Domain-Specific Modeling: No one-size-fits-all

6 October 2005

Dr. Juha-Pekka Tolvanen
MetaCase

Contingency theory and software development

- Diversity due to
  - type of systems built
  - organizations
  - cultures
  - technology (that keeps evolving)
  - tools, etc.
- Most general purpose modeling languages do not recognize the diversity
- Contingency theory advocates for flexible languages (no single language gives best result in all situations)
  - IFIP WG conferences (Olle et al. 1982, -83, -86, -88)
  - Empirical studies show that companies prefer own methods
    - 2/3 use internal, home-grown methods, Russo et al., Fitzgerald
  - Laboratory studies show that developers understand and use methods differently
    - Extend, give new meanings, create own interpretations etc. for modeling constructs (e.g. in studies by Wijers, Verhoef)

© 2005 Juha-Pekka Tolvanen / MetaCase

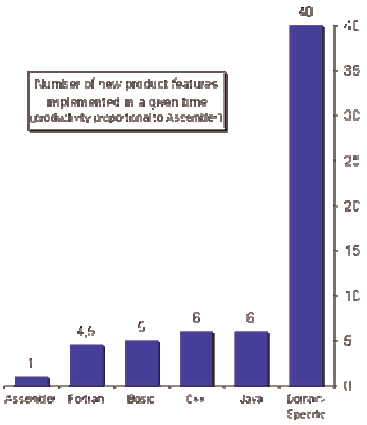
Fixed language challenge

- Fixed, general purpose, modeling languages have not made models 1st class development artifacts
  - IDEF, SSADM, Express, Merise, Euromethod, SDL, UML, SDM, ER etc.
    - With some exceptions in specific domains with SDL, schema design, Labview, etc.
- Model-Driven Development sets new requirements for languages
  - To enable code generation, testing, configuration, simulation, requirements validation, model reuse, etc.
  - Current languages offer only modest possibilities
    - It's hard to use general purpose solutions to automate specific things
- To add value modeling should **save time** and **improve quality**

© 2005 Juha-Pekka Tolvanen / MetaCase

How languages contribute to productivity and quality?

- "The entire history of software engineering is that of the rise in levels of abstraction"
- New programming languages have not increased productivity
- UML and visualization of code have not increased productivity
- Abstraction of development can be raised above current level...
- ... and still generate full production code (and ignore it!)



*Software Productivity Research & Capers Jones, 2002

© 2005 Juha-Pekka Tolvanen / MetaCase

Let's see examples from different domains...

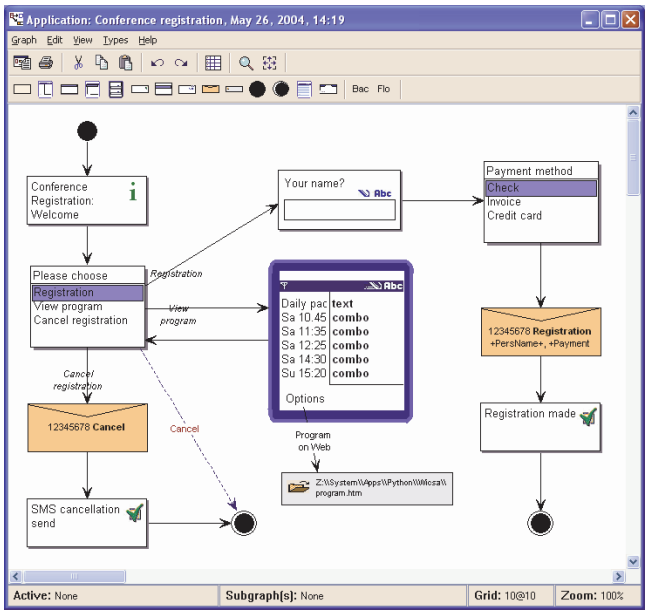
- Smartphone applications
- Telecom service creation
- eCommerce marketplace
- Web applications
- IP telephony services
- Applications in microcontroller
- Workflow applications

© 2005 Juha-Pekka Tolvanen / MetaCase

Case1: Enterprise apps in smartphones

- Symbian/Series 60 for enterprise application development
- Platform provides basic services
- Modeling language to define application logic using basic widgets and services
- Code generator produces 100% of implementation
- Complete chain from model to running app

© 2005 Juha-Pekka Tolvanen / MetaCase



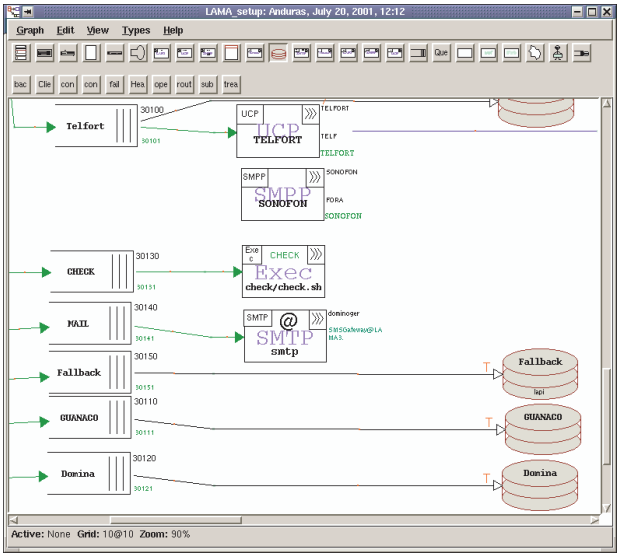
© 2005 Juha-Pekka Tolvanen / MetaCase

© 2005 Juha-Pekka Tolvanen / MetaCase

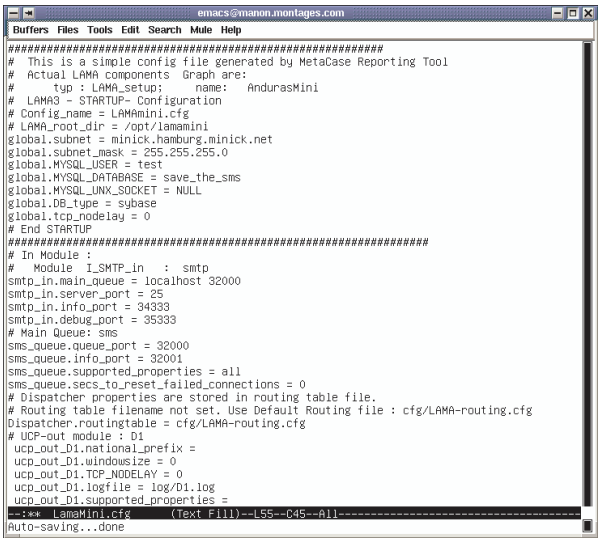
Case2: Configuration of services

- Telecom services and their configuration
- Users visually specify new configuration models
- Generate various configurations from single design
  - One model
  - Multiple outputs
- Reusable component library
- Code generators refers to external files

© 2005 Juha-Pekka Tolvanen / MetaCase



© 2005 Juha-Pekka Tolvanen / MetaCase

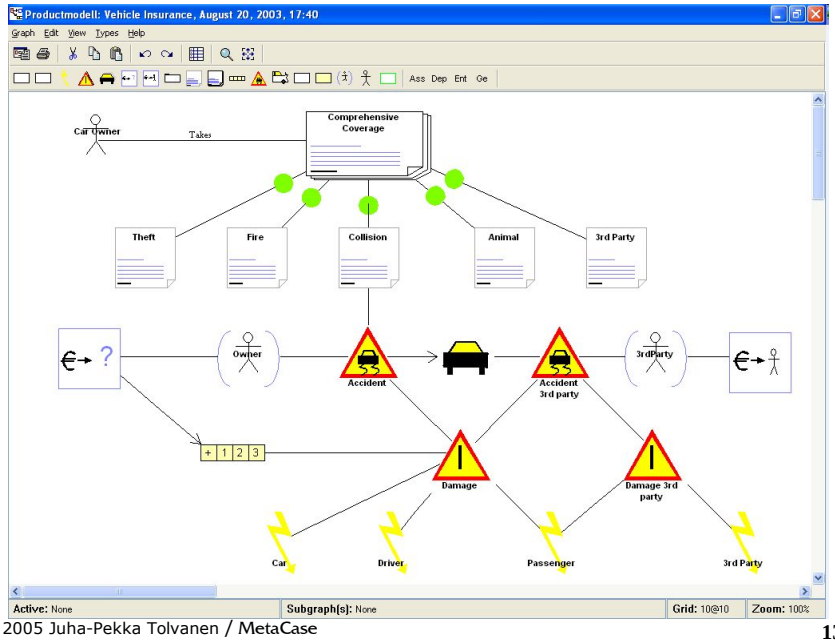


© 2005 Juha-Pekka Tolvanen / MetaCase

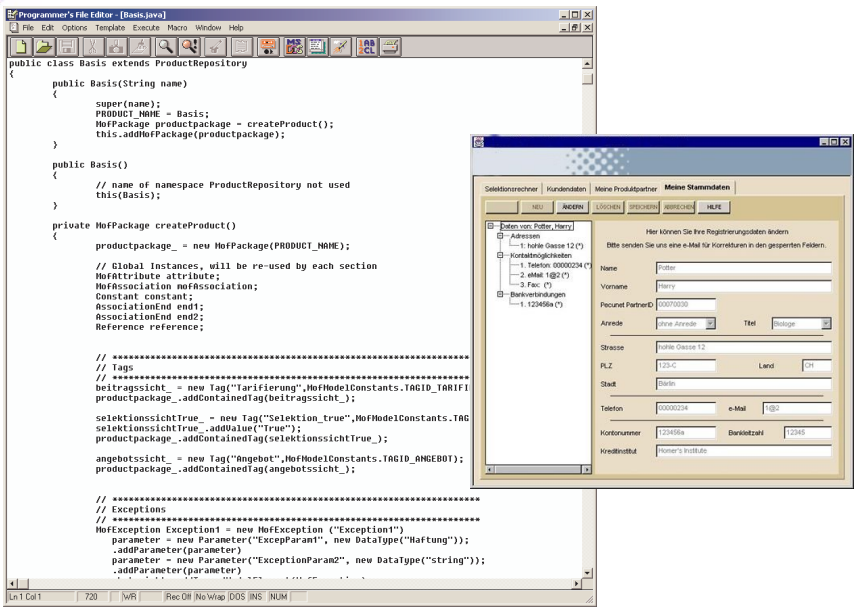
Case3: Insurance products & eCommerce

- Developing portal for insurances and financial products
- Need to specify several hundred financial products
- Insurance experts specify visually insurance products and generate code to the portal
- Comparison to writing directly Java after first 30 products = DSM at least 3 times faster

© 2005 Juha-Pekka Tolvanen / MetaCase



© 2005 Juha-Pekka Tolvanen / MetaCase

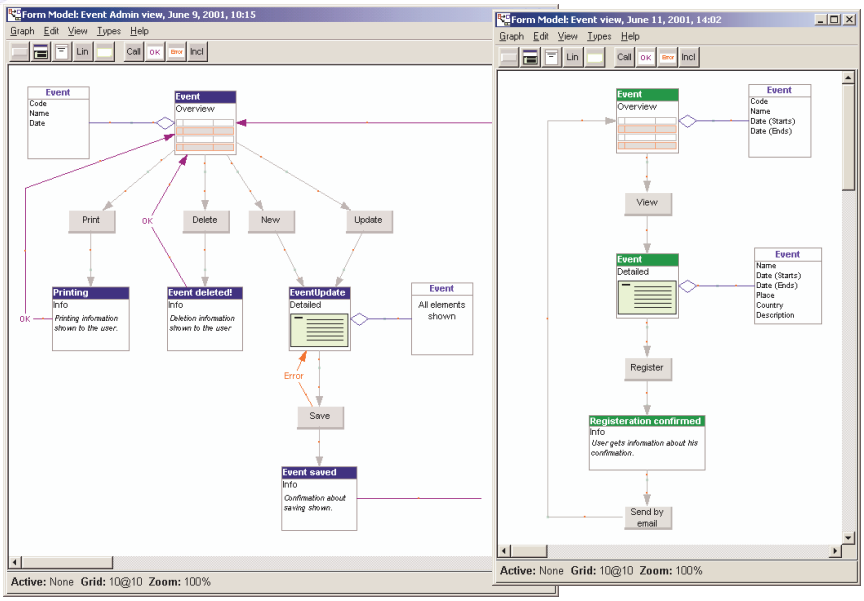


© 2005 Juha-Pekka Tolvanen / MetaCase

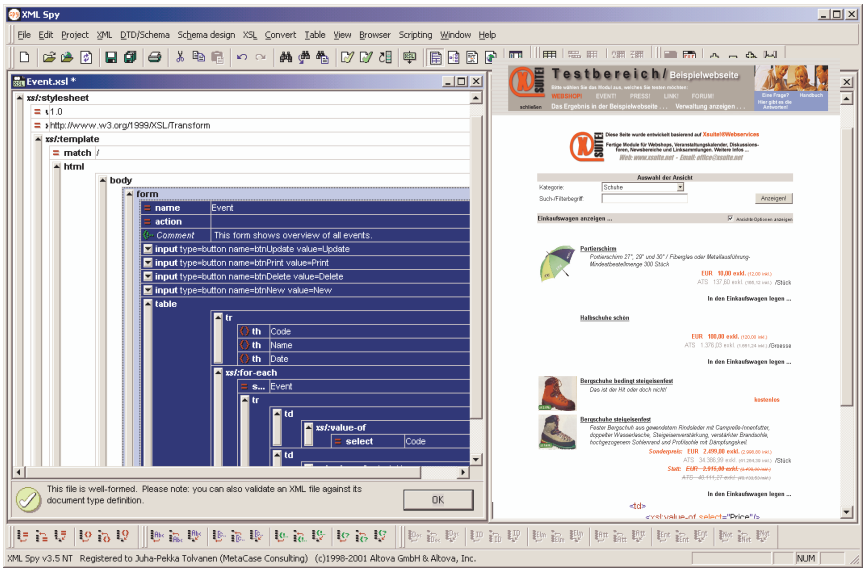
Case4: Web application

- Web application for e-commerce; product catalogs, events, press releases, and discussion forums
- Core components and basic functionality available for reuse and customization needs
- Each customer can specify own data content, behavioral logic and user interface
- Code generators produce running Java applets, stylesheets and xml files
- Generation of documents for both internal and external use

© 2005 Juha-Pekka Tolvanen / MetaCase

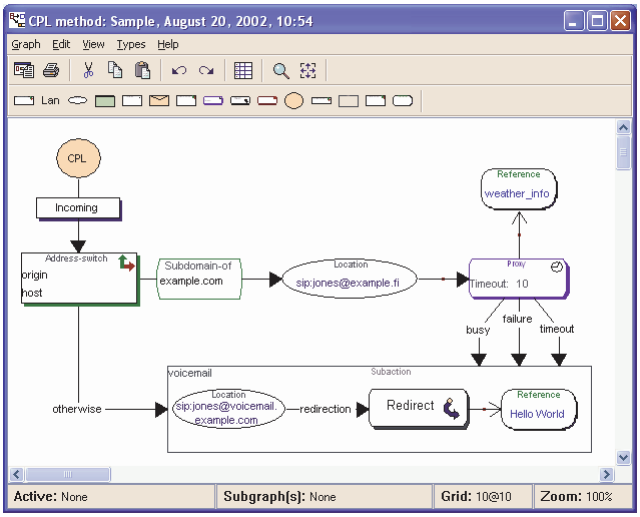


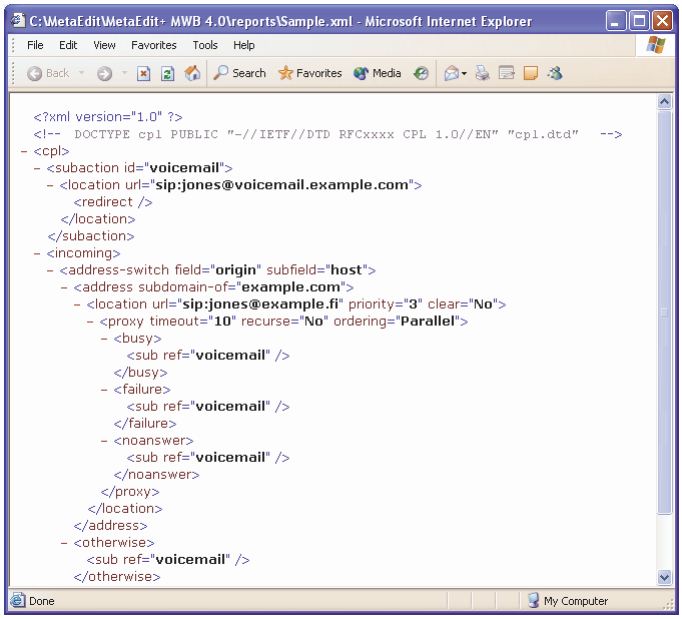
© 2005 Juha-Pekka Tolvanen / MetaCase



Case5: Call Processing Language

- Specify services than can run safely on Internet telephony servers
- Designs can be considered valid and well-formed already at the design stage
- Language use concepts familiar to the service developer
  - Switches, Locations and Signaling actions etc.
- Generate full service from the model
- There are also cases where the language has been extended to cover also domain extensions and new requirements e.g. for Java and VoiceXML.

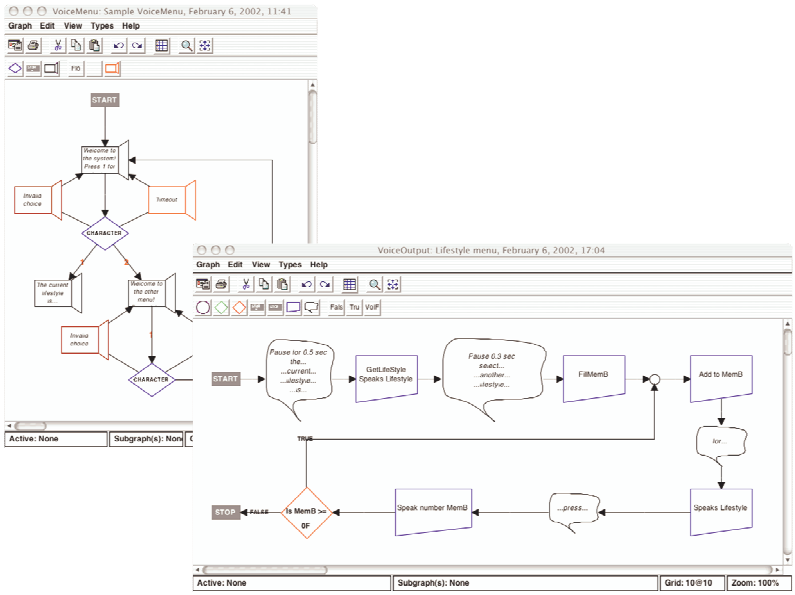

CPL method: Sample, August 20, 2002, 10:54
Graph Edit View Types Help
Lan
CPL
Incoming
Address-switch
origin
host
Subdomain-of
example.com
Location
sip:jones@example.fi
Proxy
Timeout: 10
Reference
weather_info
busy
failure
timeout
otherwise
voicemail
Subaction
Location
sip:jones@voicemail.example.com
redirection
Redirect
Reference
Hello World
Active: None
Subgraph(s): None
Grid: 10@10
Zoom: 100%


C:\MetaEdit\MetaEdit+ MWB 4.0\reports\Sample.xml - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
<?xml version="1.0" ?>
<!-- DOCTYPE cpl PUBLIC "-//IETF//DTD RFCxxxx CPL 1.0//EN" "cpl.dtd" -->
- <cpl>
- <subaction id="voicemail">
- <location url="sip:jones@voicemail.example.com">
<redirect />
</location>
</subaction>
- <incoming>
- <address-switch field="origin" subfield="host">
- <address subdomain-of="example.com">
- <location url="sip:jones@example.fi" priority="3" clear="No">
- <proxy timeout="10" recurse="No" ordering="Parallel">
- <busy>
<sub ref="voicemail" />
</busy>
- <failure>
<sub ref="voicemail" />
</failure>
- <noanswer>
<sub ref="voicemail" />
</noanswer>
</proxy>
</location>
</address>
- <otherwise>
<sub ref="voicemail" />
</otherwise>
Done
My Computer

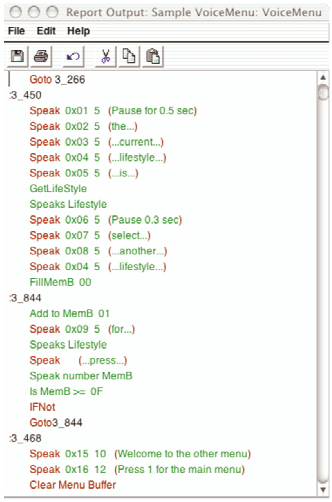
Case6:
VoiceMenu for microcontroller

- Voice VoiceMenu for microcontroller based home automation system
- Remote control for lights, heating, alarms, etc.
- VoiceMenus are programmed straight to the device with assembler-like language (8bit)
- Modeling language to define overall menu structure and individual voice prompts
- Code generator produces 100% of menu implementation
- Development time for a feature from a week to a day!

© 2005 Juha-Pekka Tolvanen / MetaCase



© 2005 Juha-Pekka Tolvanen / MetaCase



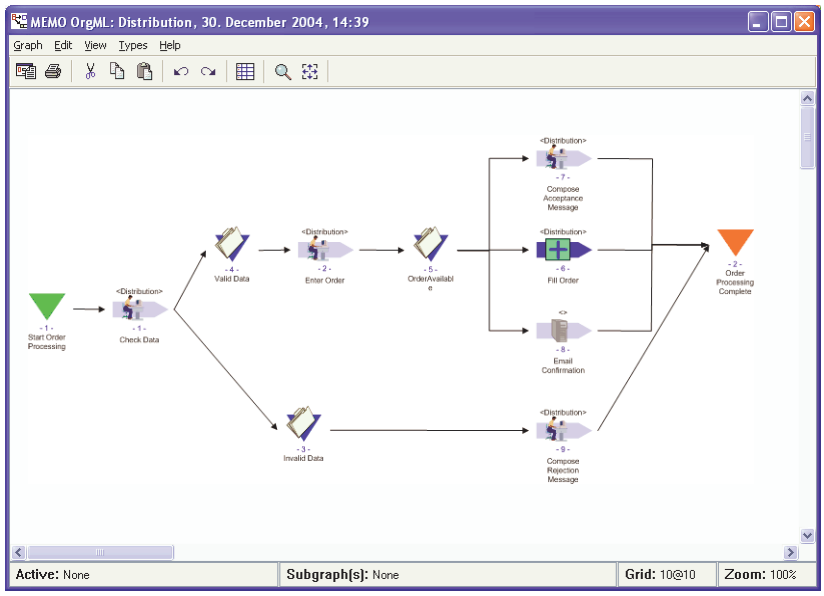
© 2005 Juha-Pekka Tolvanen / MetaCase

Case7: Business Process Modeling for XPDL

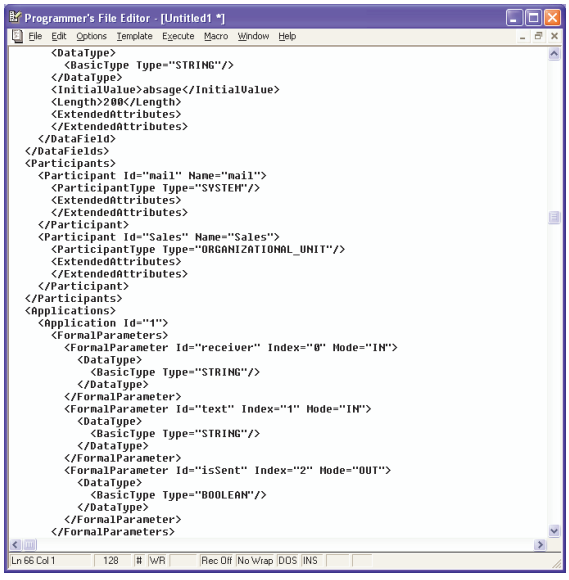
- Defining business processes to be executed in a workflow engine
- Modeling language about business processes
  - Contractors, Organizational units, Messages, Events, various type of Processes, etc.
- Generator to produce XPDL (XML Process Definition Language from Workflow Management Coalition (WfMC))
- XPDL executed in a workflow engine

* Jung, J.: Mapping of Business Process Models to Workflow Schemata 2004

© 2005 Juha-Pekka Tolvanen / MetaCase



© 2005 Juha-Pekka Tolvanen / MetaCase



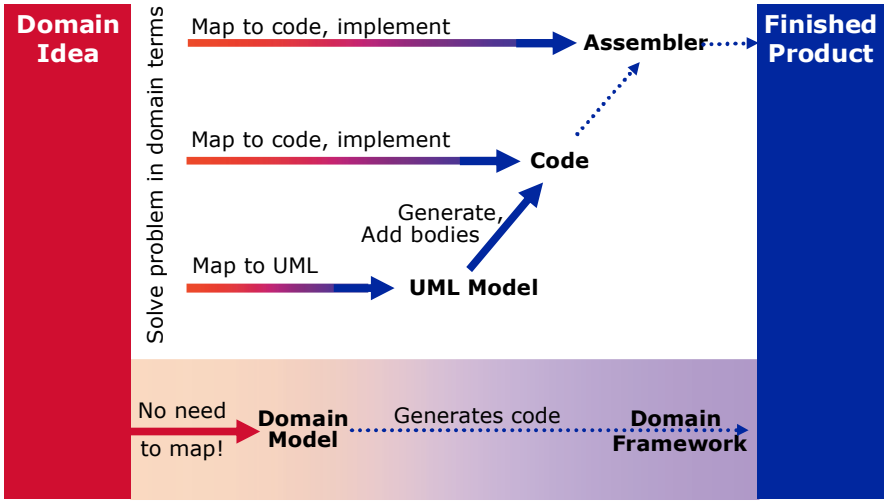
© 2005 Juha-Pekka Tolvanen / MetaCase

Why these are possible (now)?

- Need to fit only **one** company's requirements!
- Modeling is Domain-Specific
  - Works for one application domain, framework, product family etc.
  - Language has concepts people already are familiar with
  - Models used to solve the problem, not to visualize code
- Generator is Domain-Specific
  - Generate just the code needed from models
    - Efficient full code
    - No manual coding afterwards
    - No reason for round-tripping
  - Generator links to existing primitives/components/platform services etc.
  - Can produce Assembler, 3GL, object-oriented, XML, etc.

© 2005 Juha-Pekka Tolvanen / MetaCase

Modeling domain vs. modeling code



© 2005 Juha-Pekka Tolvanen / MetaCase

Domain-Specific Modeling

- **Captures domain knowledge (as opposed to code)**
  - Raise abstraction from implementation world
  - Uses domain abstractions
  - Applies domain concepts and rules as modeling constructs
    - model correctness, error prevention and optimization
  - Narrow down the design space
    - often focus on single range of products
- **Lets developers design products using domain terms**
  - ➔ Apply familiar terminology
  - ➔ Solve the RIGHT problems
  - ➔ Solve problems only ONCE!
    - directly in models, not again by writing code, round-trip etc.

© 2005 Juha-Pekka Tolvanen / MetaCase

Let's look industry experiences: Some reported cases

- Nokia; Mobile Phone product line
- Bell Labs / AT&T / Lucent; 5ESS telecommunications switch,
- Honeywell; embedded software architectures
- ORGA; SIM toolkit & JavaCard
- Pecunet; B2B E-Business: insurance
- LexiFi; mlFi, financial contracts
- DuPont; Activity Modeling
- NASA; Architecture Definition Language
- NASA ASE group; Amphion
- NASA JPL; embedded measurement systems
- USAF; Message Transformation and Validation
- …

Taken from www.DSMForum.org

© 2005 Juha-Pekka Tolvanen / MetaCase

DSM Case Study: Nokia

- DSM and related code generators for mobile phone*
- Order of magnitude productivity gains (10x)
  - "A module that was expected to take 2 weeks... took 1 day from the start of the design to the finished product"
- Focus on designs rather than code
  - Domain-oriented method allows developers to concentrate on the required functionality
- Training time was reduced significantly
  - "Earlier it took 6 months for a new worker to become productive. Now it takes 2 weeks"

* MetaCase, Nokia case study

© 2005 Juha-Pekka Tolvanen / MetaCase

DSM Case Study: Lucent

- 5ESS Phone Switch and several DSMs *
- Reported productivity improvements of about 3-10 times
  - From several cases
  - From several DSM languages
- Shorter intervals between product releases
- Improved consistency across product variants
  - "DSM should always be used if there are >3 variants"

* D. Weiss et al, Software Product-Line Engineering, Addison-Wesley

© 2005 Juha-Pekka Tolvanen / MetaCase

DSM case study: USAF

- Development of message translation and validation system (MTV)*
- Declarative domain-specific language
- + code generators and customization of components

Compared DSM against component-based development:

- DSM is 3 times faster than code components
- DSM leads to fewer errors: about 50% less
- DSM gives "superior flexibility in handling a greater range of specifications" than components

* Kieburtz et al., A Software Engineering Experiment in Software Component Generation, ICSE

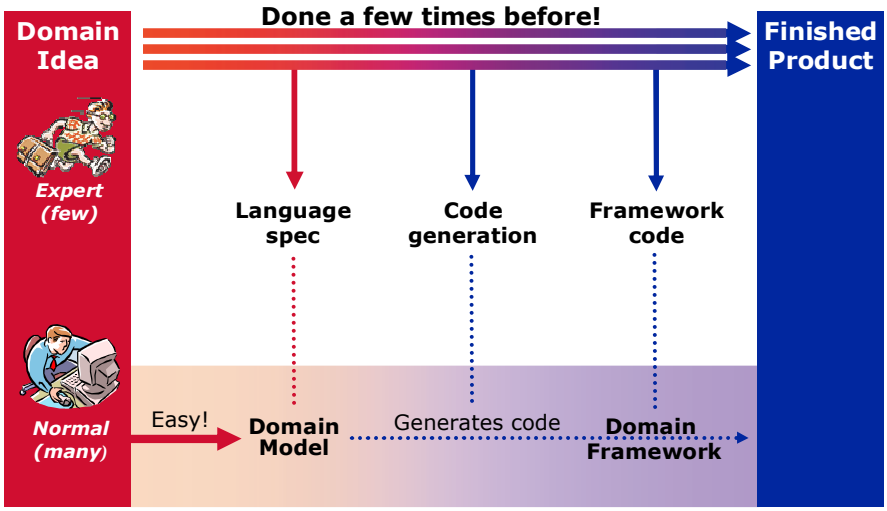
© 2005 Juha-Pekka Tolvanen / MetaCase

Where DSM makes most sense?

- Repetitive development tasks
  - Large portion of the work similar to earlier products (or several products made in parallel)
- Domain expertise needed
  - Non-programmers can participate
- These normally include:
  - Product Family
  - Platform-based development
  - Configuration
  - Business rule definitions
  - Embedded devices

© 2005 Juha-Pekka Tolvanen / MetaCase

How to implement DSM



© 2005 Juha-Pekka Tolvanen / MetaCase

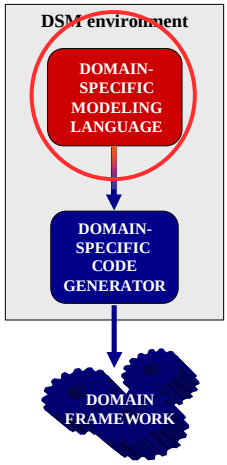
How to implement DSM

- Expert developer defines the DSM, others apply it
  - Expert defines the domain always better than less-exprerienced developers
  - Always better to define the concepts and mappings once, rather than let everyone do it all the time
- Delegate the job between the language, generator and domain framework
- Separation of concerns
  - Your experienced developers knowyour domain and code (not the tool vendor)
- DSM is agile: as much or as little as you want
- DSM implementation process is iterative and incremental

© 2005 Juha-Pekka Tolvanen / MetaCase

Defining modeling languages

- The most important asset of a DSM environment
  - application engineers use it
  - generator and platform largely invisible
- Often includes elements of familiar modeling paradigms
  - state machine
  - flow model
  - data structure, etc.
- Language specified as a metamodel



© 2005 Juha-Pekka Tolvanen / MetaCase

Requirements for modeling languages

- effectivity (effectiveness)
- efficiency
- completeness
- consistency
- accuracy
- well-defined products
- determinism
- relevance
- formalizability
- communicable
- reducing complexity
- stepwise
- integrated

© 2005 Juha-Pekka Tolvanen / MetaCase

Problem domain	Solution domain/ generation target	Approach
Telecom services	Configuration scripts	1
Insurance products	J2EE	1
Business processes	Rule engine language	1
Industrial automation	3 GL	1, (2)
Platform installation	XML	1, (2)
Medical device configuration	XML	1, (2)
Machine control	3 GL	1, 2
Call processing	CPL	2, (1)
Geographic Information System	3 GL, propriety rule language, data structures	2
SIM card profiles	Configuration scripts and parameters	2
Phone switch services	CPL, Voice XML, 3 GL	2, (3)
eCommerce marketplaces	J2EE, XML	2, (3)
SIM card applications	3 GL	3
Applications in microcontroller	8-bit assembler	3
Household appliance features	3 GL	3
Smartphone UI applications	Scripting language	3
ERP configuration	3 GL	3, 4
ERP configuration	3 GL	3, 4
Handheld device applications	3 GL	3, 4
Phone UI applications	C	4, (3)
Phone UI applications	C++	4, (3)
Phone UI applications	C	4, (3)
Phone UI applications	C++	4, (3)

*** Approaches used to defining DSM languages, SPLC, 2005**

Identifying DSM constructs

- Use domain concepts directly as modeling constructs
  - already known and used
  - established semantics exist
  - natural to operate with
  - easy to understand and remember
  - requirements already expressed using them
  - architecture often operates on domain concepts
- Focus on expressing design space with the language
  - use parameters of variation space
  - keep the language simple
  - try to minimize the need for modeling
  - do not visualize product code!
    - better to ”forget” your current code
- Apply suitable computational model(s) as a starting point

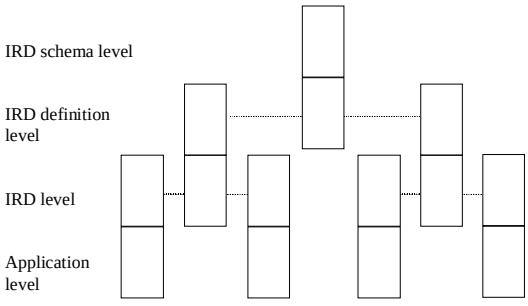
Identifying DSM constructs, 2

- Enrich chosen computational models with domain-specific concepts and rules
  - look at the type of design languages already used
- Investigate various alternatives for describing domain with the chosen models, e.g.
  - model element(s)
  - element properties
  - certain collection of elements
  - relationships between elements
  - model organization structures
- Specify as a metamodel in some format
  - draft samples with pen & paper
  - document early as a metamodel
  - implement in some metamodel-based tool
  - test it with real models

© 2005 Juha-Pekka Tolvanen / MetaCase

Levels of meta

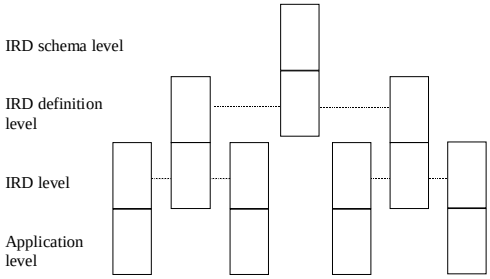
- Common in linguistics abstractions (Lyons 1977)
- Tool developers implemented these levels already at 70's
  - We need four levels, but operate at two at a time
- Became ISO's standard in IRDS reference framework (and later again with OMG)



© 2005 Juha-Pekka Tolvanen / MetaCase

IRDS reference framework

The IRD Definition Schema Level:
•A metametamodel according to which the IRD Definition level objects can be described.
•e.g. 'concept' (Wijers 1991) or OPRR's 'Object' metatype (Smolander 1992).
IRD Definition level:
•Schemata and application programs specs
•Metaclass level of languages such as Smalltalk.
•Metamodels
•e.g. specification of UML
The IRD level:
•DB schemata and app. programs
•Class level of class-based languages
•Models
•e.g. Customer entity
The application level
•Application data and execution.
•Instances of class-based languages.
•Instances
•e.g. Customer "Juha-Pekka"

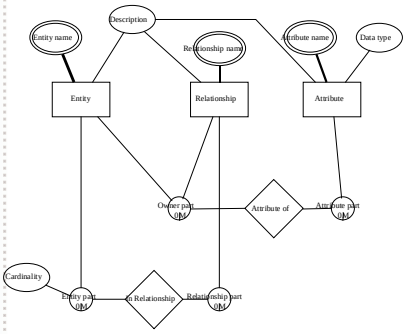


© 2005 Juha-Pekka Tolvanen / MetaCase

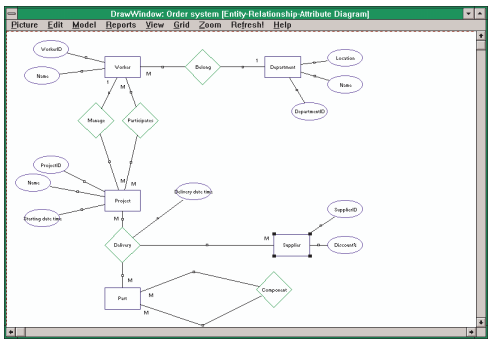
Implementing metamodels

- Metamodels support implementation of domain-specific development tools (modeling, interfaces, transformations)

Metamodel of ERD

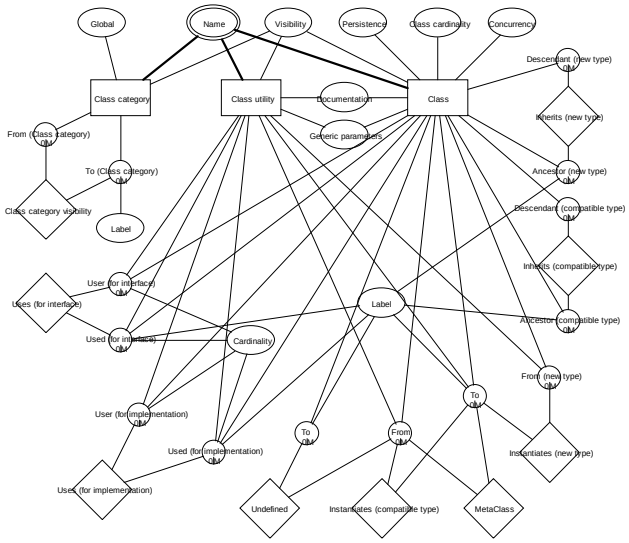


ERD Modeling (based on the metamodel)



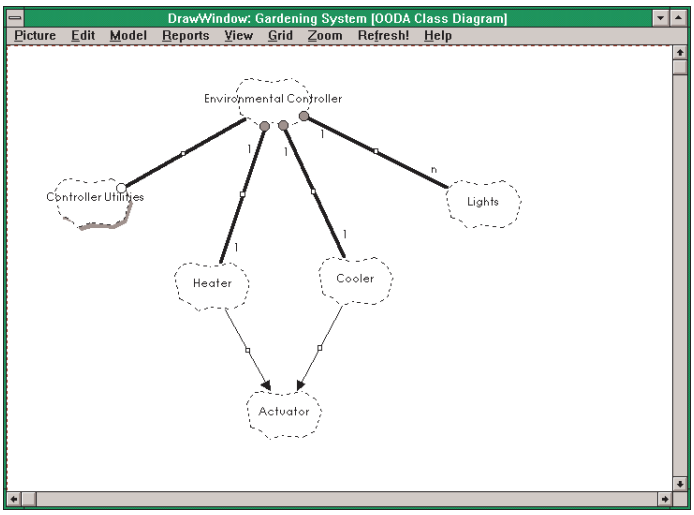
© 2005 Juha-Pekka Tolvanen / MetaCase

Example metamodel: Booch



© 2005 Juha-Pekka Tolvanen / MetaCase

Booch in use



© 2005 Juha-Pekka Tolvanen / MetaCase

Metamodeling languages

- Metamodeling is based on languages too!
- These vary from purpose
  - illustrating vs. formalizing methods
  - build tool support
  - integrate tools
  - exchange models
- What kind of representation for metamodels
  - graphical (ER, NIAM, OPRR, GOPRR, MOF, MOF+OCL, MS-DSL tool)
  - matrix (O/A Matrix),
  - text (ObjectZ, MDL, MEL, MOF/OCL), or
  - template based (GOPRR)

© 2005 Juha-Pekka Tolvanen / MetaCase

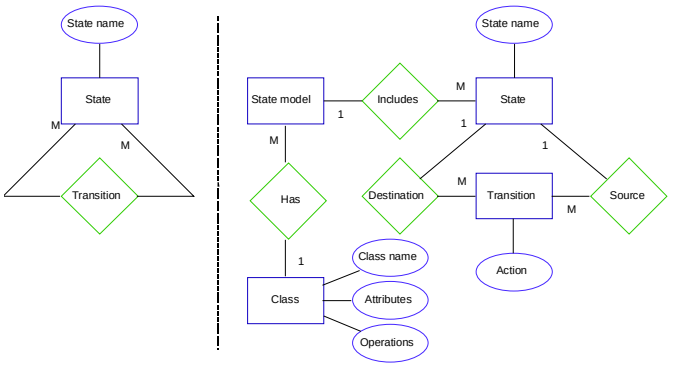
A short review to modeling power of metamodeling languages

- Example from object-oriented design method:
  - the life-cycle of class instances must be specified with one or more state models.
  - A state model contains states and transitions between two states.
  - A state must be specified by a name and a class may have only one state with a given name.
  - Each transition must be specified with an action which is executed when a transition occurs.
  - An action is specified as an operation of a class.

© 2005 Juha-Pekka Tolvanen / MetaCase

Metamodeling languages: ER

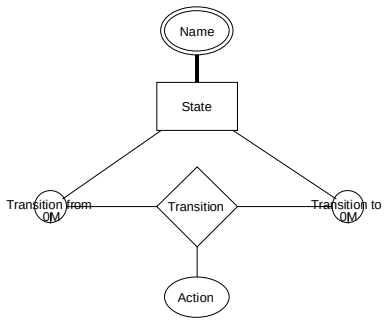
- What is missing from the metamodel:
  - Different modeling languages? (mapping of action)
  - Transitions should be always connected to states?
  - Mandatory state name?
  - Unique names?
  - …?



© 2005 Juha-Pekka Tolvanen / MetaCase

Metamodeling languages: OPRR

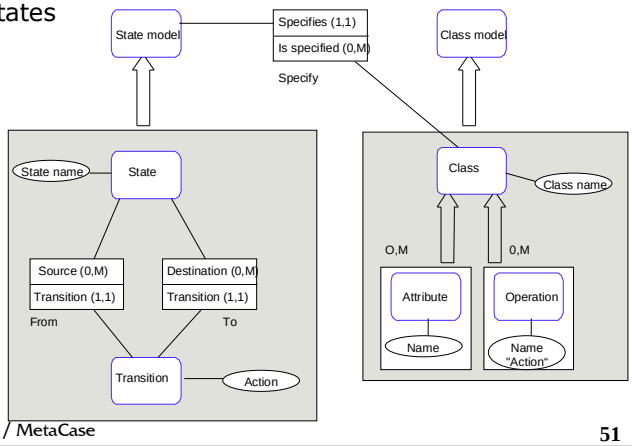
- What is missing from the metamodel:
  - Different modeling languages?
    - mapping of action to an operation of a class
    - mapping of state (model) to class
  - Mandatory state names?
  - Unique names?
  - …?



© 2005 Juha-Pekka Tolvanen / MetaCase

Metamodeling languages: CoCoA

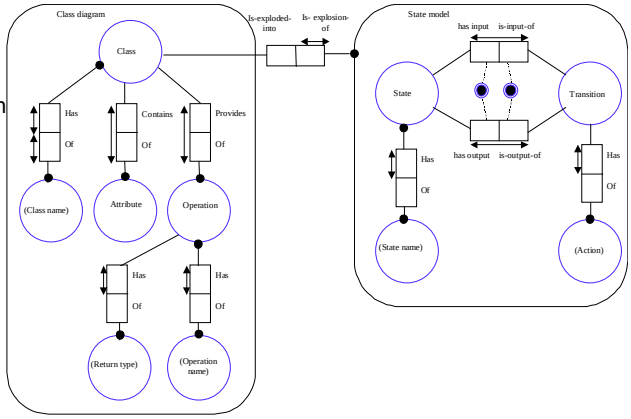
- What is missing from the metamodel:
  - Dependency of operations to actions
  - Mandatory attributes
  - Unique states
  - …?



© 2005 Juha-Pekka Tolvanen / MetaCase

Metamodeling languages: NIAM

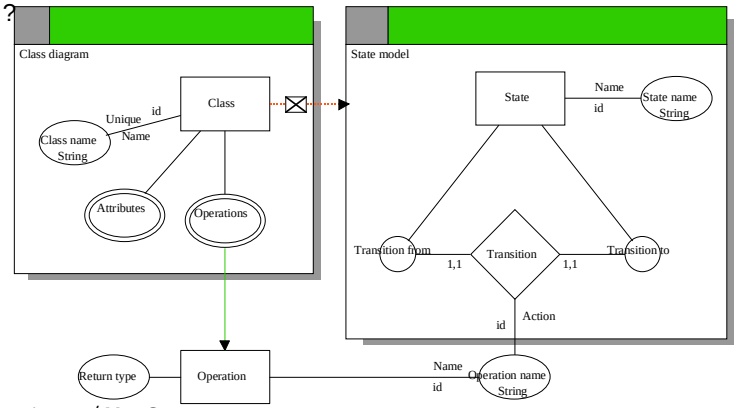
- What is missing from the metamodel:
  - Transition and state mapping to (notation) different types
  - Integrated methods?
    - Linkage between action and operation
  - …?



© 2005 Juha-Pekka Tolvanen / MetaCase

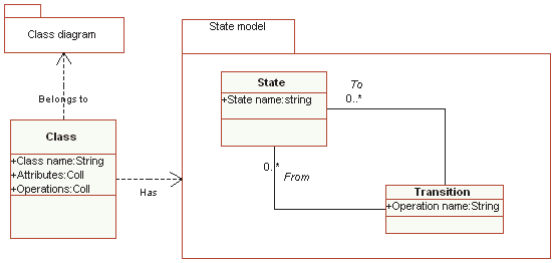
Metamodeling languages: GOPRR

- What is missing from the metamodel:
  - How many state models a class can have?
  - Uniqueness of states (for one class)?
  - Mandatory values?
  - …?



Metamodeling languages: MOF/UML

- What is missing from the metamodel:
  - How many state models a class can have?
  - mapping of action to an operation of a class
  - Uniqueness of states (namespace, for one class)?
  - Mandatory values?
  - …?



DSM definition must include also other than pure language concepts

- Initial:
  - Metamodel: concepts and rules of the language
  - Notation: symbols and their behavior
  - Tool: editors, dialogs, icons, browsers etc.
  - Generators: for code, checking, inspection, docs etc.
  - Language help
  - Connectivity with other tools
- Continuously:
  - DSM language (and tool) sharing
  - Language updates (of metamodel, notation)
  - Generator updates
  - Model updates based on changed language
  - …often in multi developer settings

© 2005 Juha-Pekka Tolvanen / MetaCase

Tools support is essential

- Building DSM must be fast, cheap and easy
- A variety of tools available
  - Lex & Yacc
  - Customizable IDE
  - Metamodel-based tools
- 5 ways to get the tools
  1. Write own tool from scratch
  2. Write own tool based on frameworks
  3. Metamodel, generate tool skeleton, add code
  4. Metamodel, generate full tool
  5. Integrated modeling and metamodeling environment

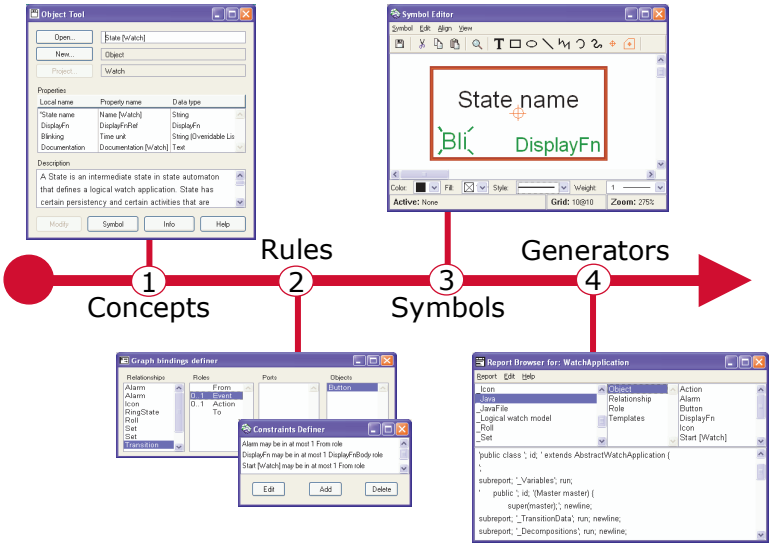
© 2005 Juha-Pekka Tolvanen / MetaCase

What about tools?

- Tools for textual languages (late 70's ->)
  - SEM (Teichroew and Yamato)
  - Others include Plexsys, Metaplex, Quickspec, PSL/PSA
- Tools for graphical languages (mid 80's)
  - Swedish Ramatic: set theoretical constructs to specify graphical notations/languages
  - British Eclipse: directed graphs
- Tools for graphical metamodeling (late 80's)
  - Finnish Metamodeling Editor MetaEdit: extended ER
- + tens of others in the past available: MetaView, Kogge, Virtual Software Factory, Customizer in Excelerator, Paradigm+ SDK, ConceptBase, IPSYS toolbuilder, Dome, GME etc.
  - Most of the tools focus on initial language specification and editor construction

© 2005 Juha-Pekka Tolvanen / MetaCase

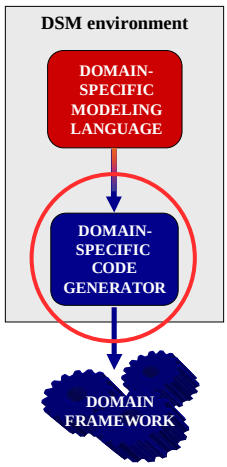
MetaEdit+: metamodel as data



© 2005 Juha-Pekka Tolvanen / MetaCase

Generator

- Generator translates the computational model into a required output
  1. crawls through the models
     → navigation according to metamodel
  2. extract required information
     → access data in models
  3. translates it as the code
     → translation semantics and rules
  4. using some output format
     → possibility to define output format
- There are different generator approaches
  - ”Out-of-box” generators
  - Customizable generators
  - Domain-Specific generators



© 2005 Juha-Pekka Tolvanen / MetaCase

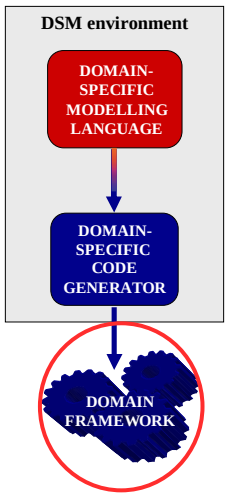
Implementing code generators

- Keep generator (and generation process) as simple as possible
  - Raise variation handling into the modeling language (as data)
  - Push low-level implementation issues down to the framework
- Try to generate as little code as possible
  - Glue code only
  - Change the target platform or make domain framework if you can
- Use as many prebuilt building blocks (from the platform) as possible
  - Generated code can call components
  - Generator knows how to do it, developer doesn’t need to know

© 2005 Juha-Pekka Tolvanen / MetaCase

Domain framework

- Provides an interface for the target platform and programming language
- Raise the level of abstraction on the platform side
- Achieved by atomic implementations of commonalities and variabilities
  - especially for behavior
  - implementation as templates and components
- Include **interface** for the code to be generated
  - often the only needed part for static variation (e.g. for XML schema)



© 2005 Juha-Pekka Tolvanen / MetaCase

Implementing code generators, 2

- Move to the generator
  - Language syntax variation
  - Output format
- Keep generator modular to reflect changes
- Target 100% generation output
  - Never modify the generated code
    - think about changing assembler after compiling
  - Correct the generator or framework instead
    - No round-trip-related problems
- Template vs. programmable generator?
  - templates simpler and easier to use, but also more restricted by capabilities
  - Programmable generator better for more complex needs
    - external generators from the modeling tool perspective

© 2005 Juha-Pekka Tolvanen / MetaCase

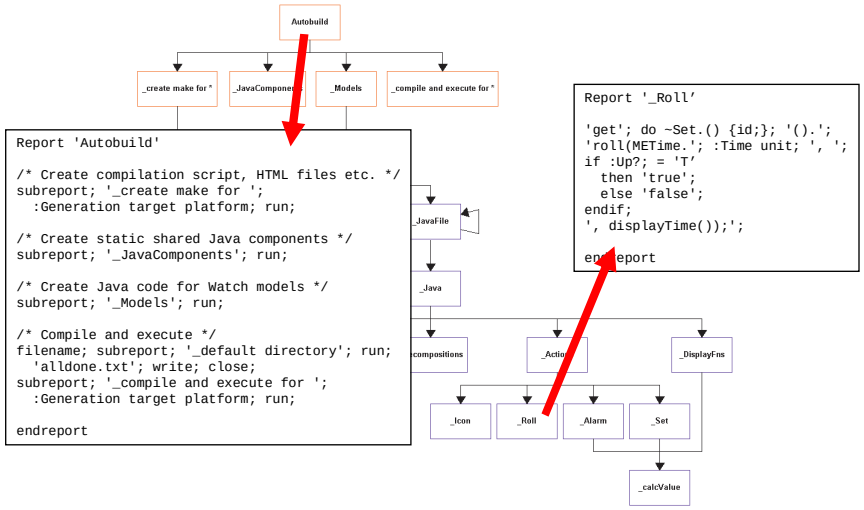
Generator degrees of freedom

- Different levels of generators: modular / tree structure
  1. Generator per file to be generated
  2. Generator per section in a file
  3. Generator per metamodel element
- Different Model of Computation implementations
  - Sequential
  - Function calls
  - Switch-case structure
  - Transition tables, etc.
- Different levels of code that generated code can call or subclass
  - Other generated code
  - Domain framework components
  - Platform functions
- Different generation options for different runs
  - Different top-level generators
  - Top-level graph for generation options

© 2005 Juha-Pekka Tolvanen / MetaCase

Code generator structure

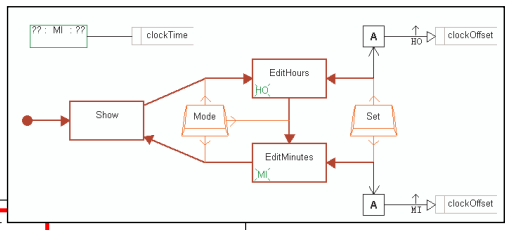
- Modular implementation to manage complexity



© 2005 Juha-Pekka Tolvanen / MetaCase

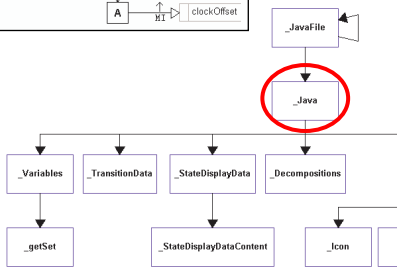
Transition tables - Watch/Java

- Java from the extended state machine; Watch



Generator ouput

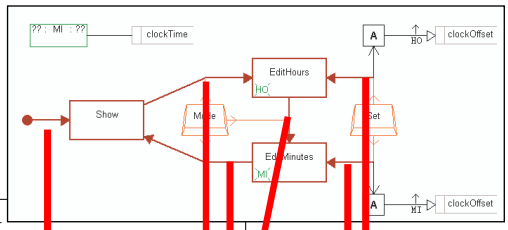
```
public SimpleTime(Master master) {
  super(master);

  addTransition ("Start [Watch]", "", 0, "Show");
  addTransition ("Show", "Mode", 0, "EditHours");
  addTransition ("EditHours", "Set", a22_2926, "EditHours");
  addTransition ("EditHours", "Mode", 0, "EditMinutes");
  addTransition ("EditMinutes", "Set", a22_1405, "EditMinutes");
  addTransition ("EditMinutes", "Mode", 0, "Show");

  addStateDisplay("Show", -1, METime.MINUTE, d22_977);
  addStateDisplay("EditHours", METime.HOUR_OF_DAY, METime.MINUTE,
    d22_977);
  addStateDisplay("EditMinutes", METime.MINUTE, METime.MINUTE,
    d22_977);
};
```



© 2005 Juha-Pekka Tolvanen / MetaCase

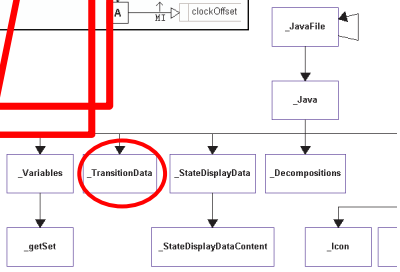
Transition tables - Watch/Java

- Java from the extended state machine; Watch



Generator ouput

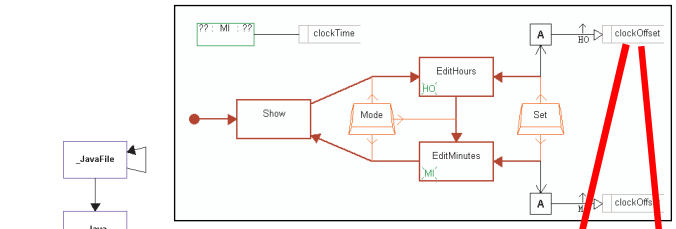
```
public SimpleTime(Master master) {
  super(master);

  addTransition ("Start [Watch]", "", 0, "Show");
  addTransition ("Show", "Mode", 0, "EditHours");
  addTransition ("EditHours", "Set", a22_2926, "EditHours");
  addTransition ("EditHours", "Mode", 0, "EditMinutes");
  addTransition ("EditMinutes", "Set", a22_1405, "EditMinutes");
  addTransition ("EditMinutes", "Mode", 0, "Show");

  addStateDisplay("Show", -1, METime.MINUTE, d22_977);
  addStateDisplay("EditHours", METime.HOUR_OF_DAY, METime.MINUTE,
    d22_977);
  addStateDisplay("EditMinutes", METime.MINUTE, METime.MINUTE,
    d22_977);
};
```

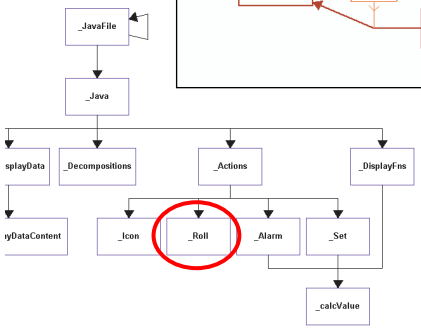


© 2005 Juha-Pekka Tolvanen / MetaCase

Code generator example (cont.)

- Generating watch applications





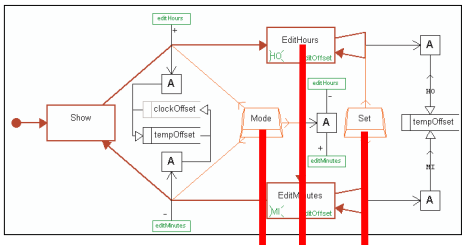
Generator output

```
public Object perform(int methodId) {

  switch (methodId) {
    case a22_2026:
      getclockOffset().roll(METime.HOUR_OF_DAY, true,
        displayTime());
      return null;
    case a22_1405:
      getclockOffset().roll(METime.MINUTE, true,
        displayTime());
      return null;
    case d22_977:
      return getclockTime();
  }
  return null;
}
```

© 2005 Juha-Pekka Tolvanen / MetaCase

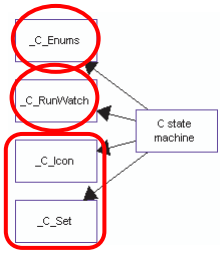
Switch-case – Watch/C

- C from the extended state machine; Watch



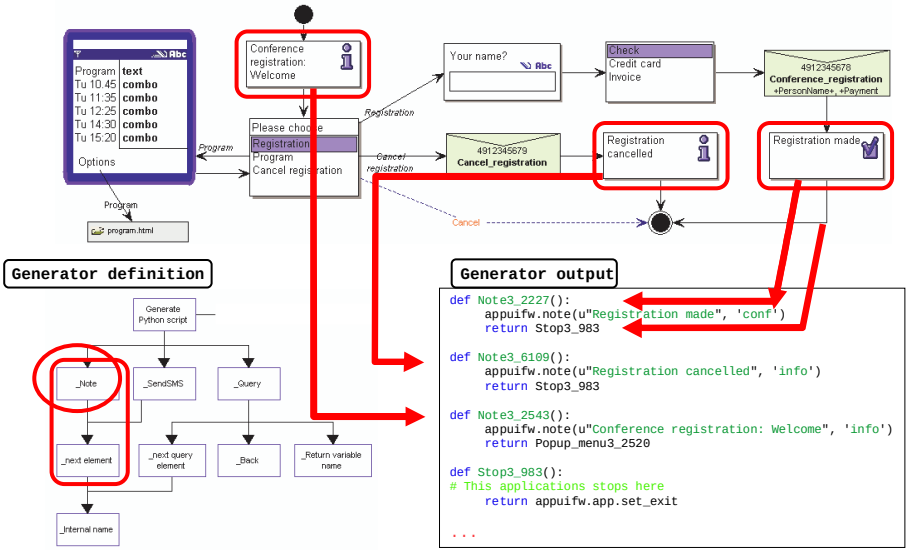
Generator ouput

```
void handleEvent()
{
  switch (state)
  {
  case EditHours:
     switch (button)
     {
        case Set:
           state = EditHours;
           break;
        case Mode:
           icon (Off,editHours);
           icon (On,editMinutes);
           state = EditMinutes;
           break;
        default:
           break;
     }
```



© 2005 Juha-Pekka Tolvanen / MetaCase

Function calls - S60/Python

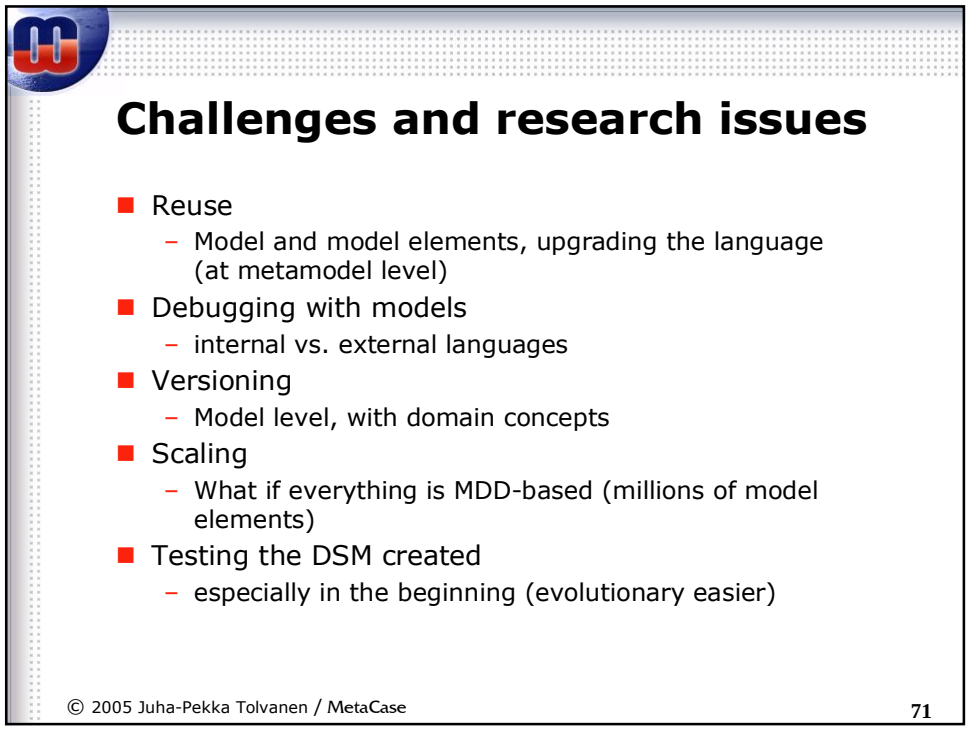


© 2005 Juha-Pekka Tolvanen / MetaCase

Other-than-code generators

- Power of having single source for multiple targets!
  - Checking completeness and uniformity
  - Configuration
  - Testing and analysis
  - Automated build → automating compile and execution
  - Help text
  - User guides
  - Documentation and review

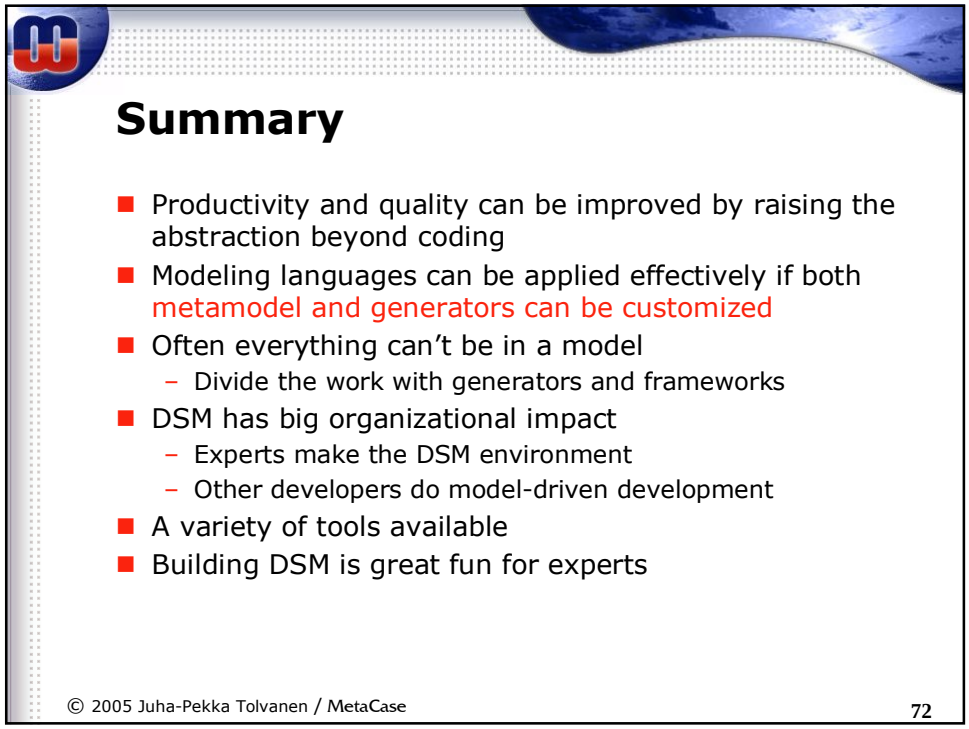
© 2005 Juha-Pekka Tolvanen / MetaCase



Challenges and research issues

- Reuse
  - Model and model elements, upgrading the language (at metamodel level)
- Debugging with models
  - internal vs. external languages
- Versioning
  - Model level, with domain concepts
- Scaling
  - What if everything is MDD-based (millions of model elements)
- Testing the DSM created
  - especially in the beginning (evolutionary easier)

© 2005 Juha-Pekka Tolvanen / MetaCase



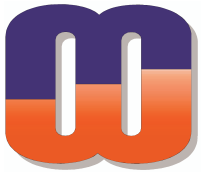
Summary

- Productivity and quality can be improved by raising the abstraction beyond coding
- Modeling languages can be applied effectively if both metamodel and generators can be customized
- Often everything can't be in a model
  - Divide the work with generators and frameworks
- DSM has big organizational impact
  - Experts make the DSM environment
  - Other developers do model-driven development
- A variety of tools available
- Building DSM is great fun for experts

© 2005 Juha-Pekka Tolvanen / MetaCase

Thank you!

Question and comments?



Juha-Pekka Tolvanen, jpt@metacase.com
www.metacase.com

USA:
MetaCase
5605 North MacArthur Blvd.
11th Floor, Irving, Texas 75038
Phone (972) 819-2039
Fax (480) 247-5501

International:
MetaCase
Ylistönmäentie 31
FI-40500 Jyväskylä, Finland
Phone +358 14 4451 400
Fax +358 14 4451 405

© 2005 Juha-Pekka Tolvanen / MetaCase

Literature and further links

- DSM Forum, www.dsmforum.org
- Brinkkemper, S., Lyytinen, K., Welke, R., Method Engineering - Principles of method construction and tool support, Chapman & Hall, 1996
- Czarnecki, K., Eisenecker, U., Generative Programming, Methods, Tools, and Applications, Addison-Wesley, 2000.
- Gray, J., Rossi, M., Tolvanen, J-P, (eds.) Special issue of Journal of Visual Languages and Computing on Domain-Specific Modeling with Visual Languages, Vol 15 (3-4), 2004
- Jung, J.: Mapping of Business Process Models to Workflow Schemata - An Example Using MEMO-OrgML and XPDL, Arbeitsberichte des Instituts für Wirtschaftsinformatik, Nr. 47, Koblenz 2004
- Kieburtz, R. et al., A Software Engineering Experiment in Software Component Generation, Proceedings of 18th International Conference on Software Engineering, Berlin, IEEE Computer Society Press, March, 1996.
- Pohjonen, R., Kelly, S., Domain-Specific Modeling, Dr. Dobb's, 8, 2002
- Tolvanen, J.-P., Pohjonen, R., Automated Production of Family Members: Lessons Learned. Proceedings of International workshop of Product Line Engineering, Technical Report at Fraunhofer IESE (eds. K. Schmid, B. Geppert) 2002.
- Weiss, D., Lai, C. T. R., Software Product-line Engineering, Addison Wesley Longman, 1999.

© 2005 Juha-Pekka Tolvanen / MetaCase

DSM related events

- Workshops on Domain-Specific Modeling (5th at OOPSLA 2005)
- IEEE Symposium on Visual Languages and Formal Methods (VLFM '03)
- Engineering Methods to Support Information Systems Evolution' (EMSISE'03)
- International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT '02)
- International Workshop on Model Engineering, ECOOP'00

© 2005 Juha-Pekka Tolvanen / MetaCase