g.: VTL, JET
  - M2M
    - Less popular
    - E.g.: QVT, ATL
    - « XSLT-like »

Transformations (ctd.)

- What language for transformations?
  - Endogenes
    - Same metamodel
    - E.g.: UML Statemachine / UML Classe diagrams
  - Exogenes
    - Different metamodels
    - E.g.: UML Statemachine / Petri nets

Transformations (ctd.)

- Usual term (« Y cycle »)



Transformations (ctd.)

- Quiz
  - Compilation: is it transformation?
  - Could you provide me an exogene transformation example?

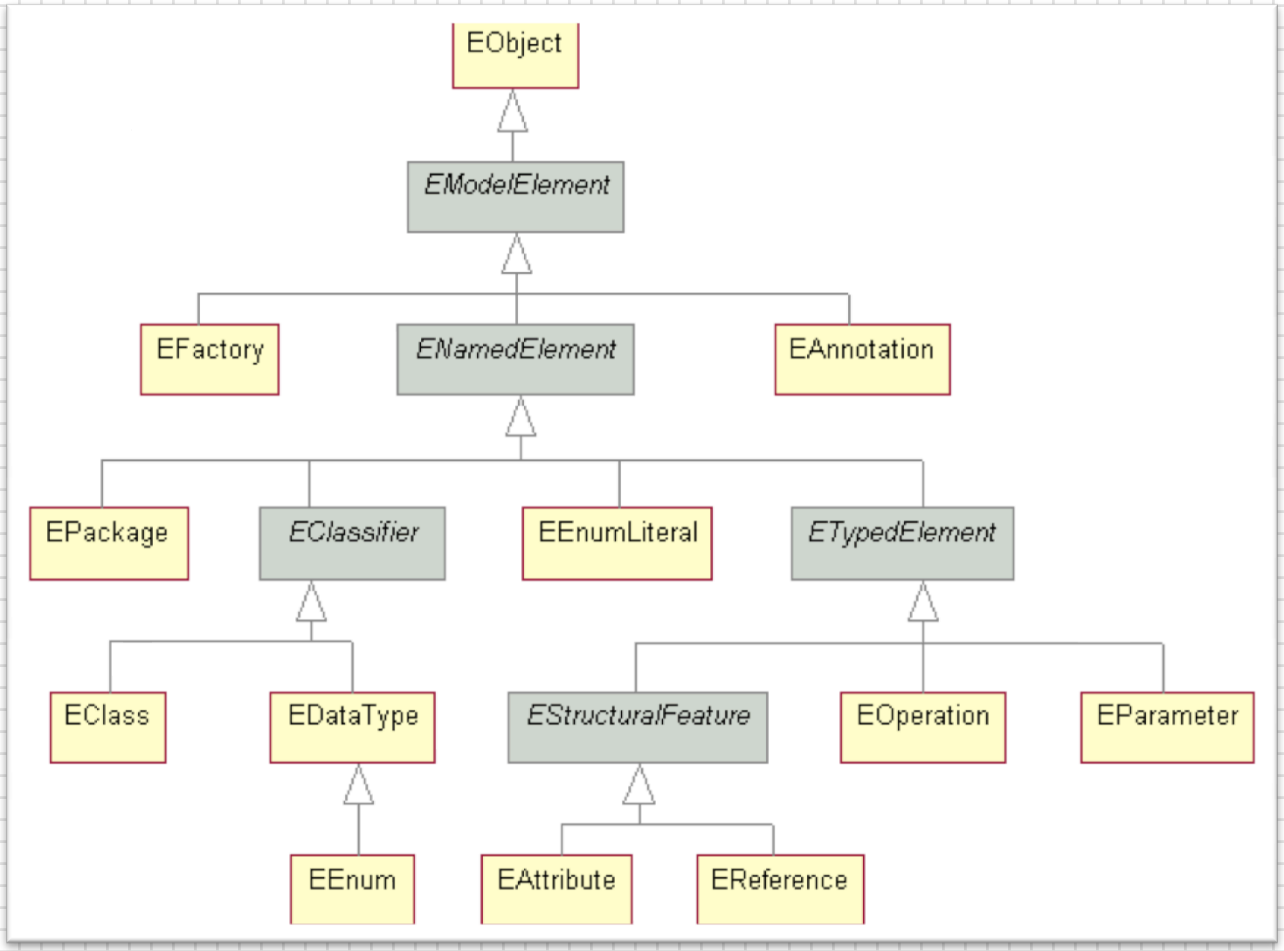
Basic tools

- Eclipse
  - EMF (*Eclipse Modeling Framework*)
    - Framework for manipulating models
    - Persistence management
    - meta-meta-model: Ecore
  - GMF (*Graphical Modeling Framework*)
  - GEF (*Graphical Editing Framework*)

Eclipse Modeling Framework

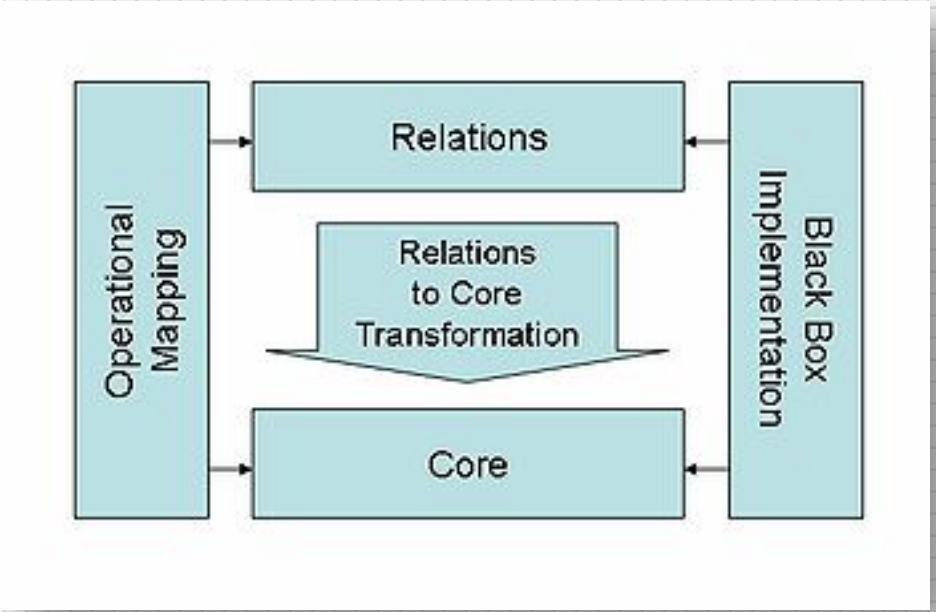
- Input files
  - UML (Ecore-compatible tools)
    - Plugin UML2
    - Papyrus
    - Rational (.mdl)
  - XMI
    - Pb. With compatibility
  - Java annotated
    - @model

Ecore


EObject
EModelElement
EFactory
ENamedElement
EAnnotation
EPackage
EClassifier
EEnumLiteral
ETypedElement
EClass
EDataType
EStructuralFeature
EOperation
EParameter
EEnum
EAttribute
EReference

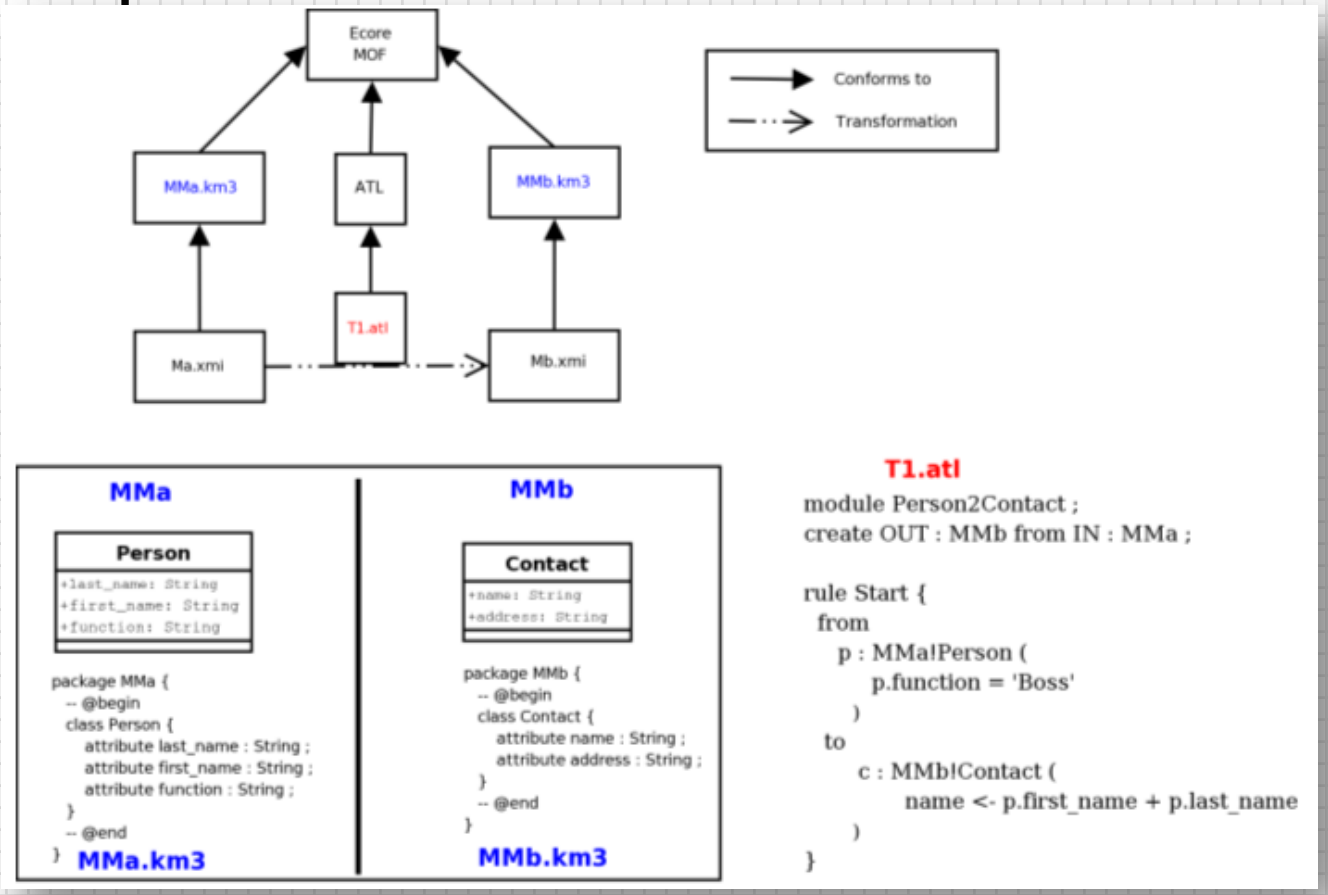
ATL (*Atlas Transformation Language*)

- French answer to QVT
- *Query/View/Transformation*
  - OMG standard



ATL (ctd.)

- Example 1

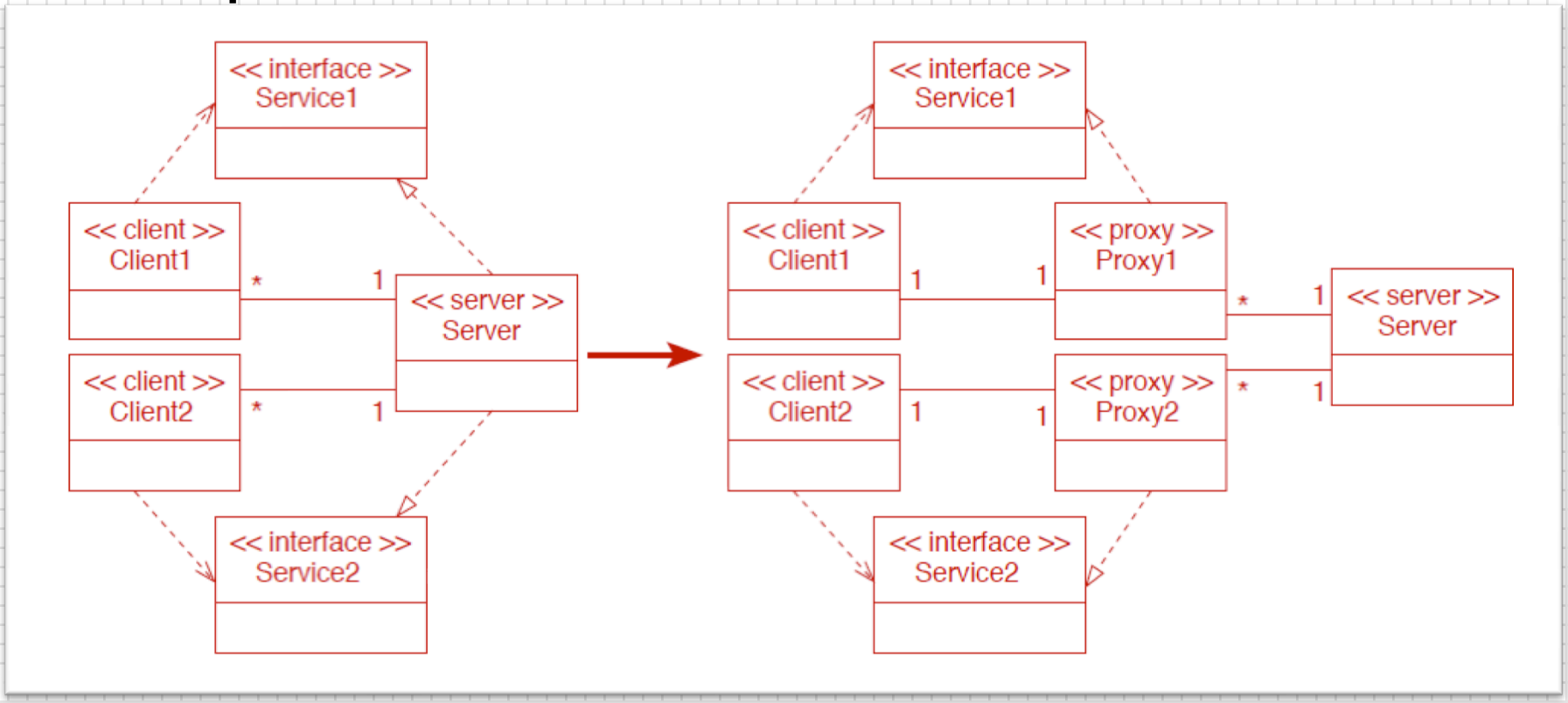


ATL (ctd.)

- Example 2
  - Taken from:
    - http://web.univ-pau.fr/~ecariou/cours/idm

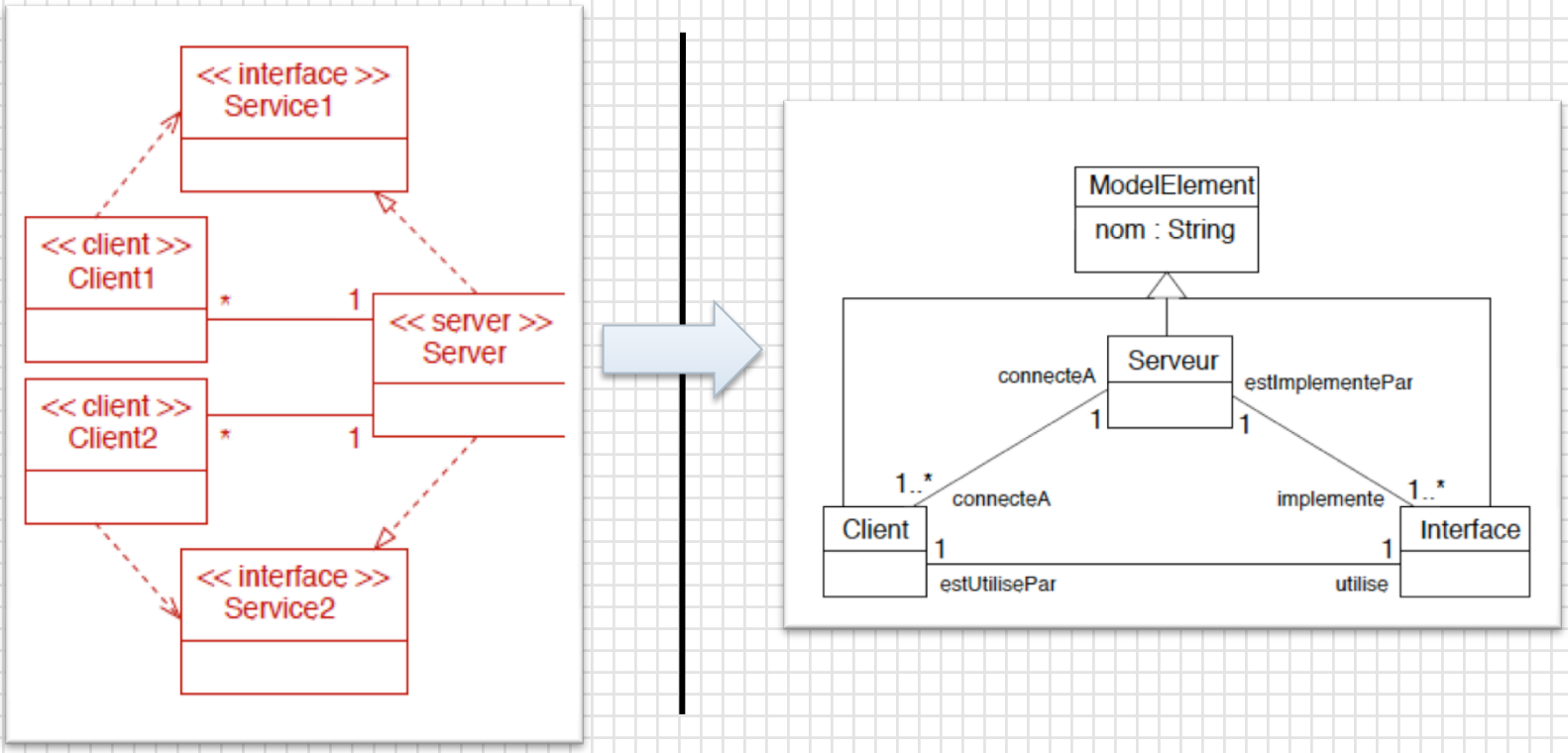
ATL example

- Expected transformation



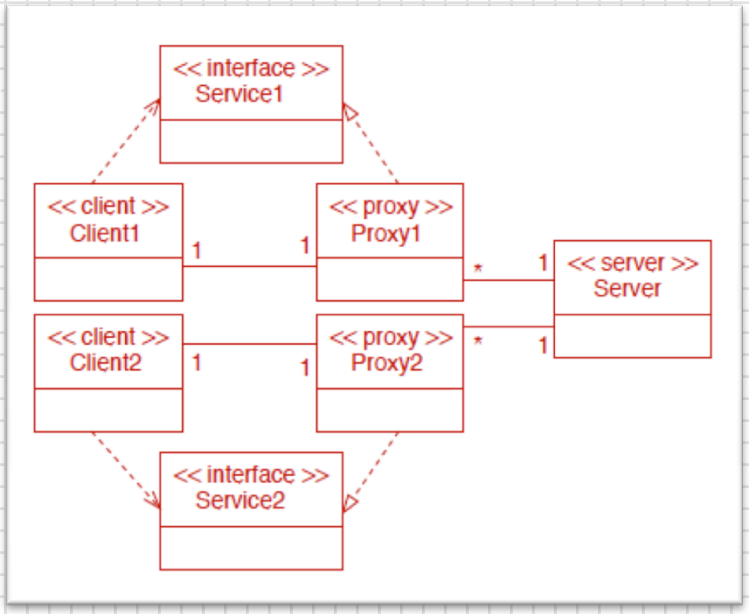
ATL example (ctd.)

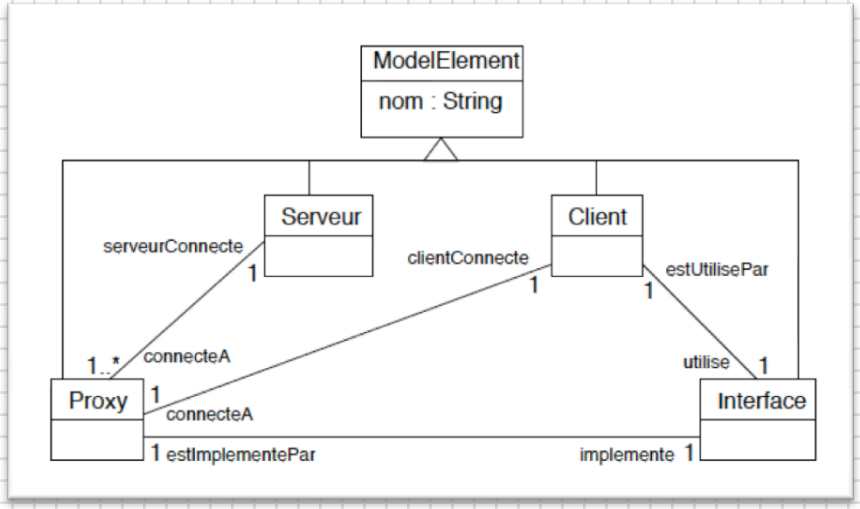
- MM definition



ATL example (ctd.)

- MM definition (ctd.)





ATL example (ctd.)

- Transformations description
  - Header

```
module TransformationProxy; -- Module Template
create OUT : ClientProxyServeur from IN : ClientServeur;
```

ATL example (ctd.)

- Transformations description
  - Server modification

```
rule LeServeur {
        from
                serveurSource : ClientServeur!Serveur
        to
                serveurCible : ClientProxyServeur!Serveur (
                        nom <- serveurSource.nom,
                        connecteA <- ClientProxyServeur!Proxy.allInstances()
                )
}
```

ATL example (ctd.)

- Transformations description
  - Clients modifications

```
rule Clients{
        from
                clientSource : ClientServeur!Client
        to
                clientCible : ClientProxyServeur!Client (
                        nom <- clientSource.nom,
                        utilise <- clientSource.utilise,
                        connecteA <- proxy
                ),
                proxy : ClientProxyServeur!Proxy (
                        nom <- clientSource.nom + ClientServeur!Serveur.allInstances().first().nom,
                        clientConnecte <- clientCible,
                        serveurConnecte <- ClientProxyServeur!Serveur.allInstances().first(),
                        implemente <- clientCible.utilise
                )
}
```

ATL example (ctd.)

- Transformations description
  - Interface modifications

```
rule Interfaces {
        from
                interSource : ClientServeur!Interface
        to
                interCible : ClientProxyServeur!Interface (
                        nom <- interSource.nom,
                        estUtiliseePar <- interSource.estUtiliseePar,
                        estImplementeePar <- (ClientProxyServeur!Proxy.allInstances()
                                -> select (p | p.implemente = interCible).first())
                )
}
```

MD* best practices

- Reference
  - MD* Best Practices, Markus Völter, 2008.

http://www.voelter.de/data/articles/DSLBestPractices-Website.pdf

Content

- Why learning MDE from code ?
- Model or code
- Metaprogramming
- Aspect Oriented Programming
- Model-Driven Engineering
- Self-Adaptive Systems

Ambient systems

- Definition (what I mean):
  - Human beings + components + interactions
    - Physical entities of distributed software
    - Capable of interacting with their environment
    - With autonomous behavior
    - Able to self-adapt to their context
- Problems:
  - Collection of components (interaction, emerging properties)
  - AI (decision taking, self-adaptation, etc.)
  - Environment and context (definition, monitoring)
  - Methods and tools for developing such systems

Ambient systems (IRIT lab.)

Anytime

24h/24
7/7
professionnelle
sphère privée
…

Intelligence

Analyse du contexte, auto-adaptation
dynamique aux situations.
Capacités de modélisation, de raisonnement, émotions…

Équipes IC3, LILaC, SMAC, SIG

Veille

Faculté du système à 'sentir' en permanence la présence et la localisation des objets, des appareils et des personnes.
Caméras, micros, radars, technologie des puces, lecteurs RFID, …

Equipes SAMoVA, VORTEX

Ubiquité

Capacité pour l'utilisateur d'interagir, n'importe où, avec des appareils interconnectés, des capteurs, des systèmes 'enfouis'.

Réseaux dynamiques, confiance, hétérogénéité, architectures distribuées
Equipes ASTRE, MACAO, SIERA

Anywhere

Intérieur/extérieur
bureau, domicile,déplacement
fixe ou mobile

Anything

entre PC
entre personnes (terminaux émergeants, PDA,…)

Interaction Naturelle

Accès aux services le plus simple possible:
reconnaissance gestuelle,
manipulation d'objets réels, …

Equipe ICHS

*Am*bient *I*ntelligent *E*ntities

- IRIT cross-cutting group
- AmIE project:
  - 2009/2014
  - Soft. & Hard. Platform
  - Based on scenarios
- 12 IRIT teams involved!
  - 18,5 equivalent people
  - 33 PhD students
- Competitivity groups
  - AESE (Aeronautics, RT&E)
  - Cancer, Bio, Santé

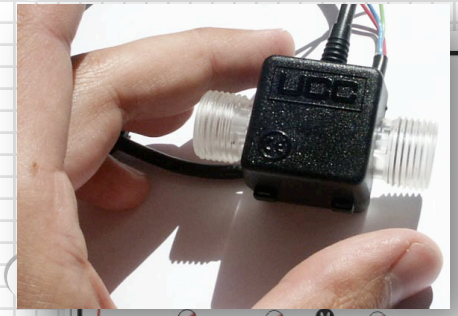
Scenarios

- Intelligent Campus
  - Social aspects
  - Financial aspects
  - Different viewpoints
    - Students
    - Professors
    - Staff
    - Technical services
- Home Supervision
  - Social aspects
  - Financial aspects
  - Different viewpoints
    - Patient
    - Medical staff
    - Relatives
    - Technicians
    - …

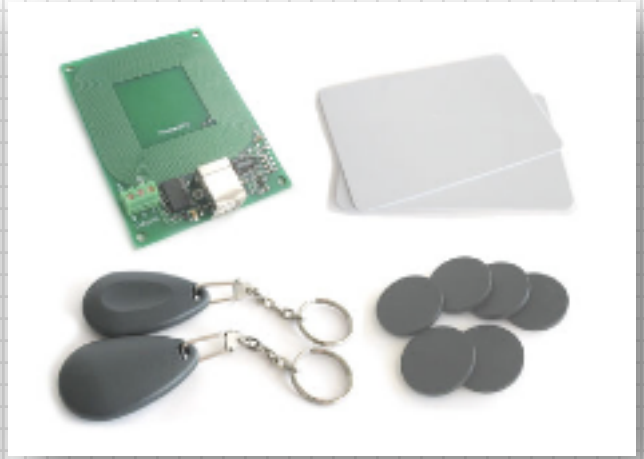
Technical Institute of Blagnac

- 4 Departments
  - Computer science (INFO)
  - Maintenance & Industry (GIM)
  - Networks & Telecoms (R&T)
  - 2A2M (Aide et Assistance pour le Monitoring et le Maintien à domicile)
- Laboratoiries: 3 CNRS labs
  - IRIT (Institut de Recherche en Informatique de Toulouse)
  - LAAS (Laboratoire d'Analyse et d'Architecture des Systèmes)
  - LAPLACE (Laboratoire Plasma et Conversion d'Energie)

Application context



[Ger@Home]



Technical details

- Static modules (nodes)
  - Sensors
    - Temperature (of the room)
    - Hydrometer
    - Luminosity
  - Webcam
  - Microphone & speaker (interaction)
- Mobile module (arm of the user)
  - Temperature (of the user)
  - Accelerometer

Technical details (cont.)

- System monitoring:
  - 1« Master pc » receiving all information from the nodes
  - Communication system: *zigbee*
  - *Crossbow Technology* Modules
  - *Imote2* Processor Radio Board (IPR2400) et *Imote2* Sensor Board (ITS400)
  - Nodes programming in .NET Micro Framework

Problems

- Number of potential actors involved
- Number of disciplines involved
- Importance of the human factor
- Critical systems (human lives)
- Security and privacy concerns
- Complex modeling and development process

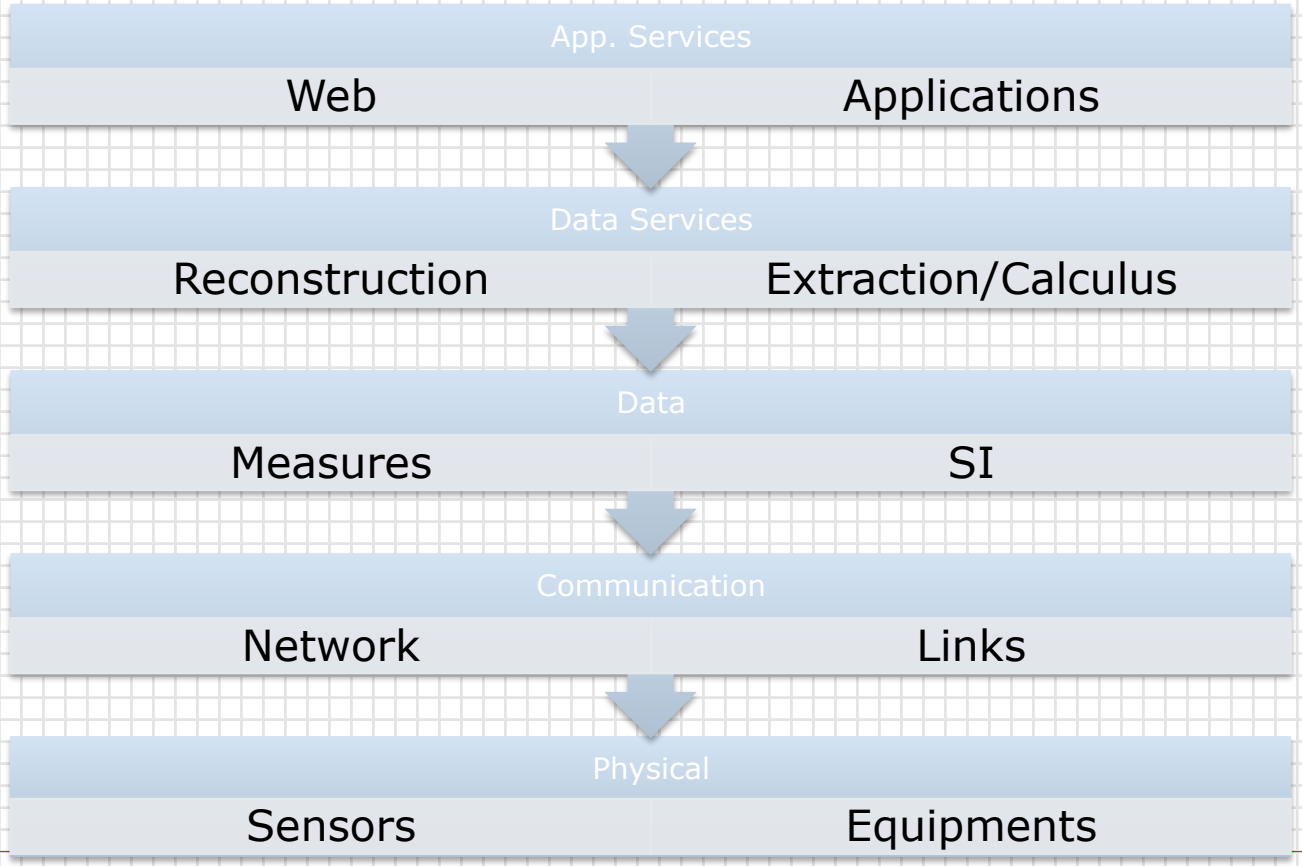
General architecture

- Context
  - Requirements
    - Vaguely defi
    - Very evolvin
  - Multiple Technologies
- Solution
  - Separate infrastructures and measures
  - Precise / Rewrite requirements (e.g. RELAX language)
  - Layer-based Architecture

S1': The synchronization process *SHALL* be initiated *AS EARLY AS POSSIBLE AFTER* Alice enters the room and *AS CLOSE AS POSSIBLE TO* 30 minute intervals thereafter.
ENV: location of Alice; synchronization interval.
MON: motion sensors; network sensors
REL: motion sensors provide location of Alice; network sensors provide synchronization interval

General architecture (cont.)

- Layer-based Architecture

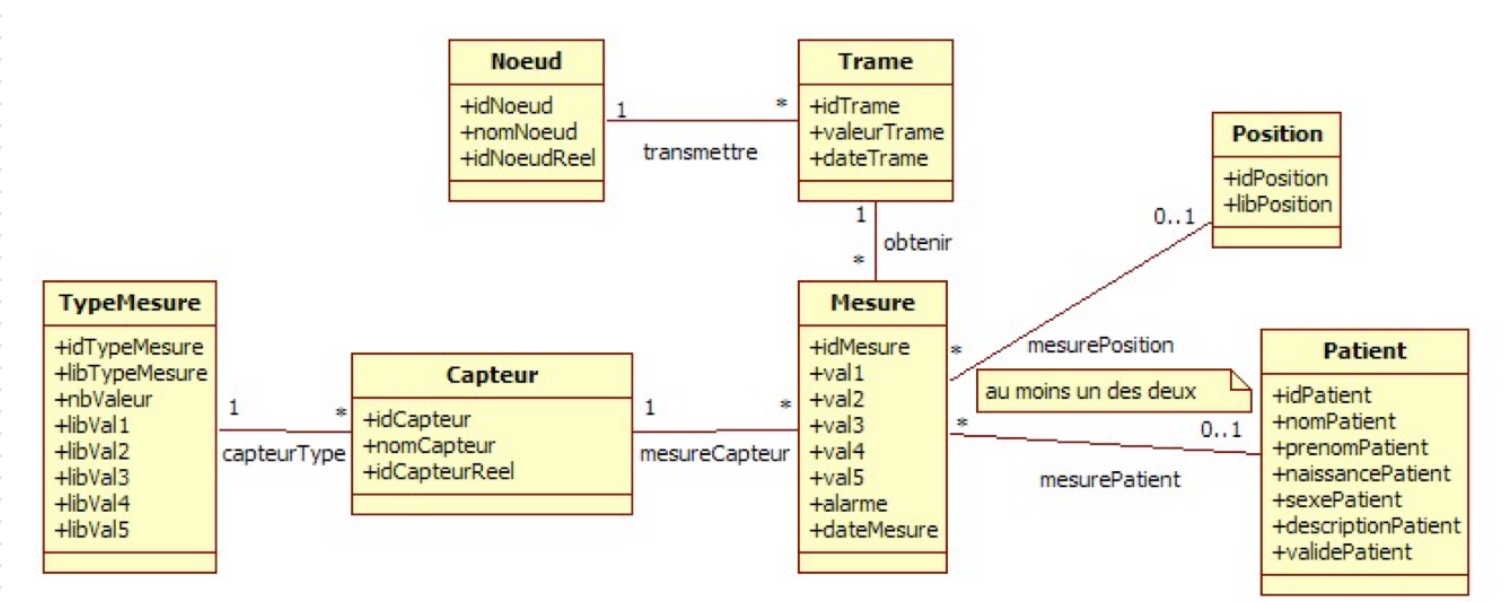


General architecture (cont.)

- Basics:
  - Abstraction of sensors
    - Data Schema independent from the hardware
  - Abstraction of processes
    - Web services definition
    - Encapsulating data access and processes
  - Abstraction of applications
- Advantages:
  - Greater evolutivity (physical, hardware and software)

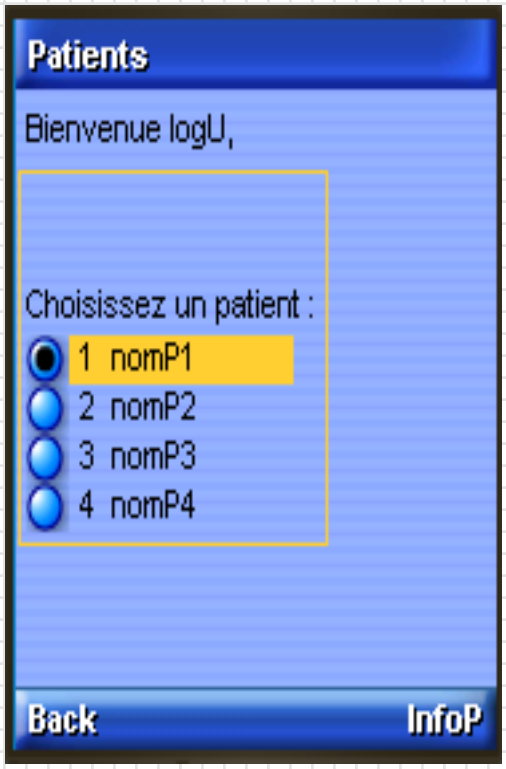
Experimentation

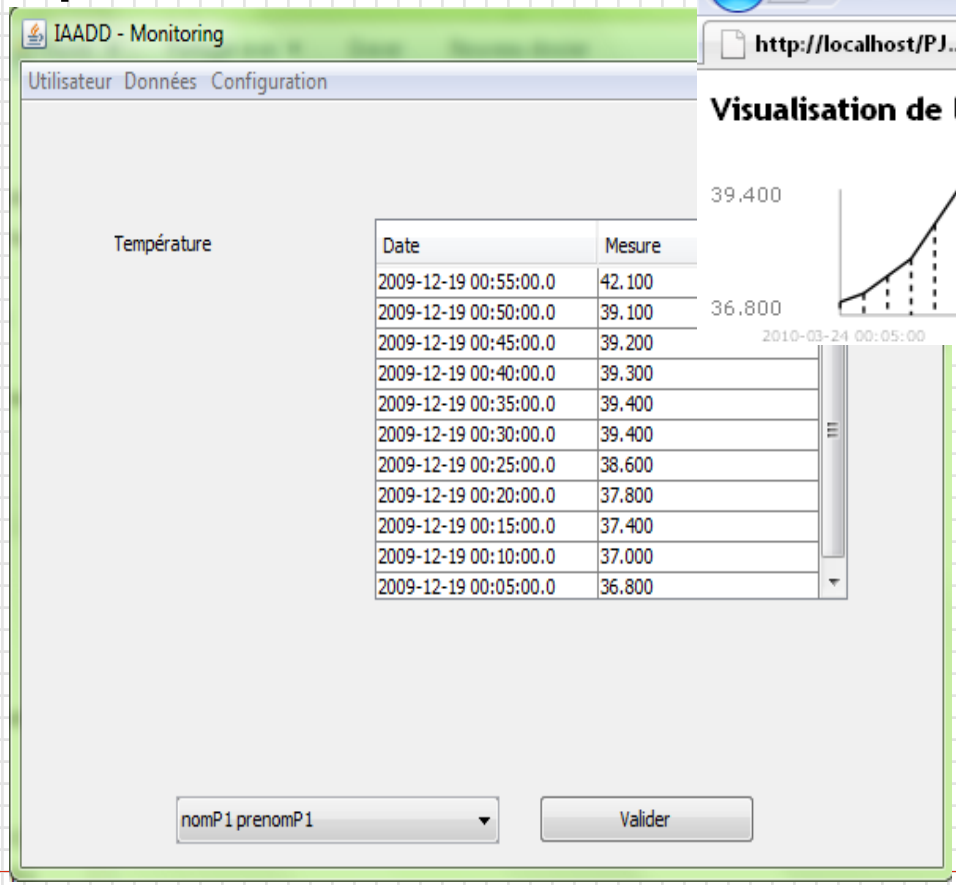
- Database



Experimentation (cont.)

- Context adaptation





Useful links

- Models
  - « On the Unification Power of Models », Jean Bézivin, Revue SoSyM, 4:2, 2005.
  - « Meaningful Modeling: What's the Semantics of *Semantics* », David Harel & Bernhard Rumpe, Revue IEEE Computer 37:10, 2004.
- MDE
  - OMG web site: http://www.omg.org/
- Tools
  - Kermeta (http://www.kermeta.org/)
  - ATL (http://www.eclipse.org/m2m/atl/)

Useful links (ctd.)

- Introspection & metaprogramming
  - http://www.polytech.unice.fr/~blay/ENSEIGNEMENT/LOG8/Meta-Programmer.pdf
  - http://www.enib.fr/info/gl2/cours/metaprog.pdf
  - Jean Bézivin. "On the Unification Power of Models", SoSyM 2005, 4(2).
- AOP
  - http://www.eclipse.org/aspectj/
  - Ruby par l'exemple (O'Reilly)
- Aspect-Oriented Modeling
  - http://aosd.net/
- Xtext
  - http://wiki.eclipse.org/Xtext/GettingStarted
  - http://www.eclipse.org/Xtext/documentation/
  - « On the Unification Power of Models », Jean Bézivin, Revue SoSyM, 4:2, 2005.