

Program Generators and Model Driven Architecture



Mark van den Brand

TU/e technische universiteit eindhoven

Program Generators and MDA

- *Program Generators*
- ATerms
- ApiGen
- Model Driven Architectures

Program Generators

- “State-of-the-art” technology: component based software development
 - Model Driven Architectures
 - Code Generation
 - Aspect Oriented Programming
 - Coordination Architectures
 - Design Patterns

Program Generators

- Automatic production of programs by means of other programs
 - A program generator reads meta-data and produces well-formed source code
 - Grammars
 - Database model
 - UML diagrams
 - A program generators makes your project “agile”

Advantages

- *Increase in productivity:*
 - generating tedious and boring parts of the code
 - code generators produce thousands of lines of code in seconds
 - changes are quickly propagated
 - Agile development

Advantages

- *Increase of Quality:*
 - bulky handwritten code tend to have inconsistent quality because increase of knowledge during development
 - bug fixes and code improvements can be consistently roled out using a generator
- *Increase of Consistency:*
 - in API design and naming convention
- *Single point of definition*
- *More design time*
- *Explicit design decisions*

Advantages

- *Architectural consistency:*
 - Programmers work within the architecture
 - Well-documented and -maintained code generator provides a consistent structure and approach
- *Abstraction: language-independent definition*
 - *Lifting problem description to a higher level*
 - *Easier porting to different languages and platforms*
 - *Design can be validate on an abstract level*

Disadvantages

- Code generator has to be written first
- Not always applicable
- There will always be some code that has to be hand written
- Generality can be a drawback, e.g.:
 - Databases must be well-designed and normalized
 - Grammars must be member of a specific class

Program Generators

- 10 code-generation rules
 1. Give proper respect to hand-coding
 2. Handwrite the code first
 3. Control the source code, e.g., CVS
 4. Make a considered decision about the implementation language
 5. Integrate the generator into the development process

Program Generators

- 10 code-generation rules (continued)
 6. Include warnings wrt modifying the generated code
 7. Make it friendly
 8. Include documentation
 9. Keep in mind that generation is a cultural issue
 10. Maintain the generator

Models of Program Generators

1. Code munging: given information in some input code, one or more output files are generated, e.g., scanner or parser
2. Inline-code expander: take source code with special markup code as input and creates production code in separate output file, e.g., inbedded SQL is replaced by C
3. Mixed-code generation: take source code with special markup code as input and replace this inline

Examples of Program Generators

- Programming environment generators
- API generators
- UML based generators
- Domain Specific Languages

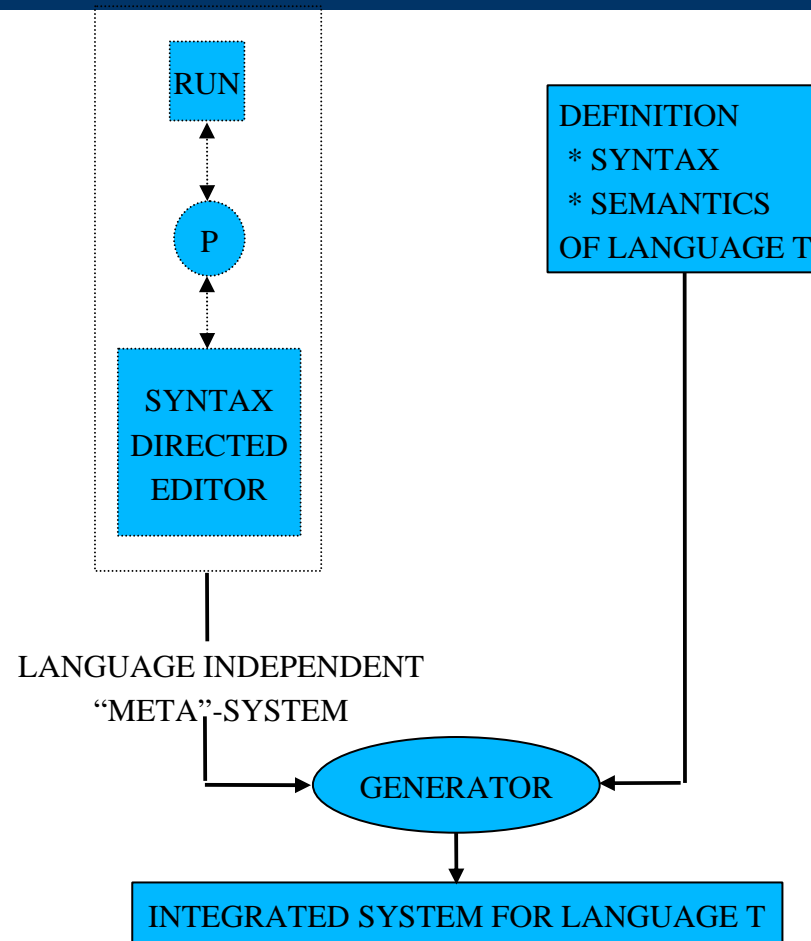
IDE Generator

- It is possible to develop a new programming environment for each (new) programming language.
 - Various, language dependent, programming environments have several parts in common, e.g., syntax directed and text editors, internal data structures, user-interface, help facilities, etc.

IDE Generator

- This observation leads to a *generator* for programming environments:
 - using a formal definition of the syntax and semantics of a language, a specific integrated environment can be generated for this language
 - generated environments for different languages have a uniform user-interface

IDE Generator



Literature on Program Generators

- “Code Generation in Action” by Jach Herrington
 - First and fourth chapter available at:
<http://www.codegeneration.net/cgia/download.html>

Program Generators and MDA

- *Program Generators*
- *ATerms*
- ApiGen
- Model Driven Architectures

ATerms

- Generic Term representation
 - Applicative, prefix terms
 - Maximal subterm sharing
 - Cheap equality test
 - Automatic generational garbage collection
 - Annotations (text coordinates, dataflow information, ...)
 - Concise and sharing preserving encoding for shipping:
 - Textual
 - binary

ATerms

- ATerms are used to represent and exchange (tree-like) data structures:
 - parse trees
 - abstract syntax trees
 - parse tables
- ATerm-library:
 - C version
 - JAVA version

ATerms

- The ATerm data type is defined as:
 - INT: (32-bits) integers
 - REAL: (64-bits) reals
 - APPL: function application, function symbol with zero or more ATerms as arguments
 - LIST: a list of zero or more ATerms
 - PLACEHOLDER: an ATerm representing the type of the placeholder, used for matching and constructing

ATerms

- The ATerm data type is defined as (continued):
 - BLOB: Binary Large data Objects, a length indication and a byte array of arbitrary (very large) binary data
 - a list of ATerm pairs (`label`, `annotation`) may be associated with an ATerm

ATerms

- Examples of ATerms:
 - Unquoted and quoted function application: `a`, `f(a)`, `"x"`, `f("Hello World\n")`
 - Integers and Reals: `1`, `2`, `3.14`, `1E-10`
 - Lists: `[1, 2.2, f("greetings")]`, `[]`
 - Placeholders: `<int>`, `<f(1, 2)>`
 - Annotations: `or(true, false) { [color, white] }`

ATerms

- Simple *match-and-make* paradigm:
 - *make* a new ATerm by providing a pattern, “and(<int>, <appl>)”, and filling in the holes
 - *match* an existing ATerm by comparing it with a pattern and decompose it according to the pattern
 - most important operations:
 - `ATerm ATmake(String p, ATerm a1, ..., ATerm an)`
 - `ATerm ATmatch(ATerm t, String p, ATerm *a1, ..., ATerm *an)`
 - `ATbool ATisEqual(ATerm t1, ATerm t2)`
 - `Integer ATgetType(ATerm t)`

ATerms

- Previous operations level one operations:
 - simple to use
 - high-level
 - inefficient
- Efficient level-two operations:
 - advanced usage
 - efficient
 - `ATgetArgument`, `ATmakeAppl`, `ATmakeInt`, **etc.**

ATerms

- Applications:
 - ASF+SDF Meta-Environment: an IDE for developing programming language definitions
 - mCRL2: a protocol verification tool uses ATerms to represent the state space.
 - Stratego compiler uses ATerms to represent abstract syntax trees.
 - TOM, a pattern matching compiler.
 - Semantic web ontology

ATerms

- More information:
 - www.aterm.org
- More reading:
 - M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier (2000). "Efficient Annotated Terms" *Software -- Practice & Experience* 30:259--291

Program Generators and MDA

- *Program Generators*
- *ATerms*
- *ApiGen*
- Model Driven Architectures

ApiGen

- Good properties of ATerms:

- Simple concept
- Simple and complete API
 - make and match
 - read and write
- Efficient



*Application Programmers
Interface (Library)*

- Bad properties of ATerms

- not typed (APPL, LIST, INT, REAL, BLOB)

ApiGen

- **Bad ATerm programming:**

```
ATerm bool1, bool2, bool3, int4;
bool1=ATparse("and(true, false)");
bool2=ATparse("and(true, <term>)");
bool3=ATparse("and(false, <int>)");
if (ATmatchTerm(bool2, bool1, &int4)) {
    ATprintf("%t", int4);
}
else {
    if (ATmatchTerm(bool3, bool1, &int4)) {
        ATprintf("%t", int4);
    }
}
```

ApiGen

- Problems are recognized:
 - All ATerms are alike
 - no static well-formedness checks possible
 - debugging extremely hard
 - maintenance (changing something) is tedious
 - Most applications work on a specific *signature* of terms (or signature of term patterns)
- Problems can be solved by generating an API on top of ATerms

ApiGen

- ADT (abstract data type): ATerm signature:

```
[  
  [Bool, true , true],  
  [Bool, false, false],  
  [Bool, and , and <fieldName (fieldType) > , <rhs (Bool) >] ,  
  [Bool, or , or(<lhs (Bool) >, <rhs (Bool) >)]  
]
```

<fieldName (fieldType) >

Sort

Name

Pattern

ApiGen

- `adt-to-{c,java}`
 - Generate an API from an ADT
 - ATerm library “under the hood”
 - Typed access functions in the API
 - constructors
 - getters and setters
 - predicates

ApiGen

– Example:

– `adt-to-c -i Bool.adt -o Bool`

– this generates a `Bool.h` file containing:

```
...
Bool makeBoolFalse();
Bool makeBoolAnd(Bool lhs, Bool rhs);
...
ATbool isValidBool(Bool arg);
inline ATbool isBoolTrue(Bool arg);
...
ATbool hasBoolLhs(Bool arg);
Bool getBoolLhs(Bool arg);
Bool setBoolLhs(Bool arg, Bool lhs);
...
```

Constructors

Checkers

Getters and Setters

ApiGen

- The file `Bool.c` is also generated:

```
Bool BoolFromTerm(ATerm t)
{ return (Bool)t; }
ATerm BoolToTerm(Bool arg)
{ return (ATerm)arg; }
...
Bool makeBoolTrue()
{ return (Bool)(ATerm)ATmakeApp10(afun0); }
...
ATbool hasBoolLhs(Bool arg)
{ if (isBoolAnd(arg)) { return Attrue; }
  else if (isBoolOr(arg)) { return ATtrue; }
  return ATfalse;
}
...
```

Conversion
functions

afun0 =
true

ApiGen

- Example:

```
adt-to-java -i Bool.adt -o Bool
```

- generates among others a `Bool_AndImpl.java` file:

```
...  
public Bool getLhs()  
{  
    return (Bool) this.getArgument(index_lhs);  
}  
  
public Bool setLhs(Bool _lhs)  
{  
    return (Bool) super.setArgument(_lhs, index_lhs);  
}  
...
```

ApiGen

Programming against the C API yields:

```
#include <bool.h>
Bool bool, lhs, rhs;
bool = makeBoolAnd(makeBoolTrue(), makeBoolFalse());
lhs = getBoolLhs(bool);
rhs = getBoolRhs(bool);
ATprintf("%t", (ATerm) bool);
```

Executing this program yields: `and(true, false)`

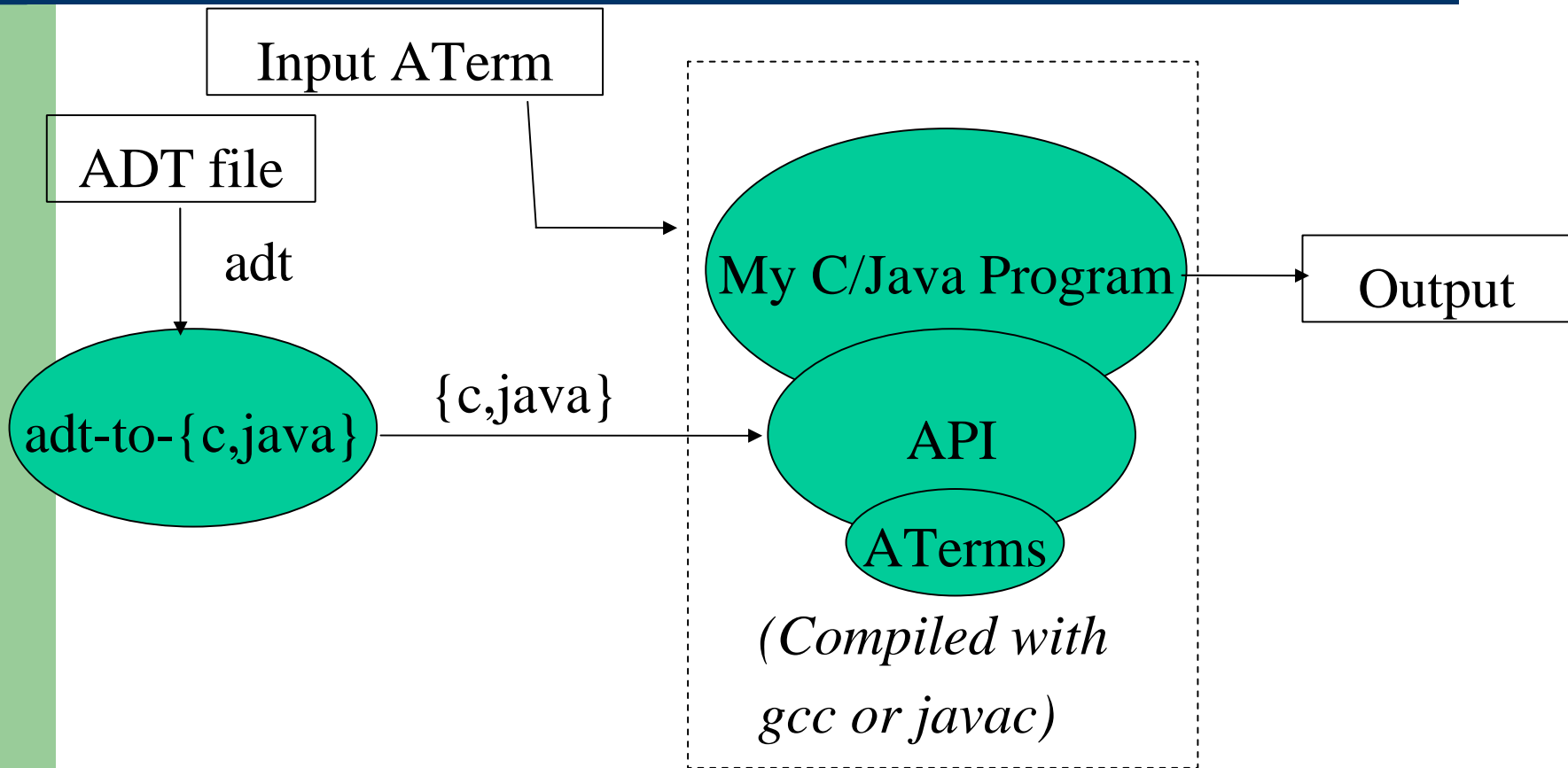
ApiGen

- Generated C code “uses” C ATerm library
 - Maximal subterm sharing via ATerm library
 - Automatic generational garbage collection
- Fully type-safe via opaque types

ApiGen

- Generated Java code “uses” Java ATerm library
- Maximal subterm sharing via a Shared Object Factory:
 - Efficient hashing
- Extra functionality:
 - Reading and writing of linearized representations
 - Generic tree traversal
 - Pattern matching

ApiGen



ApiGen

- Using `adt-to-c` we generated a C library to manipulated parse trees:

```
[[ParseTree, top, parsetree(<top(Tree)>, <amb-cnt(int)>)],  
  
 [Tree, appl, appl(<prod(Production)>, <args(Args)>)],  
 [Tree, char, <character(int)>],  
 [Tree, lit, lit(<string(str)>)],  
 [Tree, amb, amb(<args(Args)>)],  
  
 [Production, default, prod(<lhs(Symbols)>, <rhs(Symbol)>, <attributes(Attributes)>)],  
 [Production, list, list(<rhs(Symbol)>)],  
  
 [Attributes, no-attrs, no-attrs],  
 [Attributes, attrs, attrs(<attrs(Attrs)>)],  
 ...]
```

ApiGen

- Given this parse tree ADT:
 - ± 500 lines header-code is generated
 - ± 5300 lines c-code is generated

Hand-written function

```
ATbool PT_isTreeOpt (PT_Tree tree)
{
    if (PT_hasTreeProd(tree)) {
        PT_Symbol rhs = PT_getProductionRhs (PT_getTreeProd(tree));
        if (PT_isSymbolCf(rhs) || PT_isSymbolLex(rhs)) {
            rhs = PT_getSymbolSymbol(rhs);
        }
        if (PT_isSymbolOpt(rhs)) {
            return ATtrue;
        }
    }
    return ATfalse;
}
```

API generated functions

ApiGen

- Except for the parse tree API we generated given ADTs APIs for:
 - The structure editors, graphs, parse tables, etc.
- On top of the parse tree API a number of language specific API are developed

ApiGen

- ApiGen allows us type-safe ATerm programming
- Less bugs during development
 - static well-formedness of terms
- Maintenance becomes more easy
 - Change the ADT and the API changes along
- If you program against the ATerm library use ApiGen

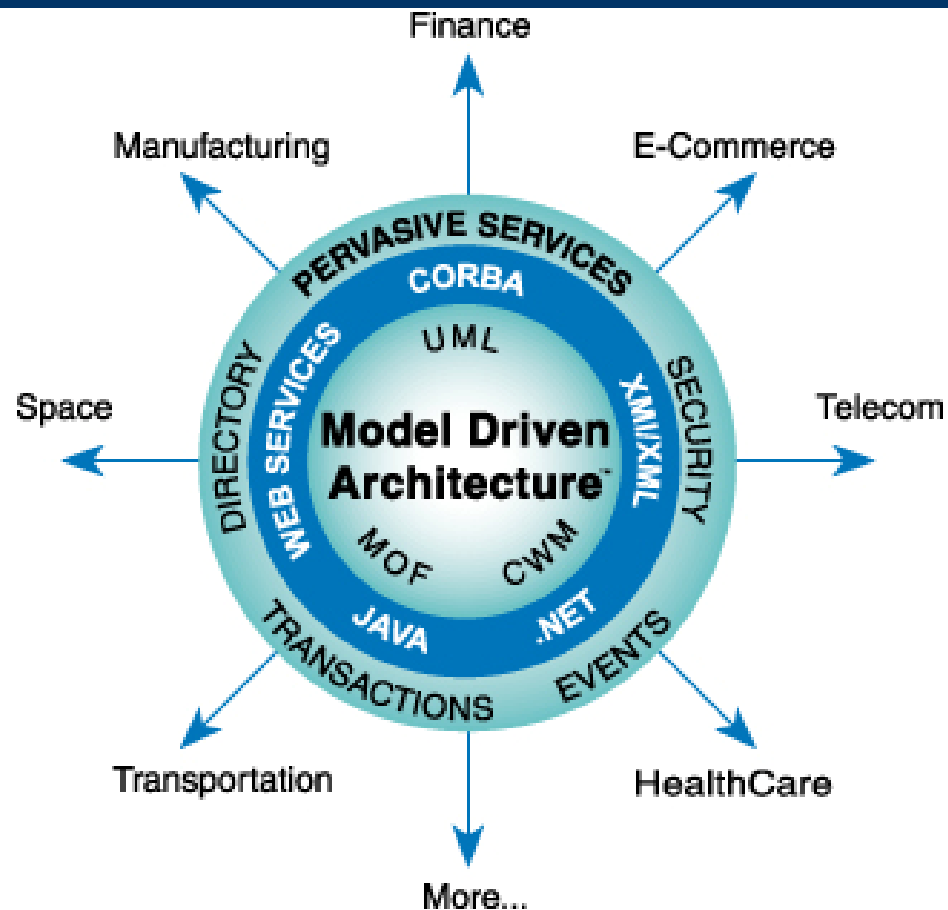
ApiGen

- Further reading:
 - H.A. de Jong and P.A Olivier. "Generation of abstract programming interfaces from syntax definitions"
Journal of Logic and Algebraic Programming, 2003
 - M.G.J. van den Brand, P.E. Moreau, and J.J. Vinju.
A generator of efficient strongly typed abstract syntax trees in Java.
IEE Proceedings - Software, 2005

Program Generators and MDA

- *Program Generators*
- *ATerms*
- *ApiGen*
- *Model Driven Architectures*

Model Driven Architectures



MDA

- Today's Software Environment:
 - Worldwide distributed systems
 - Heterogeneous platforms, languages, and applications
 - Increasing interconnectivity within and between companies
 - New technologies: XML, .NET and web services

MDA

- **Heterogeneous platforms and languages**
 - Programming languages
 - ~3 million COBOL programmers
 - ~1.6 million VB programmers
 - ~1.1 million C/C++ programmers
 - Operating systems
 - Unix, MVS, VMS, MacOS, Windows (all 8!), PalmOS...
 - Windows 3.1: it's still out there!
 - Embedded devices (mobile, set-top, etc.)

MDA

- Good News
 - Increased standarization
 - Internet protocols, SQL, UML
 - Increased openness
 - Linux, apache, etc.
 - Less custom specific development
 - Component reuse, ERP applications

MDA

- Bad news
 - Legacy applications and databases
 - ERP applications that are difficult to adapt
 - Multiple, competing middleware
 - Develop software for the future: adaptable to future modifications

MDA

- MDA is a more sophisticated way of using UML
- Raising level of abstraction:
 - General trend
 - Already well-established for front and back ends
 - WYSIWYG GUI modeling and data modeling
 - Hand coding no longer predominates
 - Tuning allowed

MDA: what is it?

- Sophisticated code generator based on UML diagrams (OMG claim!)
- High level of abstraction
- Restricted hand coding
- Programming language and platform independent!?!)
- The ultimate solution!?!)

MDA

- Informal UML models provide
 - Informal modeling
 - Used to sketch out basic concepts
 - Advantages over other informal diagram techniques: it has some form of semantics
 - Not suited for code generators and interpretation
 - Analogously informal text can not be compiled and executed as 3GLs

MDA

- Formal UML models provide
 - Precise:
 - Precision and details are *not* the same
 - Computationally complete
 - Missing properties and unresolved references are not acceptable
 - 3GL analogy ...
 - Incomplete programs can not be compiled

MDA: how does it work?

- Platform Independent Model (PIM) in UML is developed by architect, no assumptions
 - on platform
 - on programming language
 - on databases
 - on architecture (2-tier vs 3-tier)
- High level of abstraction

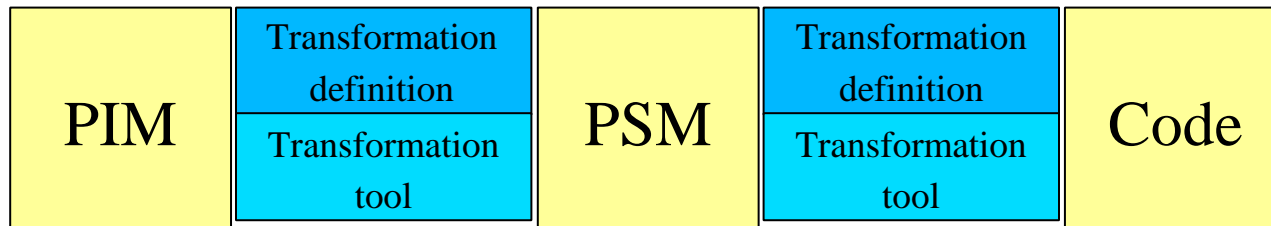
MDA: how does it work?

- PIM model is mapped to XMI, XML representation of UML
- PIM model is transformed into Platform Specific Model (PSM)
- Architectural decisions are resolved/instantiated
 - 2 tier vs 3 tier
 - CORBA
 - .NET

MDA: how does it work?

- The architecture implementation contains a series of declarative XML-based templates that generate “PSM” code
- Template resolves architectural issues for a certain layer
- Layers can be exchanged with other ones
- Mechanisms to provide architecture code extensions

MDA: how does it work?



- Code can be:
 - Enterprise Java Beans
 - Web
 - SQL

MDA: does it work?

- Transformations on XML level via XSLT
- Transformations via Java programs
- Until now only static aspects (class diagrams) are used for code generation
- Lots of hand coding involved
- Efficient?

MDA

- Various MDA implementations:
 - Commercial: OptimalJ from CompuWare
 - Open Source: Generative Model Transformer (<http://www.eclipse.org/gmt>)
- Further reading:
 - *MDA Explained* by Kleppe, Warmer and Bast
 - *MDA with Executable UML* by Raistrick, Francis, Wright, Carter and Wilkie