

Domain-Specific Languages and Generators for Model-Driven Development



22.9.2003

Juha-Pekka Tolvanen





Outline

- Introduction
- What is domain-specific modelling (DSM)
- Architecture for defining and using DSM
- Implementing modelling languages
- Implementing generators
- Summary

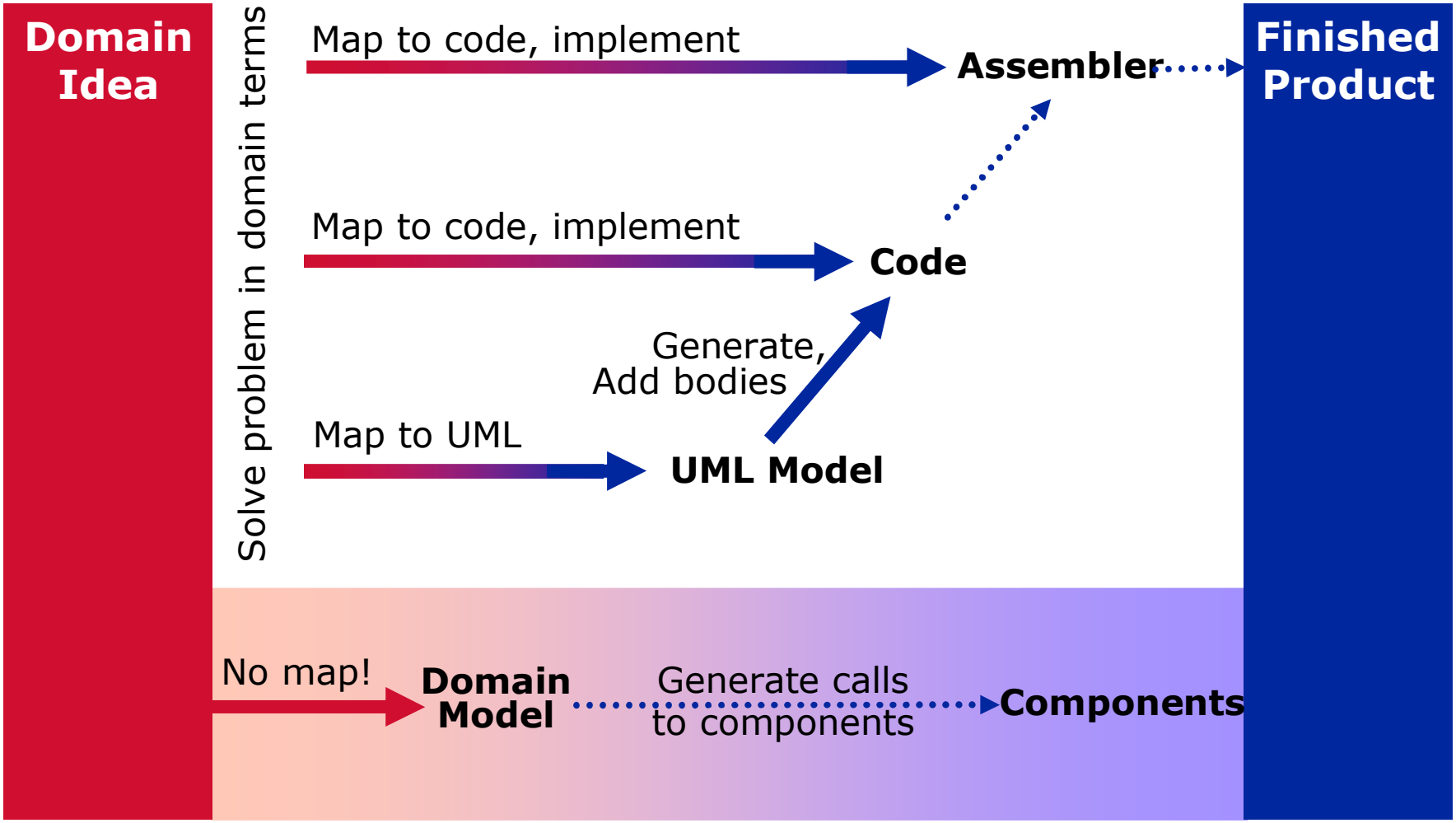
© 2003 MetaCase

Outline

- Introduction
- What is domain-specific modelling (DSM)?
  - Why DSM?
  - DSM examples from different application domains
  - Industrial experience reports
  - Where to apply

© 2003 MetaCase

Modelling domain vs. modelling code



© 2003 MetaCase



What is domain-specific modelling

- **Captures domain knowledge (as opposed to code)**
  - Raise abstraction from implementation world
  - Uses domain abstractions
  - Applies domain concepts and rules as modelling constructs
  - Narrow down the design space
  - Focus on single range of products
- **Lets developers design products using domain terms**
  - ➔ Apply familiar terminology
  - ➔ Solve the RIGHT problems
    - i.e. those problems the language is defined for
  - ➔ Solve problems only ONCE!
    - directly in models, not again by writing code

© 2003 MetaCase



Why Domain-Specific Modelling

- Development is supposed to become faster
  - less to ”write” as abstraction is raised
  - features made and changed faster
- Development becomes easier
  - lower level details hidden (see Assember for C)
  - models easier to read, understand and remember
- Expertise can be leveraged
  - domain to code mappings made only once – by the expert

© 2003 MetaCase

Example: Digital wristwatch



Domain Idea

- Product family
  - Models: His, Hers, Sport, Kid, Traveller, Diver…
- Reusable component applications
  - Time, Alarm, Timer, WorldTime, StopWatch…
- Hide complexity from modeller
  - Model-View-Controller separation
  - Separate thread for real-time display
- Implementation in Java

No map! → **Domain Model** ⋯ Generate calls to components ⋯▸ **Components**

Finished Product

© 2003 MetaCase

Comparing Code, UML, DSM



Domain Idea

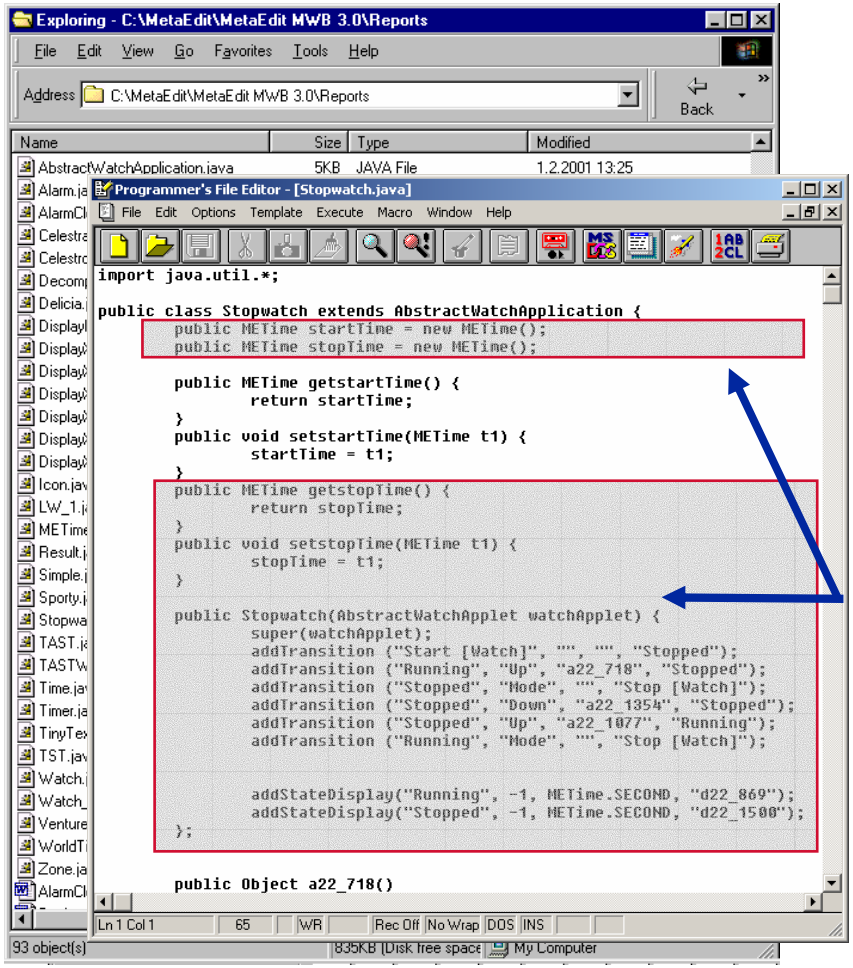
- Let's make a new model and function!
  - Pure stopwatch with lap-time function
- Points of comparison:
  - Where are we thinking: domain or code world?
  - Support for getting the right domain solution?
  - How long does it take?
  - Can we do it right first time?
  - Is everything updated that should be?

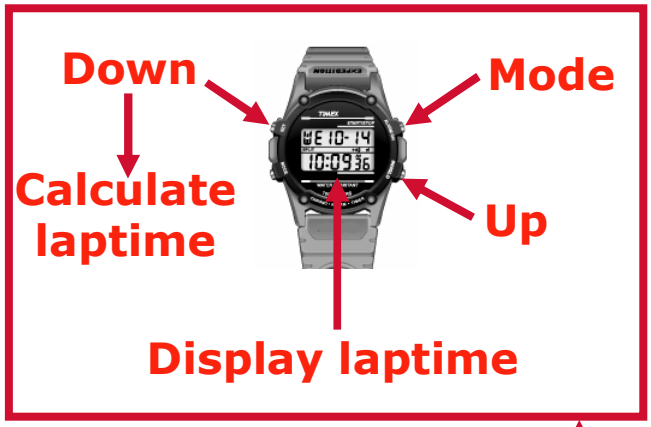
Finished Product

No map! → **Domain Model** ⋯ Generate calls to components ⋯▸ **Components**

© 2003 MetaCase

Code-based approach

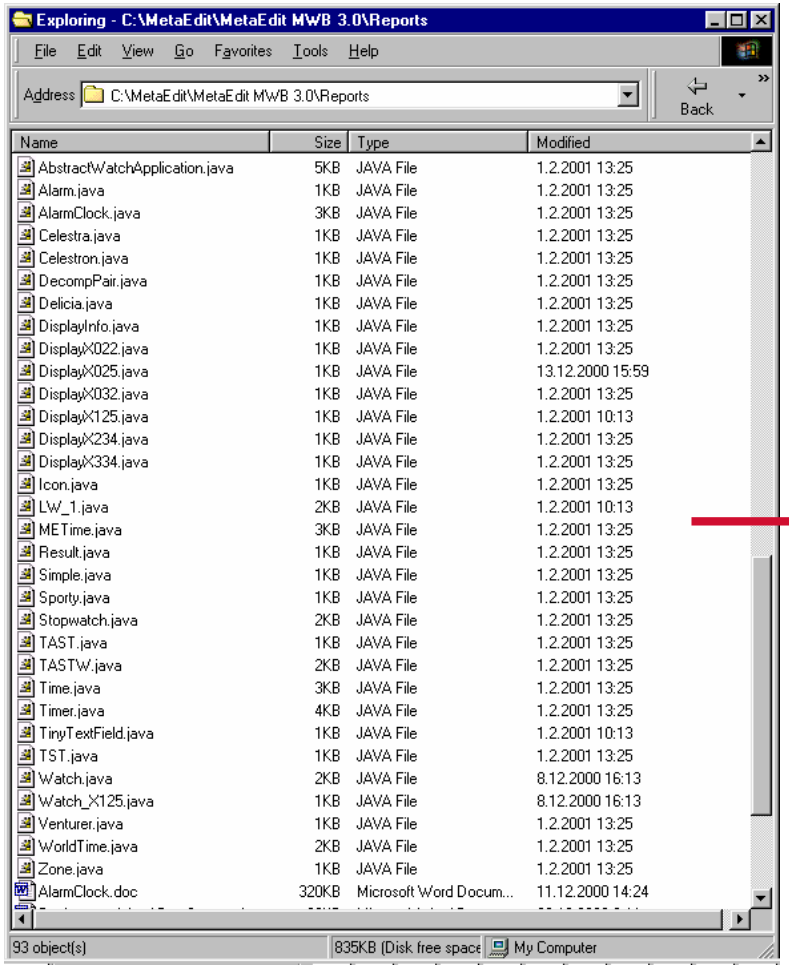


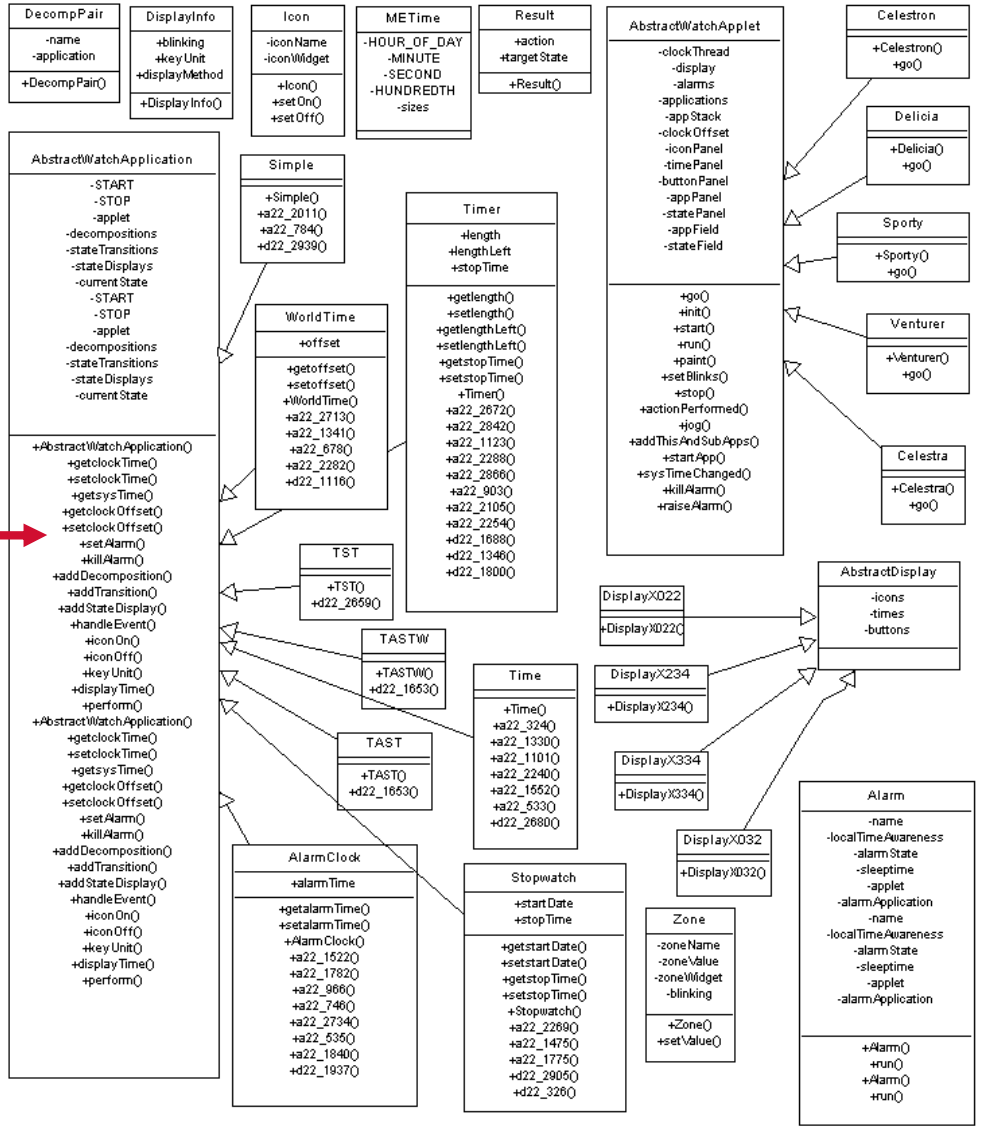


1. Read the documents
2. Find the solution
3. Find the relevant code
4. Change the right code
5. Document the code change
6. Test the changes
7. Document the solution

© 2003 MetaCase

Code visualization approach

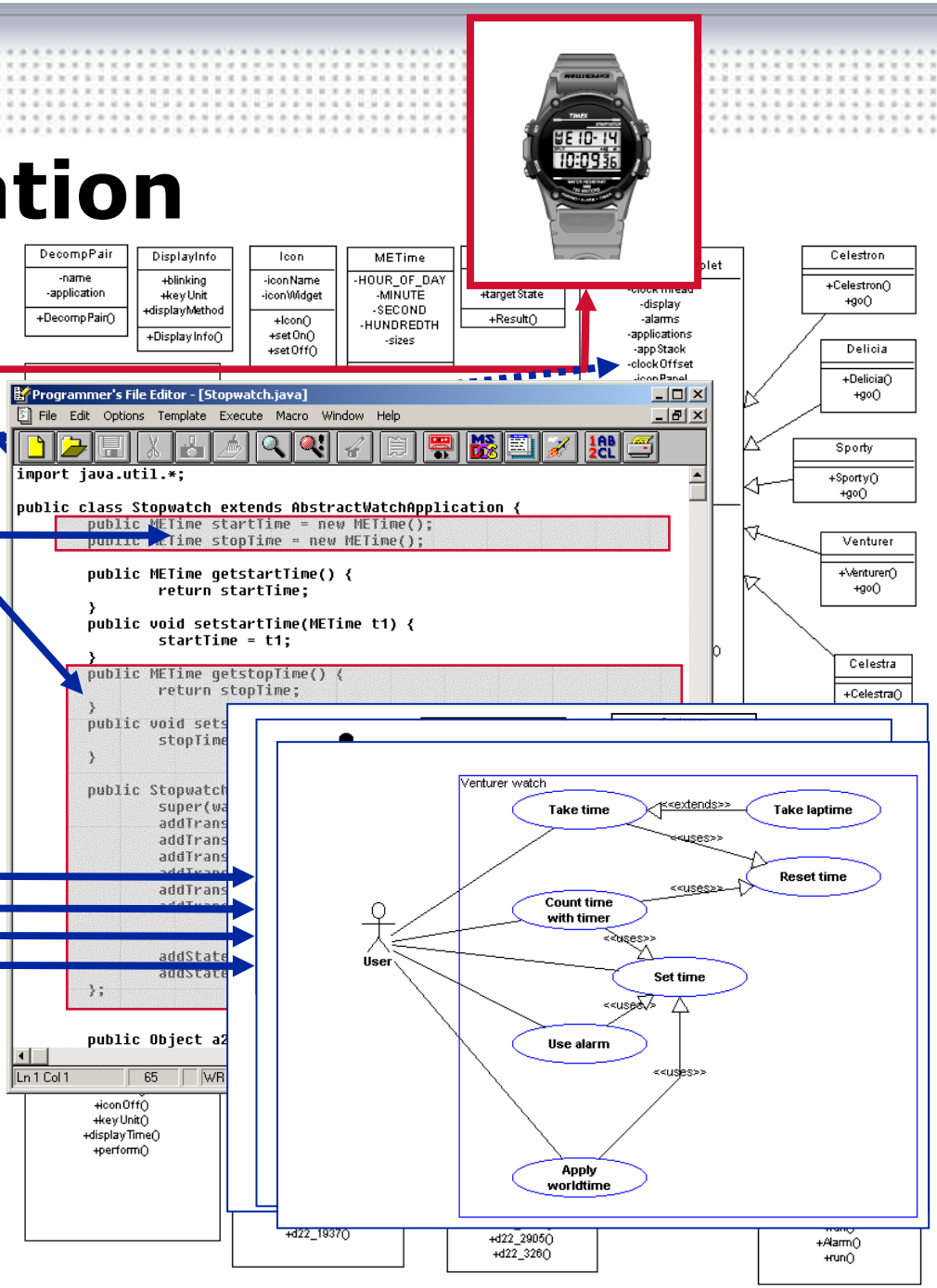




© 2003 MetaCase

Code visualization

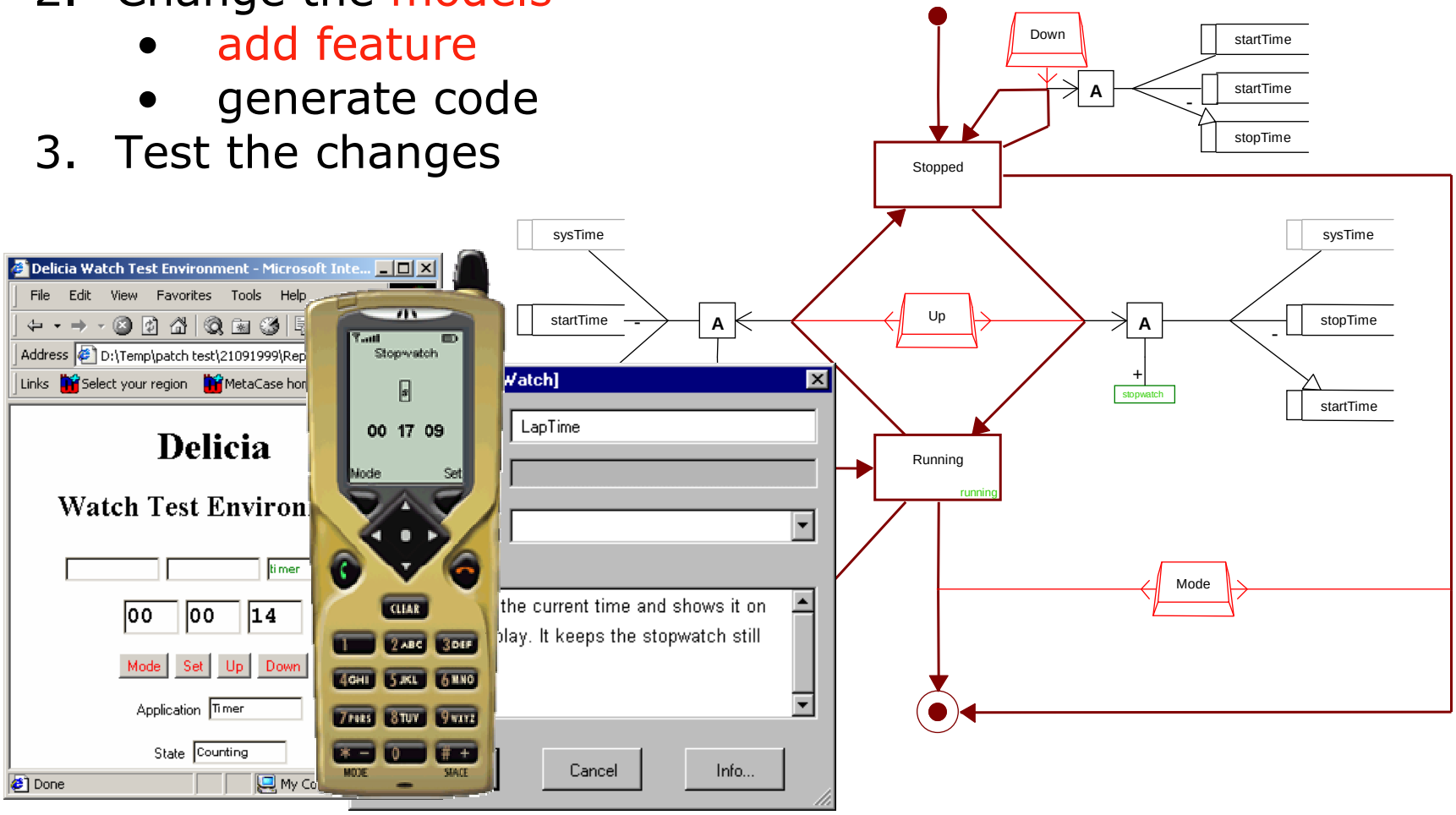
1. Read the documents
2. Find the solution
3. Find the relevant models
4. Change the right code and models
5. Document the code and model changes
6. Test the changes
7. Update models (Use cases, Class models, Message sequences models, State models etc.)
8. Document the solution



© 2003 MetaCase

Domain-Specific Modelling

1. Find the relevant models
2. Change the models
   - add feature
   - generate code
3. Test the changes



© 2003 MetaCase



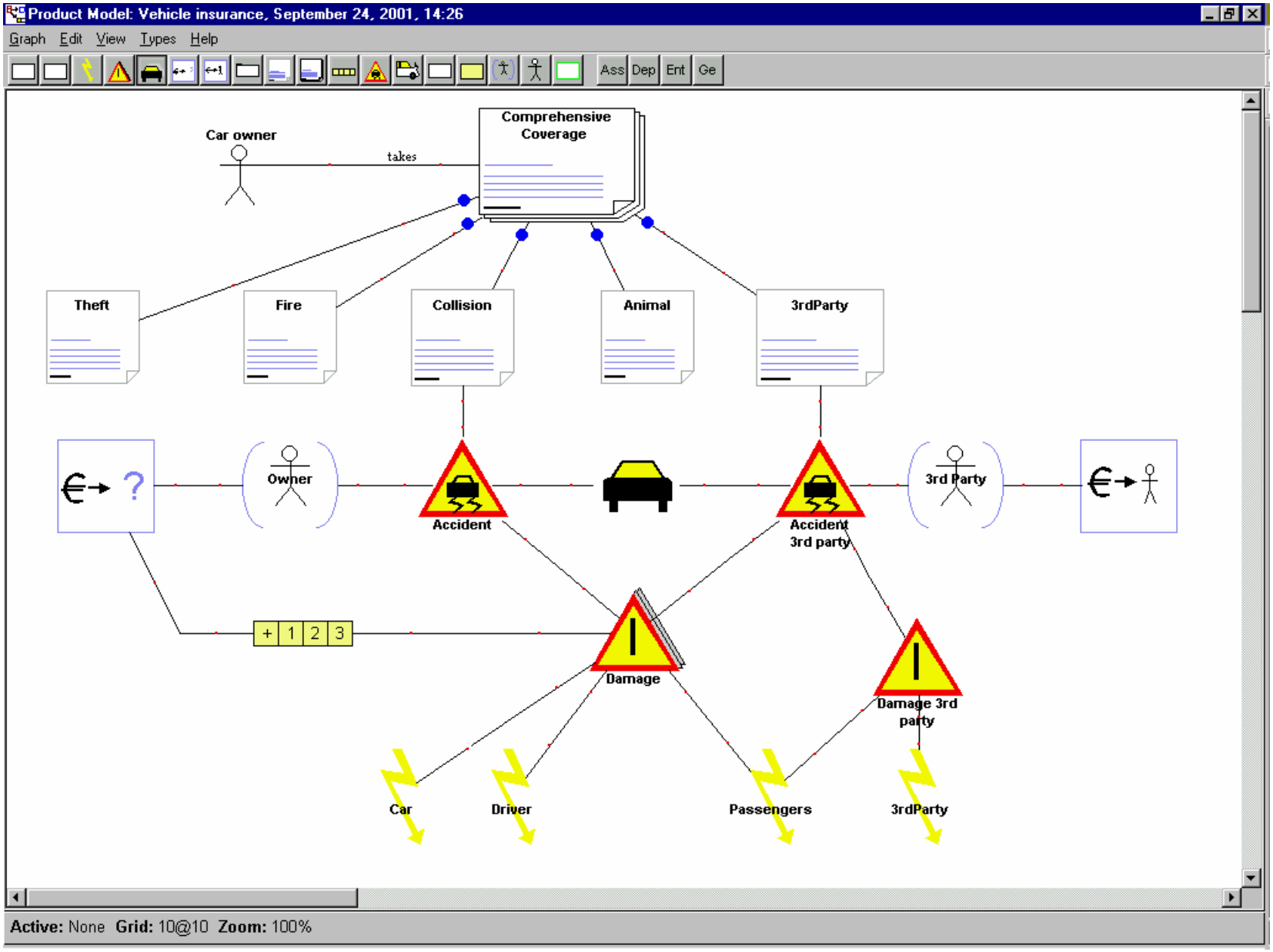
Let’s see some further examples on different application domains…

© 2003 MetaCase

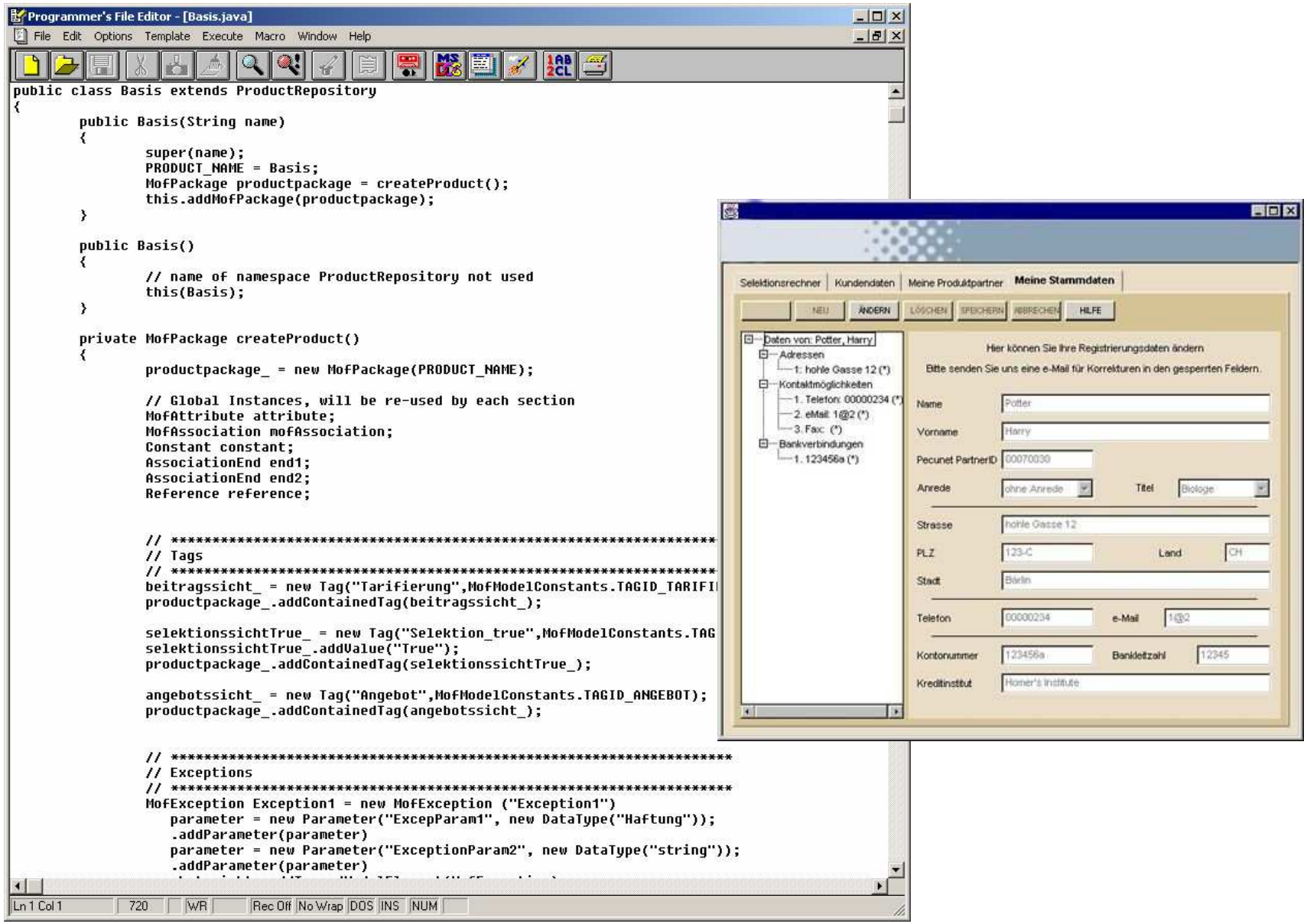
Case1:
Insurance products & eCommerce

- 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

© 2003 MetaCase



© 2003 MetaCase



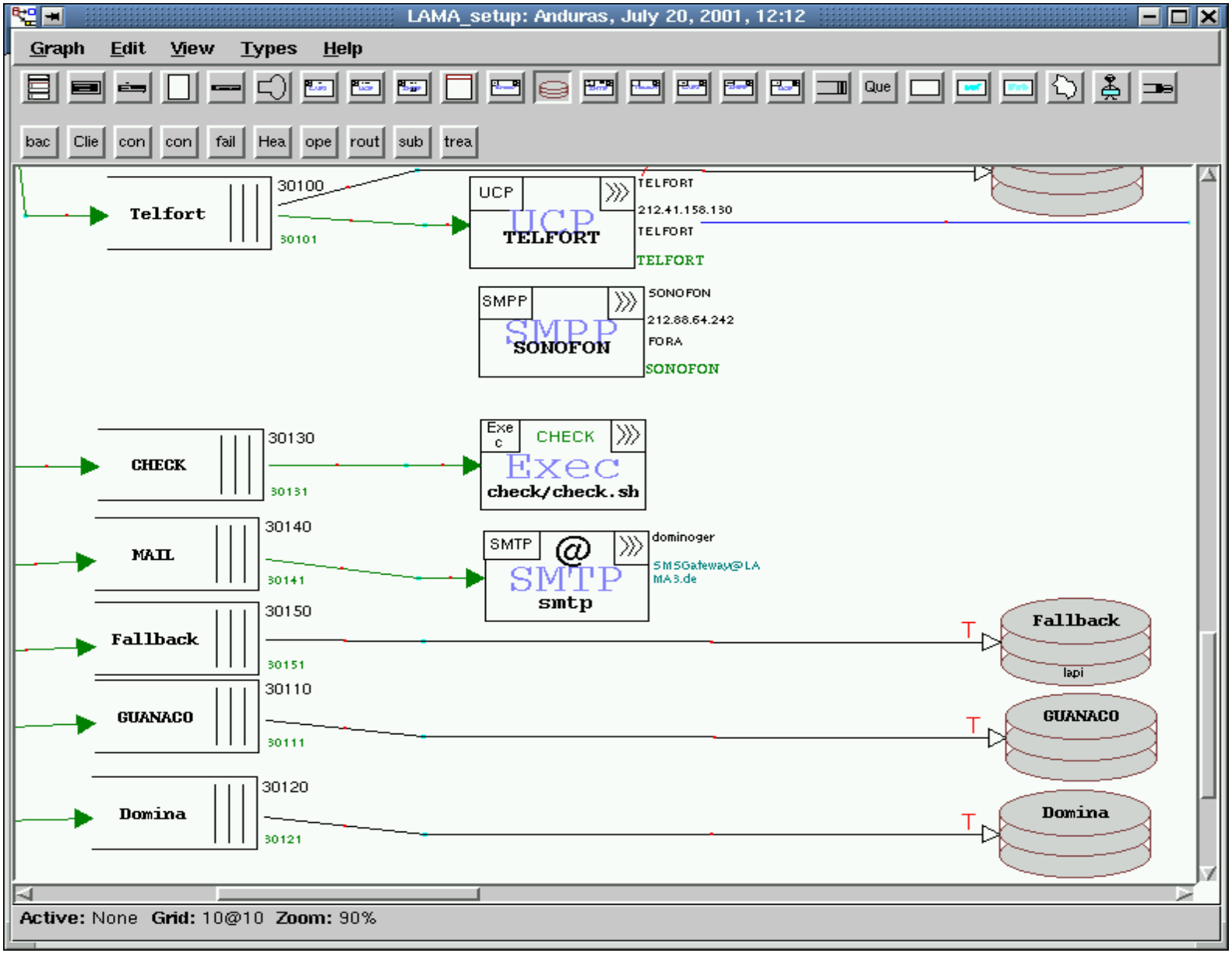
© 2003 MetaCase



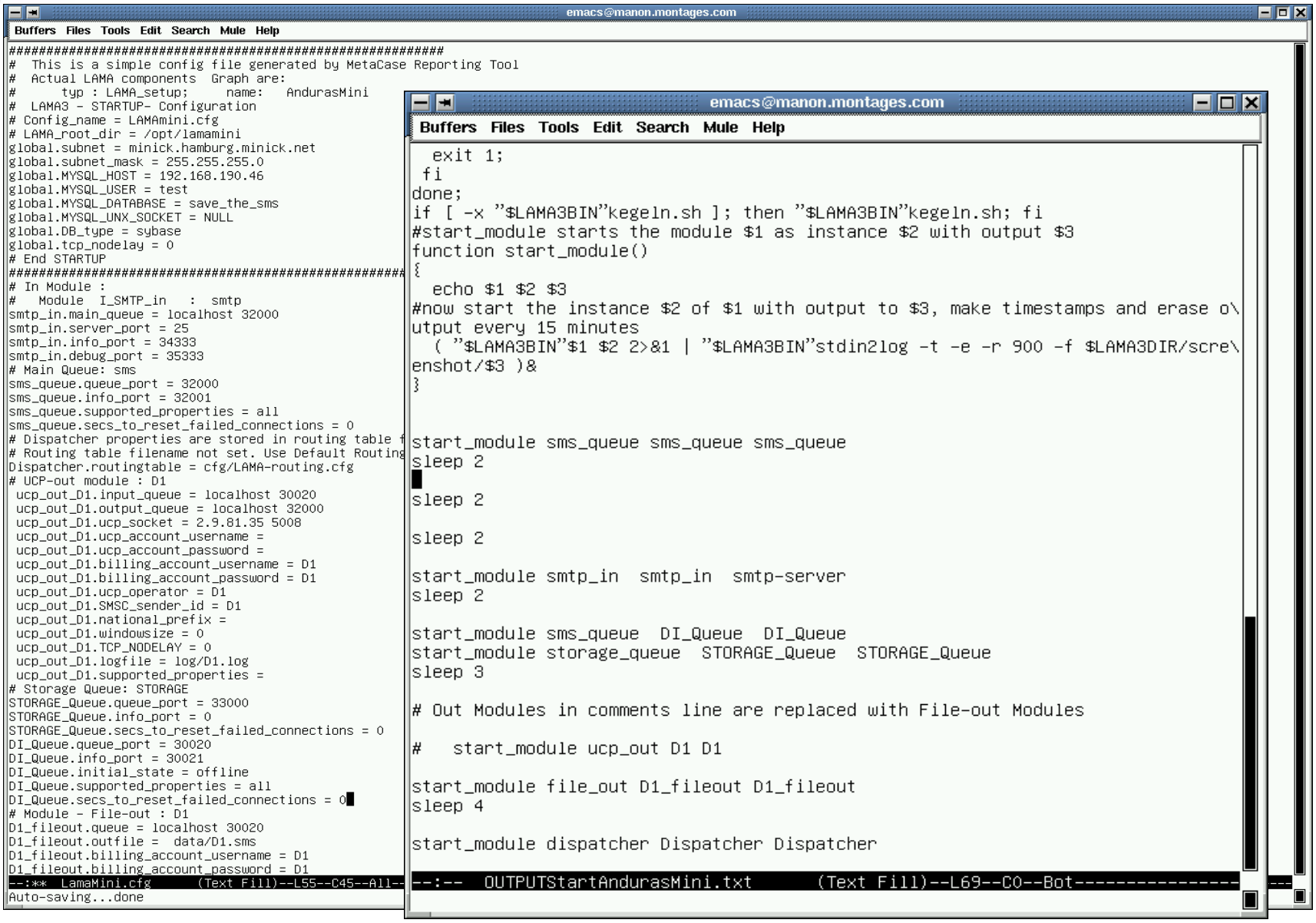
Case2: Configuration of services

- 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 refer to external files

© 2003 MetaCase



© 2003 MetaCase

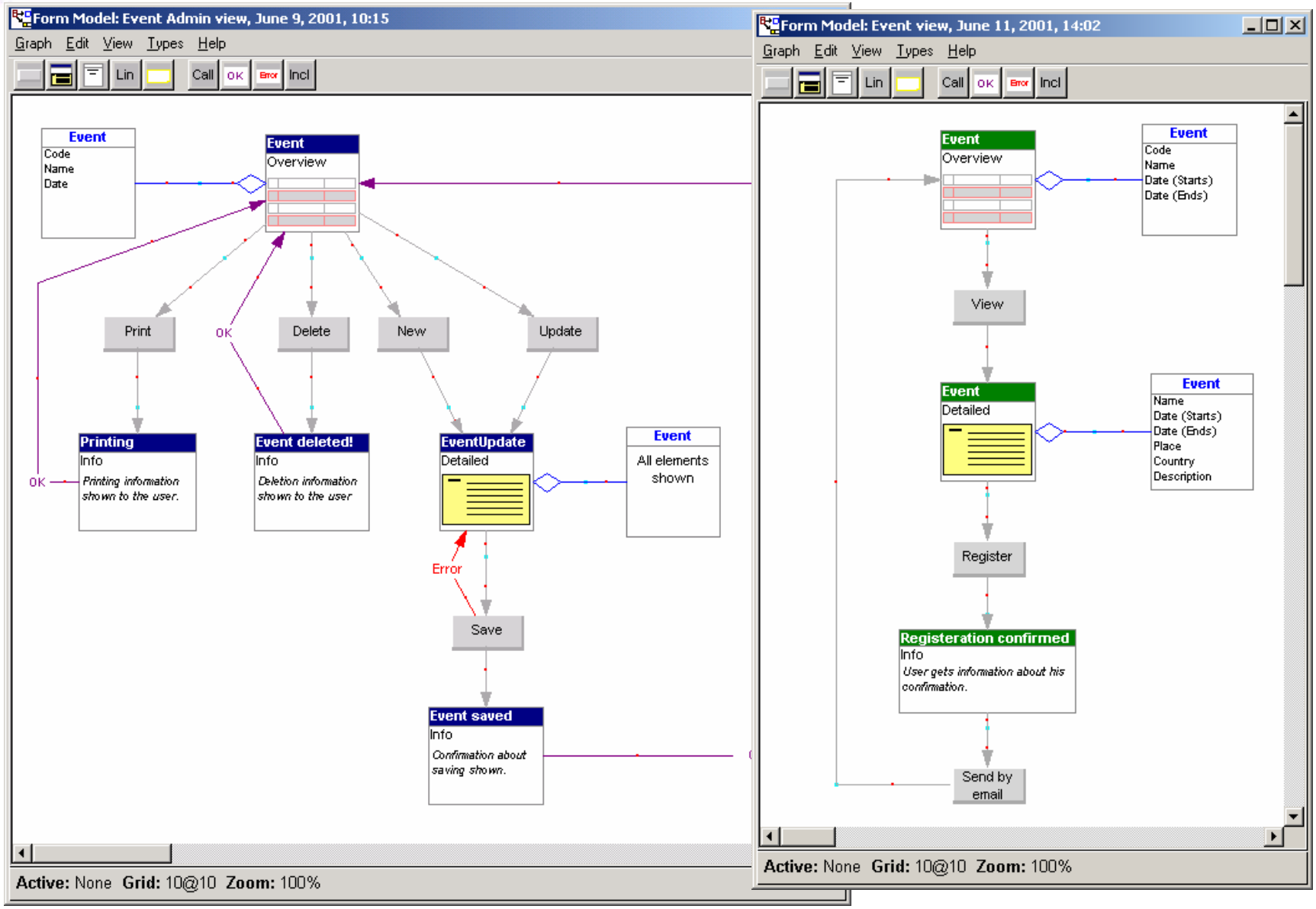


© 2003 MetaCase

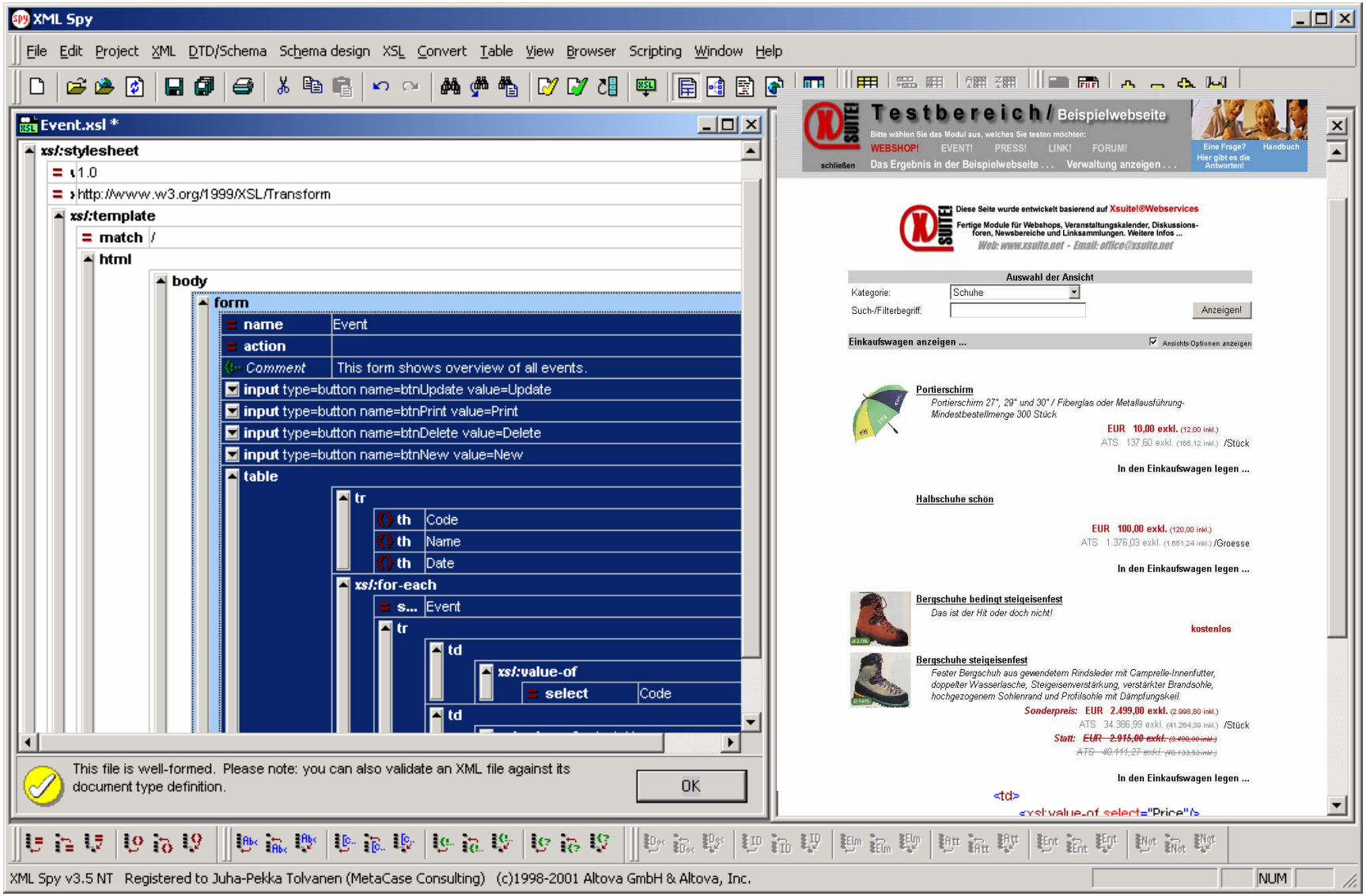
Case3: Web application

- 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

© 2003 MetaCase



© 2003 MetaCase



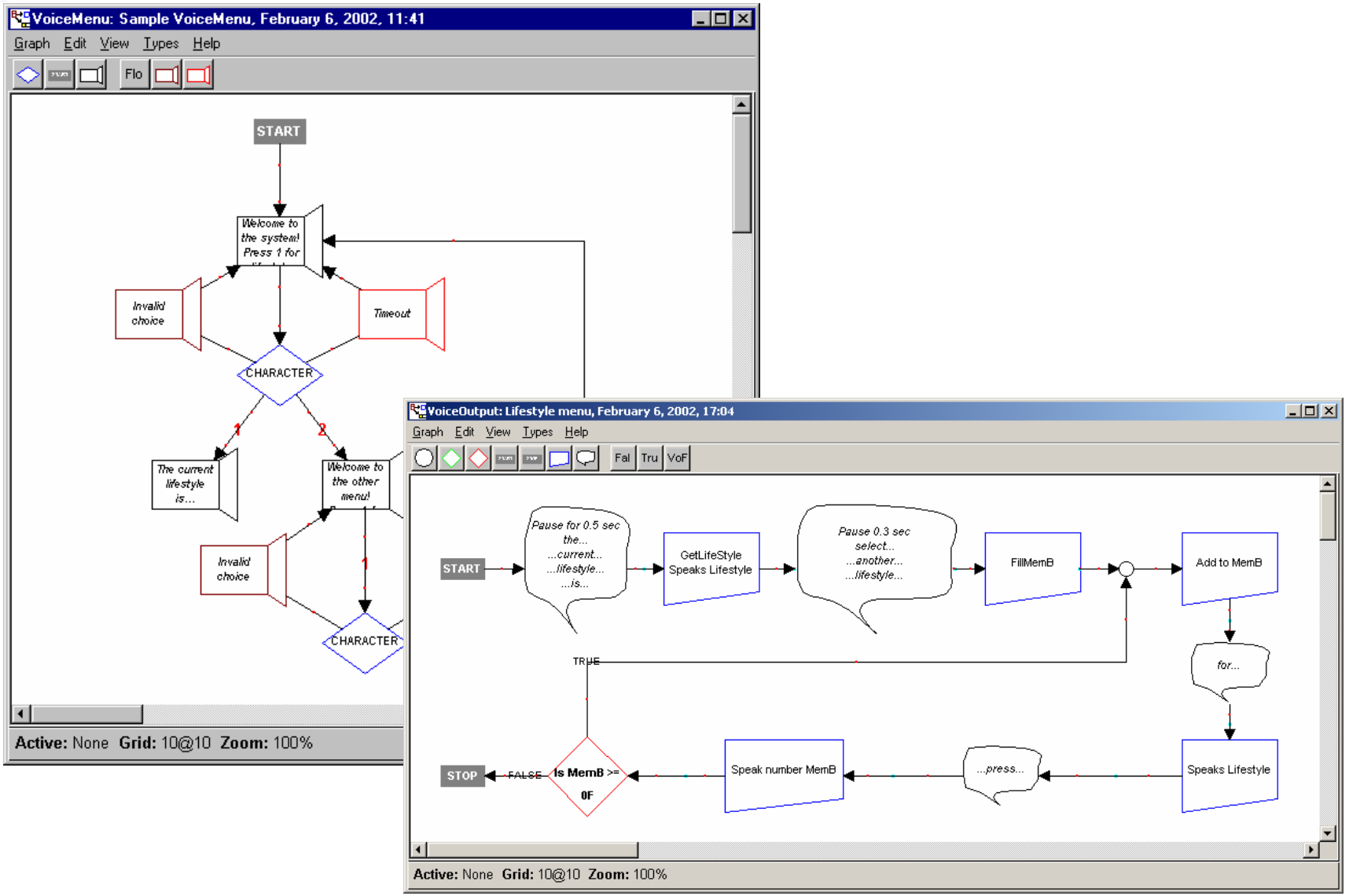
© 2003 MetaCase



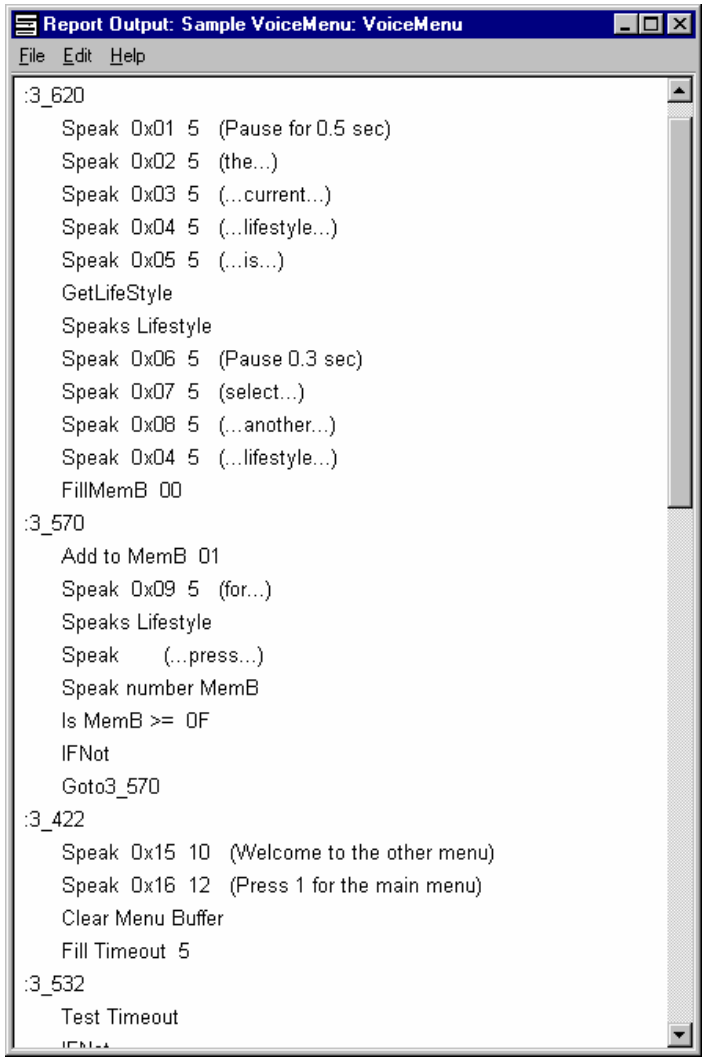
Case4: VoiceMenu for micro controller

- VoiceMenu for micro controller based home automation system
  - Remote control for lights, heating, alarms, etc.
- VoiceMenus are programmed straight to the device with assembler-like language (8bit)
- Modelling 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 down to a day!

© 2003 MetaCase



© 2003 MetaCase



© 2003 MetaCase



Let’s look then at the organizational perspective …

© 2003 MetaCase



Some reported industrial cases based on DSM

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

© 2003 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, 1999

© 2003 MetaCase

DSM Case Study: Lucent

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

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

© 2003 MetaCase



DSM case study: USAF

Development of message translation and validation system (MTV)*

- Declarative domain-specific language
- + code generators and customisation 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

© 2003 MetaCase



DSM Case Study: Nokia Tetra

- DSM for Tetra terminals*
  - Earlier development was based on SDL and manual coding
- Faster development
  - *"DSM+ has clearly boosted development speed due to its capabilities for supporting extensive reuse and managing variation"*
- Quality of product code improved
  - *"DSM rules out errors, eliminating them already in the design stage"*
- Hiding complexity made development easier
  - *"Developers with less C code experience can effectively develop features using DSM"*

* MetaCase, Nokia case study, 2003

© 2003 MetaCase

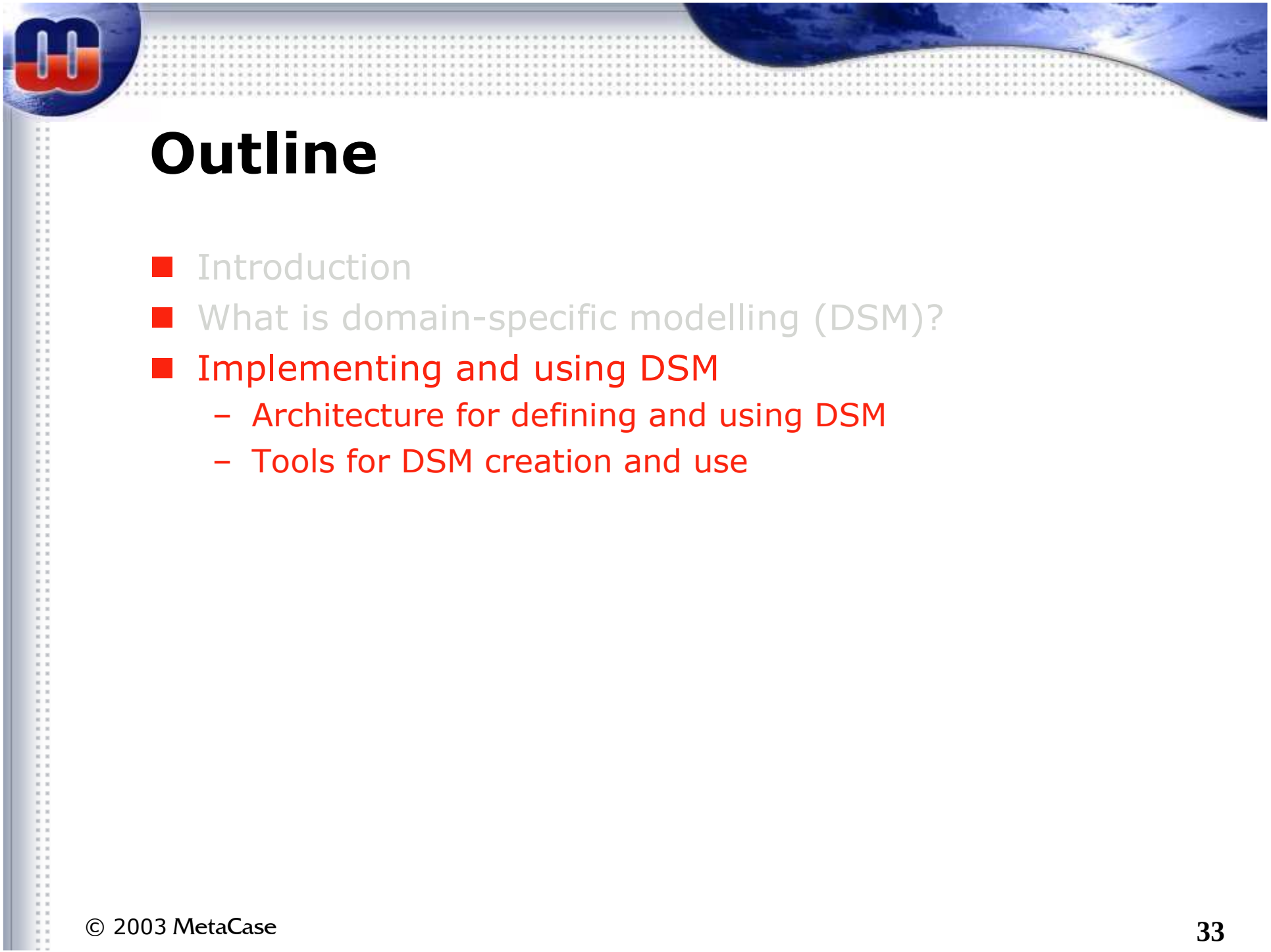


Where to apply

- Repetitive development tasks
  - Large portion of the work similar to earlier done (or done in parallel)
- Domain expertise needed
  - Non-programmers can participate

- These normally include:
  - Product Family
  - Platform-based development
  - Configuration
  - Business rule definitions
  - Embedded devices

© 2003 MetaCase

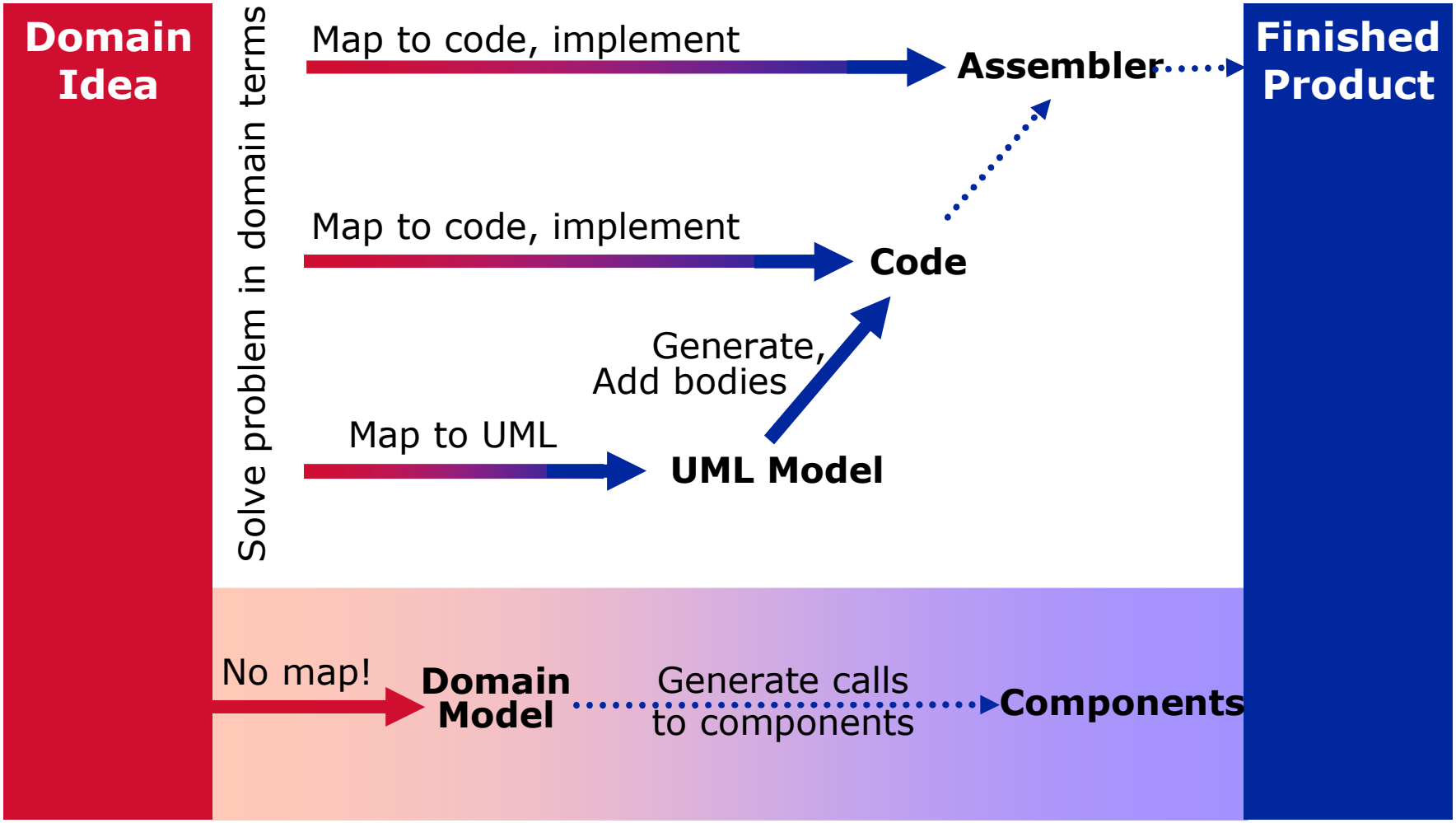


Outline

- Introduction
- What is domain-specific modelling (DSM)?
- Implementing and using DSM
  - Architecture for defining and using DSM
  - Tools for DSM creation and use

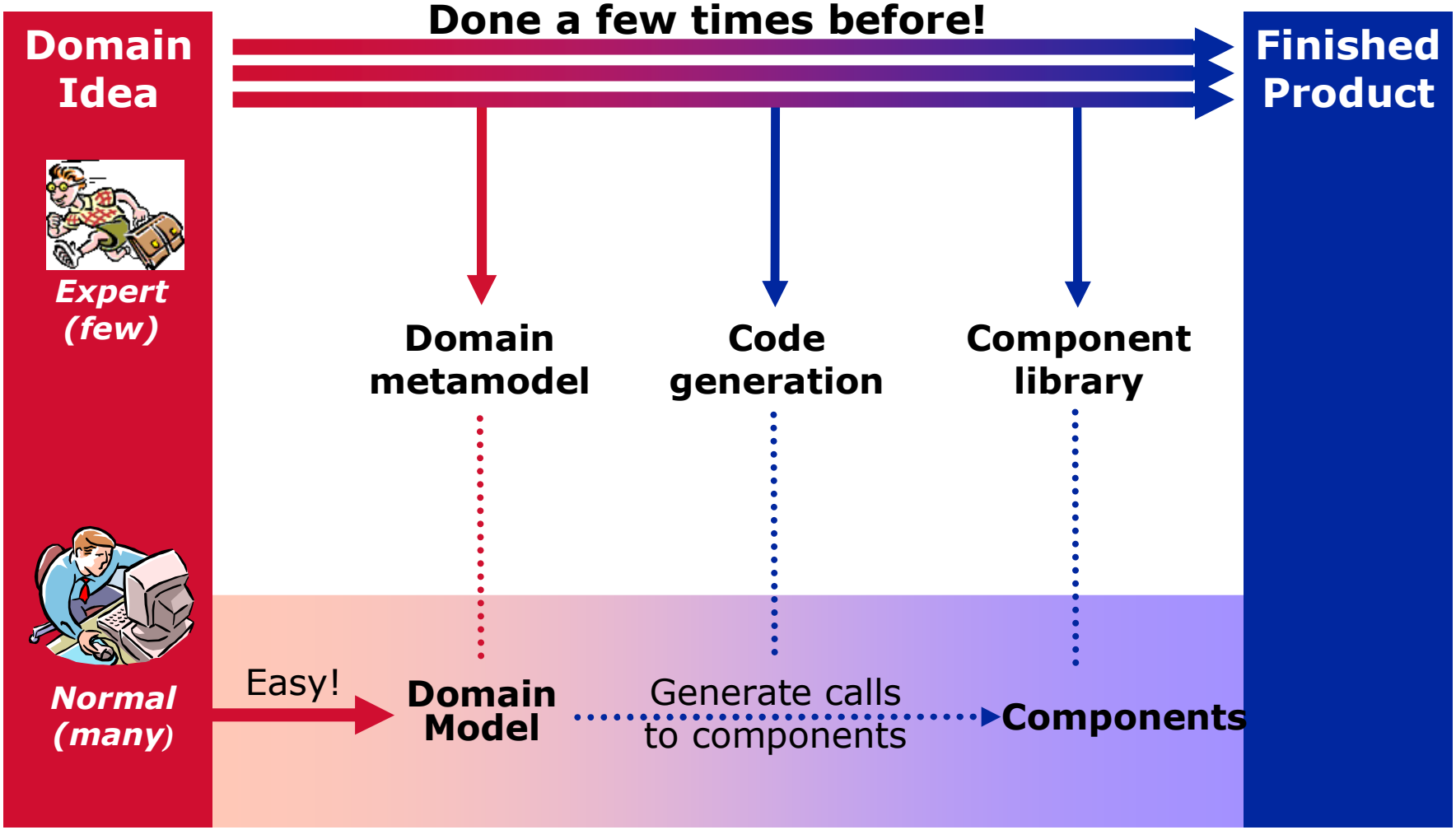
© 2003 MetaCase

Modelling domain vs. modelling code



© 2003 MetaCase

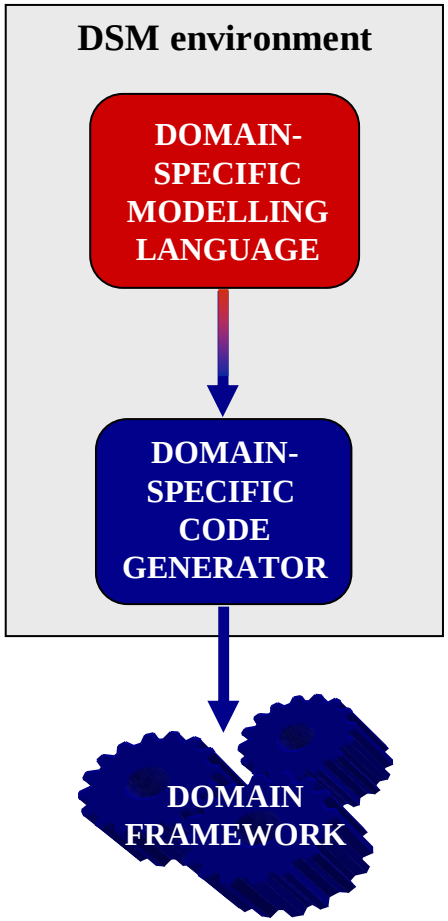
How to implement DSM



© 2003 MetaCase

DSM environment

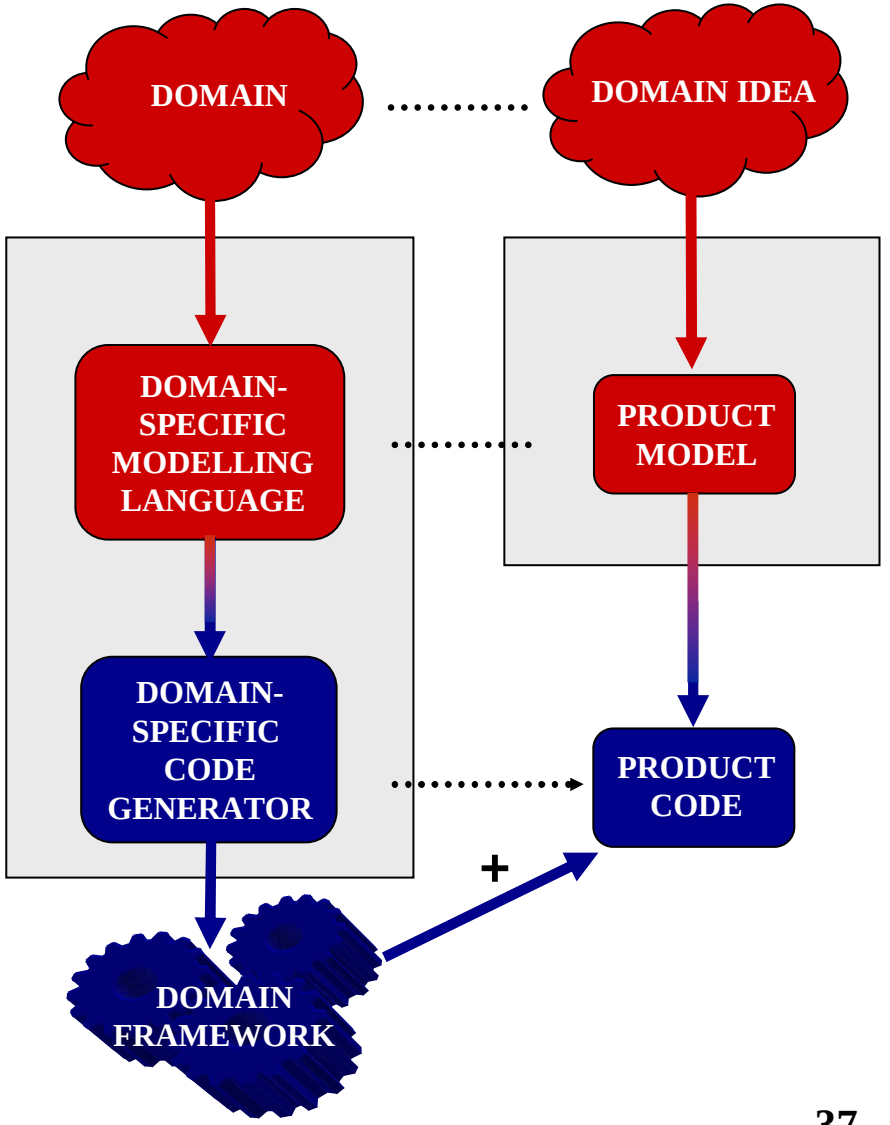
- Three things are required for a complete DSM environment:
  1. Domain-specific modelling language
     - Metamodel (of the language) maps to application domain (not to coding concepts)
     - Models describe only allowed design space
  2. Code generator(s)
     - Generators read the instance models to produce code
     - Provide variation for output formats
  3. Domain framework
     - Include common aspects used as primitives/components/platform services
     - Called by the generated code
- No single place to put the automation rules!



© 2003 MetaCase

DSM environment in use

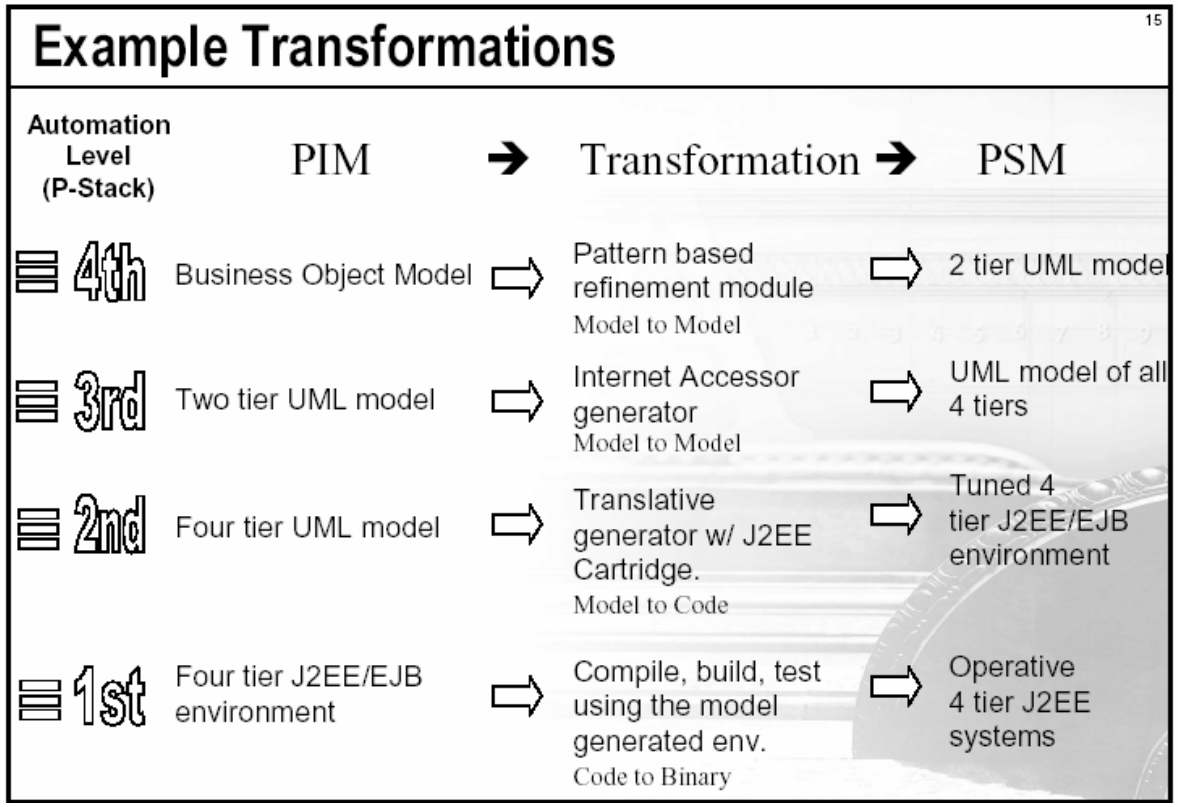
- **Few** experts define the DSM environment
- **Many** application engineers use it for product development
  - Less work to ”write”
  - Code generated
  - Models easier to read and change
  - Safer process



© 2003 MetaCase

Difference to MDA?

- Same idea on using models and transformations, but...



*** ObjektSpektrum Newsletter: Koch, T., Reckziegel, J., MDA - State of the Art**

© 2003 MetaCase



DSM aim to provide...

- Abstraction higher than coding terms
  - not about return types, classes or their stereotypes
  - but about domain concepts directly
- Fully automated generation for the modellers
  - no need to change transformed models
  - no need to write code manually after generation
- Models are not based on UML, but to a language suited for both the domain and generation needs
- Simpler in terms of management and versioning of the models and generators

© 2003 MetaCase



Tools for DSM creation and use

- Tools are relevent for both DSM implementation and use
- Good tools...
  - Minimize resource use (few man-weeks)
    - create modelling tools and generators (semi)automatically
    - guide in DSM greation
    - allow you to test DSM already during domain design
  - Allow DSM to change and reflect changes
    1. to modelling tools
    2. to design models already made
- A variety of tools available
  - from Lex & Yacc to metamodel-based tools
  - CAME tools support metamodeling
  - MetaCASE tools integrate both modeling and metamodeling environment

© 2003 MetaCase



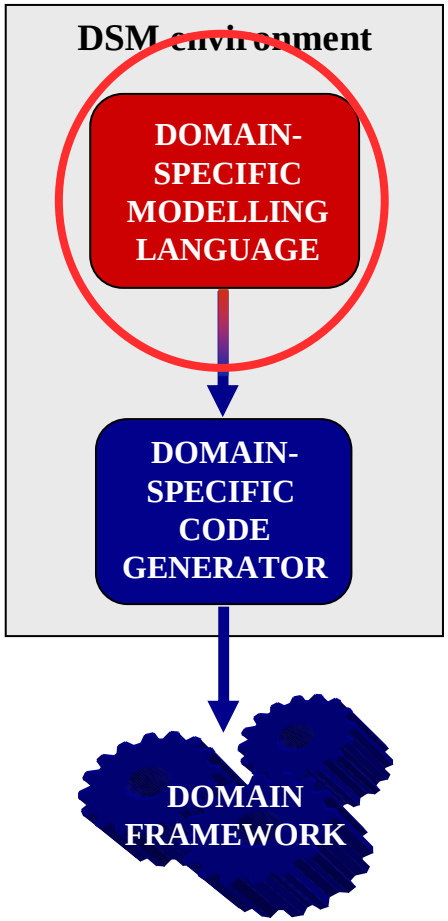
Outline

- Introduction
- What is domain-specific modelling (DSM)?
- Architecture for defining and using DSM
- Implementing modelling languages
  - Metamodels and metamodelling
  - How to identify modelling concepts and rules
  - Sample languages
  - Exercise: extending modelling language (i.e. metamodel)

© 2003 MetaCase

Implementing modelling languages

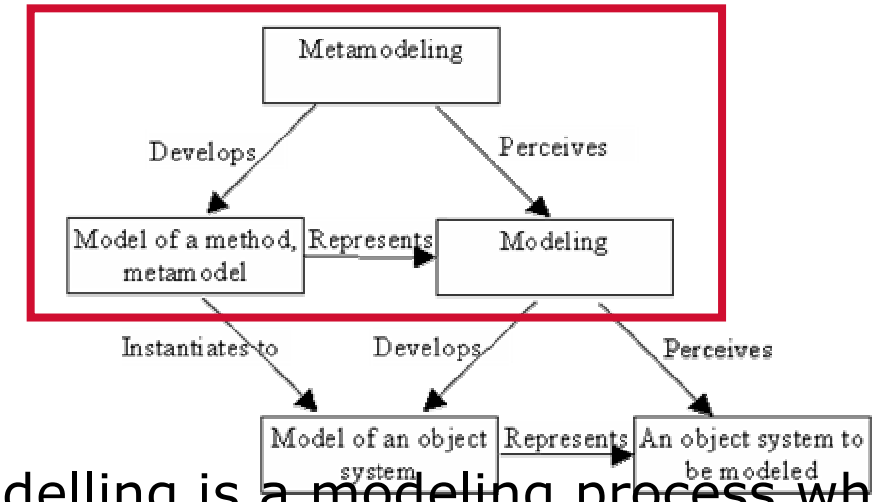
- The most important asset of a DSM environment
  - application engineers use it
  - generator and platform largely invisible
- Often defined by extending some type of a modelling language for the required computational model
  - state machine
  - flow model
  - data structure, etc.
- Language specified as a metamodel



© 2003 MetaCase

Metamodels and metamodeling

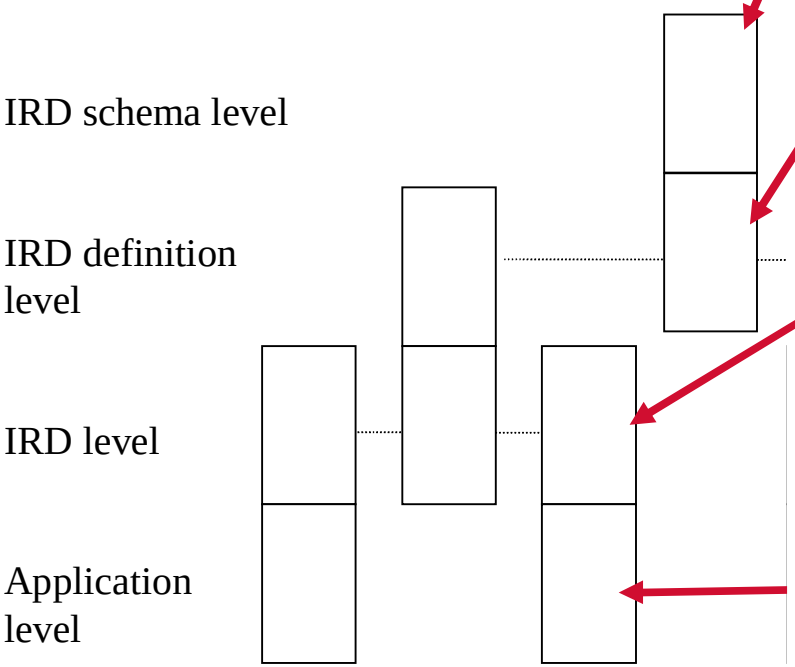
- Models play a crucial role in language definition, as in all engineering.
- A metamodel is a conceptual model of an method
  - here of a language only
- one level of abstraction higher than "normal" models



- Metamodelling is a modeling process which takes place at one level of abstraction and logic higher

© 2003 MetaCase

4 levels of IRDS by ISO

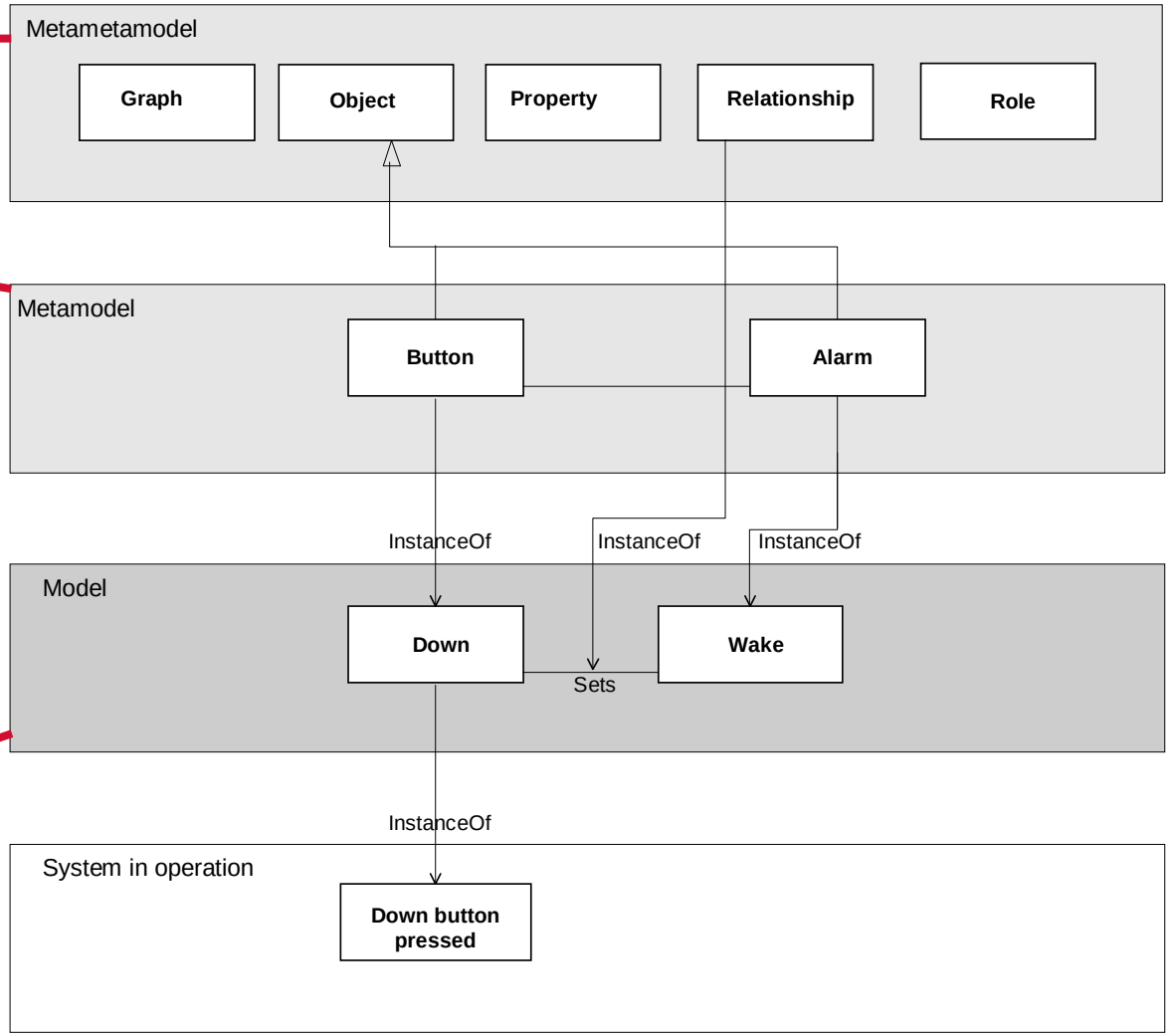


- **The IRD Definition Schema Level:**
  - A metametamodel
  - Metatypes: e.g. GOPRR's 'Property' (MetaCase), MOF's Attribute (OMG)
- **IRD Definition level:**
  - Schemata and application specs
  - Metaclass level of programming languages
  - Metamodels
  - e.g. specification of UML
- **The IRD level:**
  - 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"

© 2003 MetaCase

Metamodelling in OMG 4 levels

- GOPRR, MOF, etc.
- Language
  - ➔Concepts
  - ➔Rules
  - ➔Symbols
  - ➔Generator definitions
- Product specifications



© 2003 MetaCase



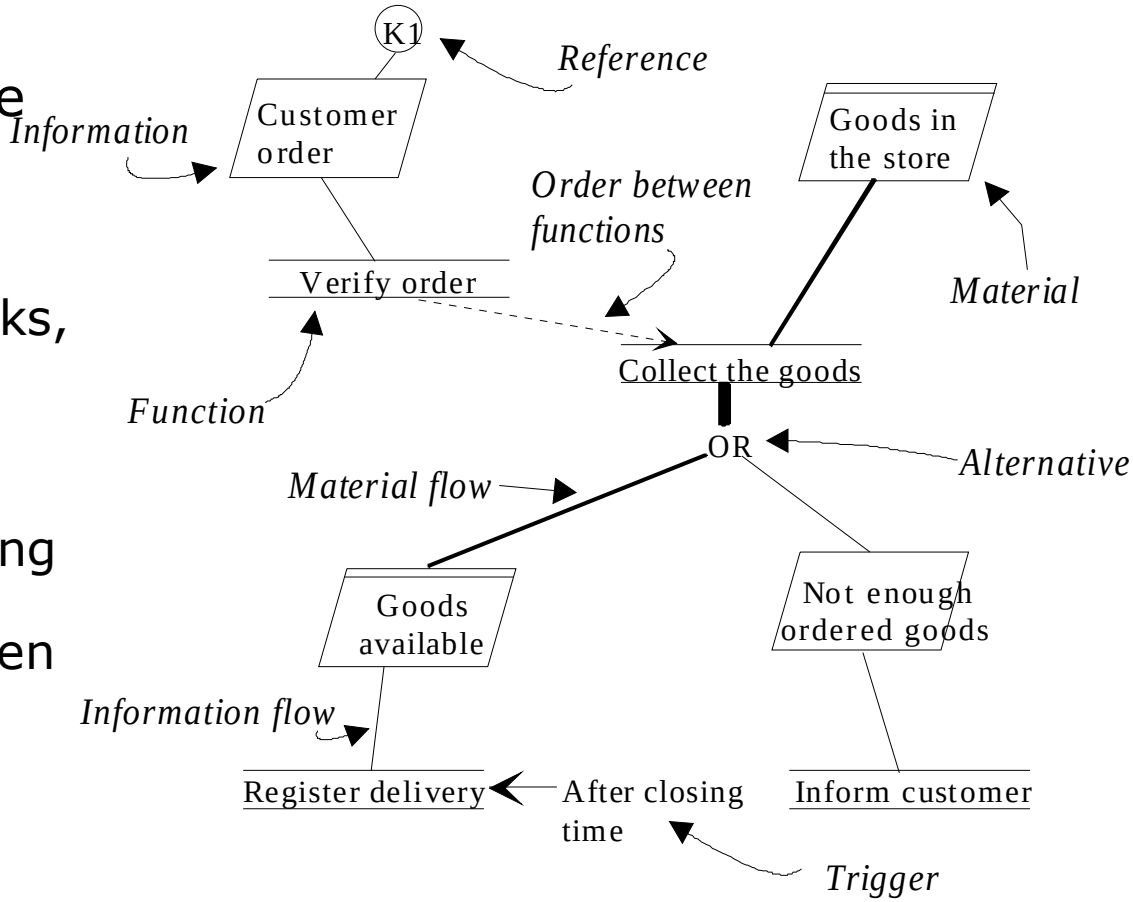
Metamodelling languages

- **Metamodelling is based on languages too!**
- These vary from purpose
  - illustrating vs. formalizing methods
  - defining tool support
    - Repository datamodel
    - Define tool functionality
- What kind of representation for metamodels
  - graphical (ER, NIAM, OPRR, GOPRR, MOF)
  - matrix (O/A Matrix)
  - text (ObjectZ, MDL, MEL, MOF)
  - form based (OPRR, GOPRR)

© 2003 MetaCase

An example: identifying language constructs

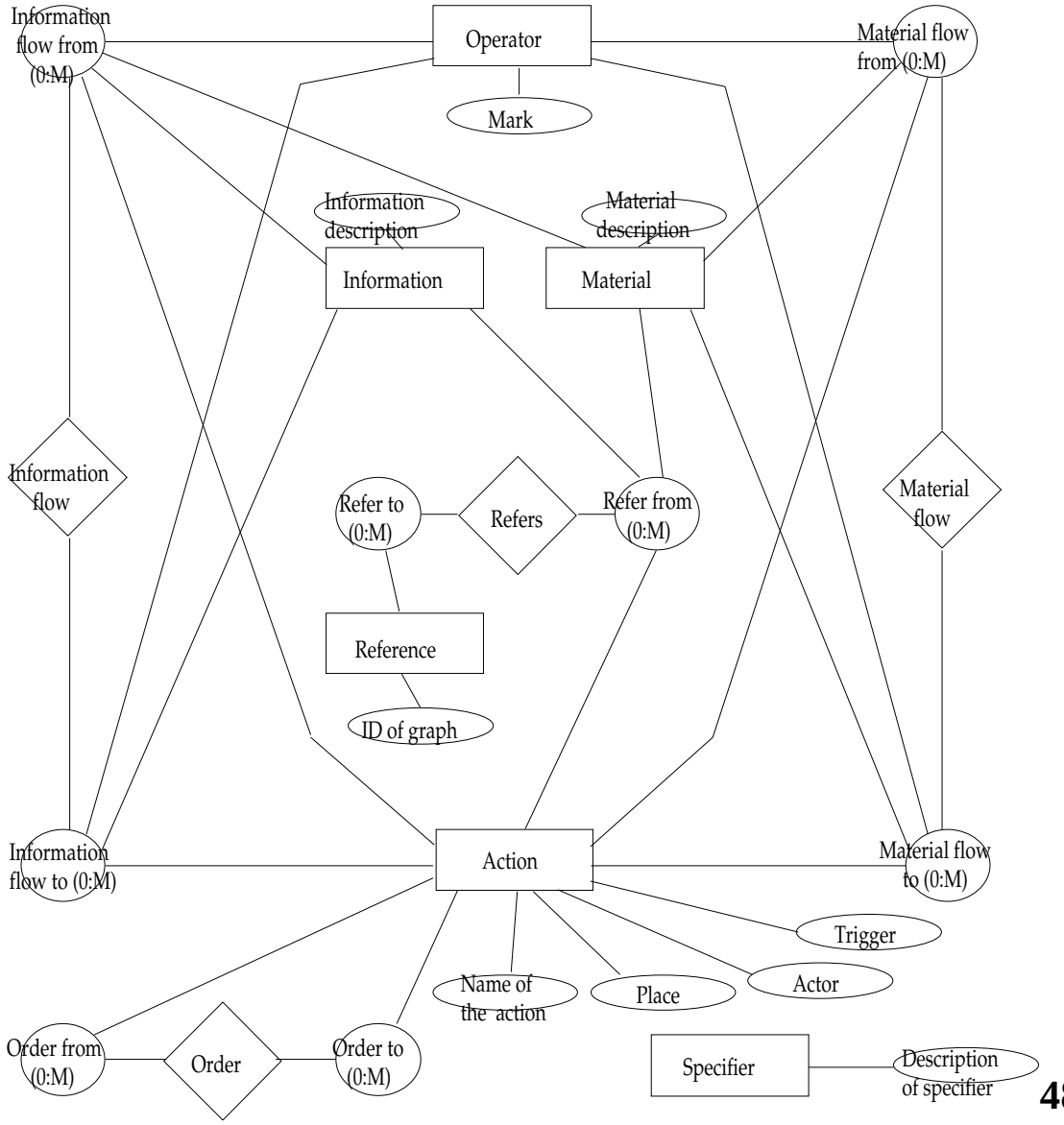
- Instance (model)
- What is the structure of the language for modeling:
  - activities/tasks,
  - order between tasks,
  - material flow,
  - information flow,
  - rules for flows,
  - triggers for initiating tasks,
  - (references between models)



© 2003 MetaCase

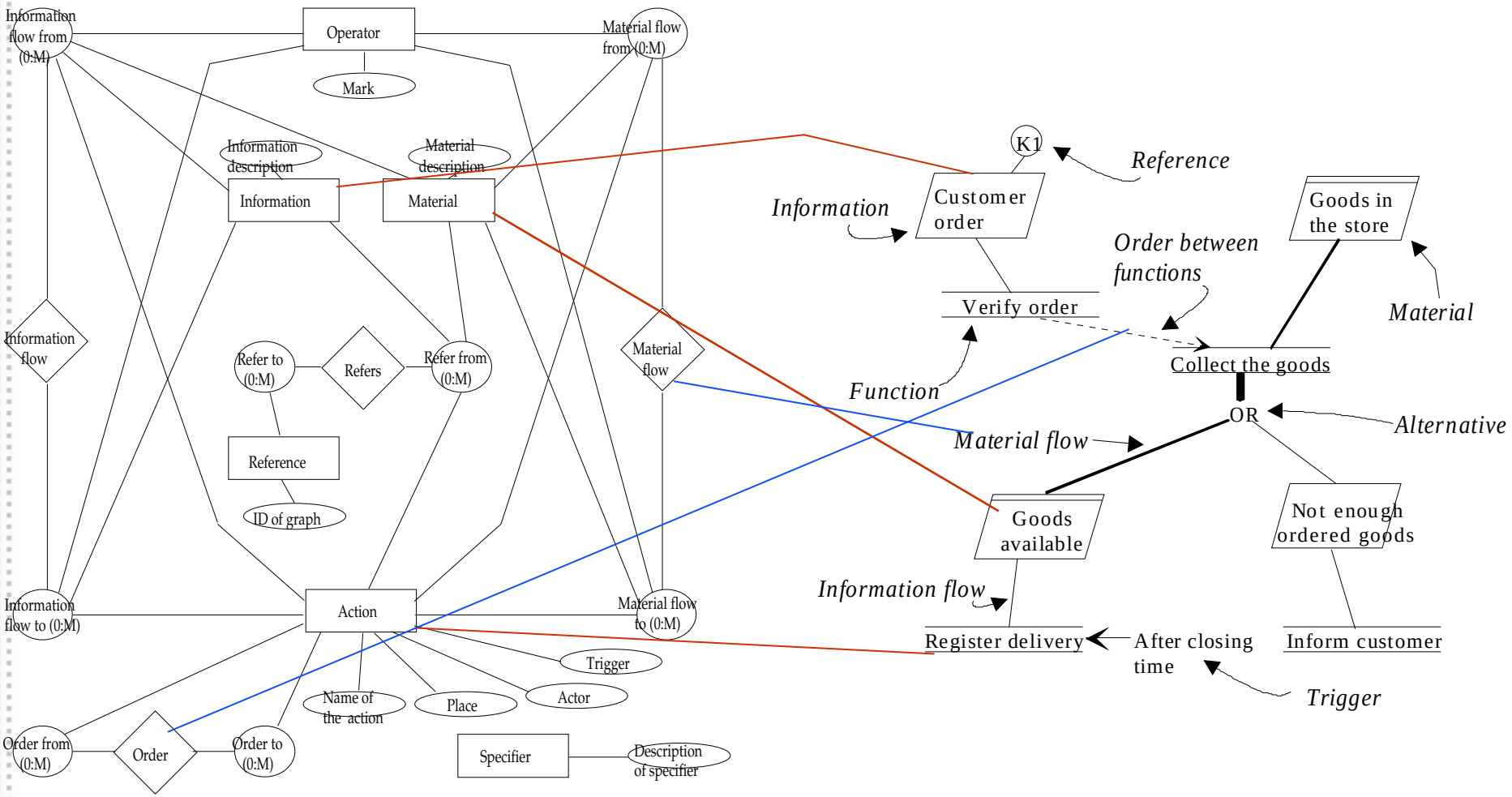
An example: metamodel

- Types (metamodel)
- Includes all the type information:
  - object (types)
  - relationship (types)
  - property (types)
  - ...



© 2003 MetaCase

Together: type-instance pair



© 2003 MetaCase



Identifying DSM constructs [1/3]

- Use domain concepts directly as modeling constructs
  - already known and used
  - established sematics 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 modelling
  - do not visualize product code!
    - better to ”forget” your current code
- Apply suitable computational model(s) as a starting point

© 2003 MetaCase



Choose computational model for DSM [1/2]

- How to specify all required design space?
- A variety of models available
  - Data models, state machine, process models, interaction, flow, etc.
- Often several models needed
- One model can incorporate aspects of other models, e.g:
  - state models can describe some static structures
  - data models can be extended for behavioural aspects (e.g. order)

© 2003 MetaCase



Choose computational model for DSM [2/2]

- Computational model(s) depends also on the output format and domain framework, e.g.
  - if most behavior is provided by the framework we need to produce only interfaces
    - suggests the use of datamodels (+extensions)
  - if dynamic behaviour of the system is expected to be modelled (and generated)
    - suggests the use of state models (+extensions)
  - if expected output is in XML, data modeling adequate

© 2003 MetaCase



Identifying DSM constructs [2/3]

- Enrich chosen computational models with domain-specific concepts and rules
  - look 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 then early as a metamodel
  - implement in some metamodel-based tool
  - test it with real models

© 2003 MetaCase



Identifying DSM constructs [3/3]

- Type level = in the language
  - best place giving first class support by the language
  - modelers don't need to know them
  - easier to be used by the generator
  - example: all metamodels
- Instance level = in the models
  - every developer needs to know them
  - how do generators know if they are filled correctly?
  - examples:
    - comments and note fields
    - naming conventions
    - profiles (stereotypes, tagged values)
    - model marking
    - constraint languages

© 2003 MetaCase

Example: Digital wrist watch

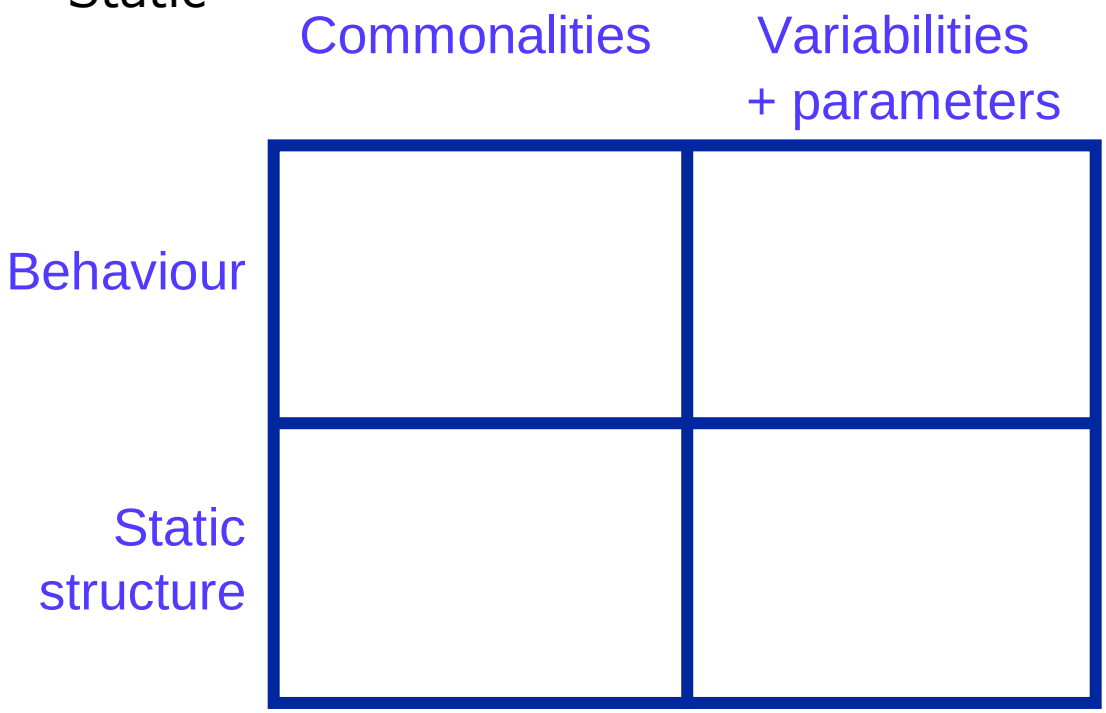
- We want to design new watch models and generate 100% of the variant implementation automatically
- Watch family has common
  - Terminology
  - Reference requirements and architecture
  - Reusable components
- So what kind of language (metamodel) we should have?



© 2003 MetaCase

Identifying modelling concepts

- Analyze architecture
- Analyze commonalities and variabilities
  - Behaviour
  - Static



© 2003 MetaCase

Behaviour commonalities

- The following statements are basic behavioural assumptations about the digital wrist watch domain:
- Behaviour

  C1 Watch applications display time units and icons

  C2 Watches execute applications in certain order

  C3 User can trigger actions with buttons

  C4 Actions operate on time units, icons, and/or alarms

  C5 Icons are shown or hidden

  C6 Alarms trigger an action

© 2003 MetaCase



Static commonalities

- The following statements are basic structural assumptations about the digital wrist watch domain:
- Static structure

C7 Watch has a physical display with zones for displaying time and icons

C8 Time is divided into two-digit time units like seconds and minutes

C9 Watches have a set of buttons

C10 Watches provide time-based applications

© 2003 MetaCase



Behaviour variabilities

- The following statements describe how watches can vary:
- Behaviour
  - V1 Time units and icons displayed vary depending on the application
  - V2 Actions applied vary depending on the application
  - V3 The sequential order of accessing applications can vary
  - V4 Watches can be executed on different platforms

© 2003 MetaCase



Static variabilities

- The following statements describe how watches can vary:
- Static structure
  - V5 The number of applications can vary
  - V6 Applications can be constructed from other (sub)applications
  - V7 The number of zones in a display unit can vary
  - V8 Unit of time shown in a zone can vary
  - V9 The number of icons shown in a display unit can vary
  - V10 Icon label text can vary
  - V11 The number of buttons can vary
  - V12 Button label can vary

© 2003 MetaCase



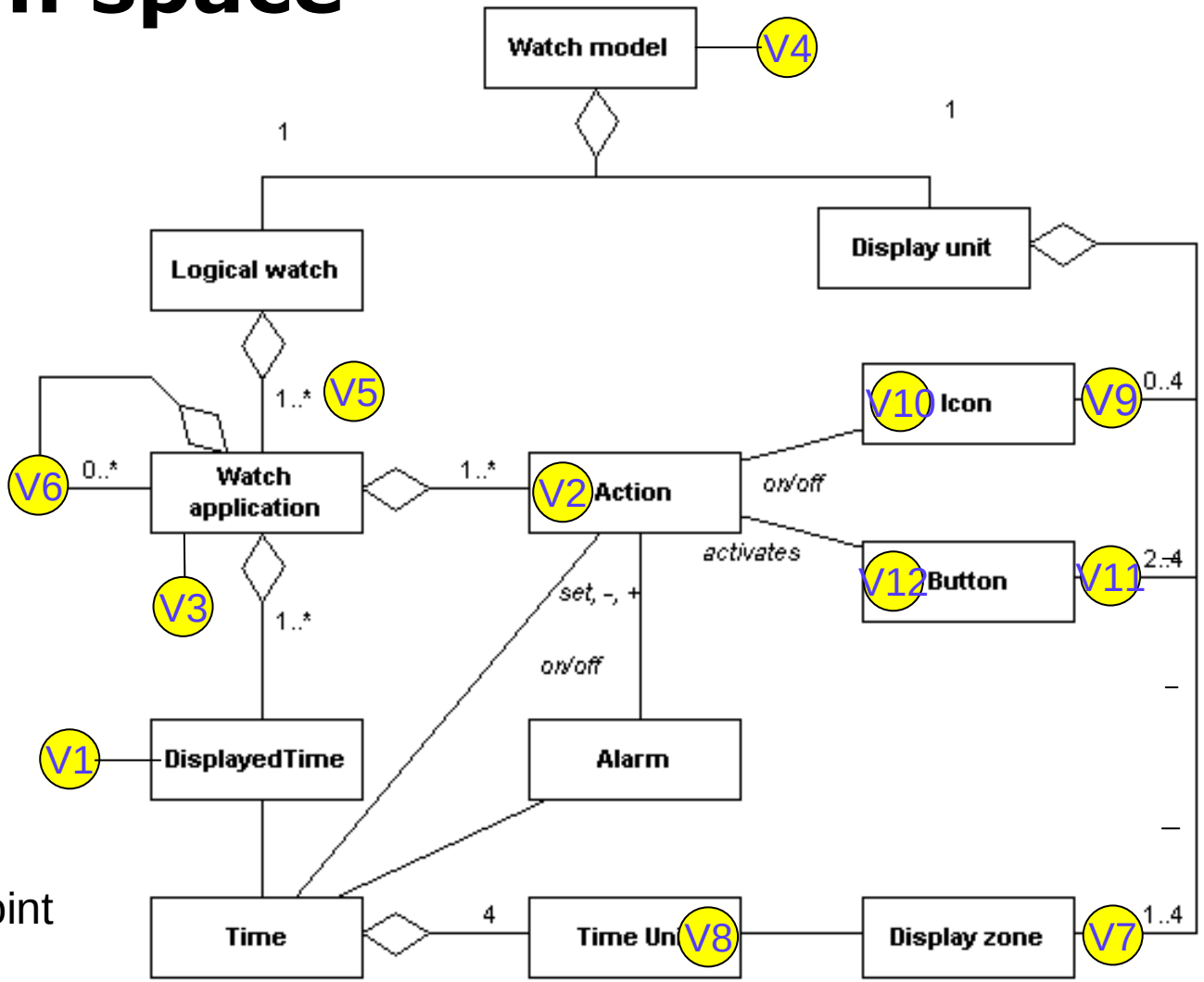
Variability space

- Variation space with variation parameters

Variability	Description	Parameter
V1	Time units and icons displayed can vary depending on the application	Displayed Times (0-N) + Displayed Icons (0-N)
V2	Actions applied vary depending on the application	[set Time, + Time, - Time, Icon on, Icon off, Alarm on, Alarm off]
V3	The sequential order of accessing applications can vary	Accessing order
V4	Watches can be executed in different OSs	[MIDP, Linux, Windows]
V5	The number of applications can vary	[1..N]
V6	Applications can be constructed from other (sub)applications	[0..N]
V7	The number of zones in a display unit can vary	[1..4]
V8	Unit of time shown in a zone can vary	[Hours, Minutes, Seconds, Hundreds of seconds]
V9	The number of icon slots shown in a display unit can vary	[0..4]
V10	Icon label text can vary	[Stopwatch, Alarm, World time, Timer]
V11	The number of buttons can vary	[2..4]
V12	Button label can vary	[Up, Down, Mode, Set]

© 2003 MetaCase

Watch family architecture and design space



Variation point

© 2003 MetaCas



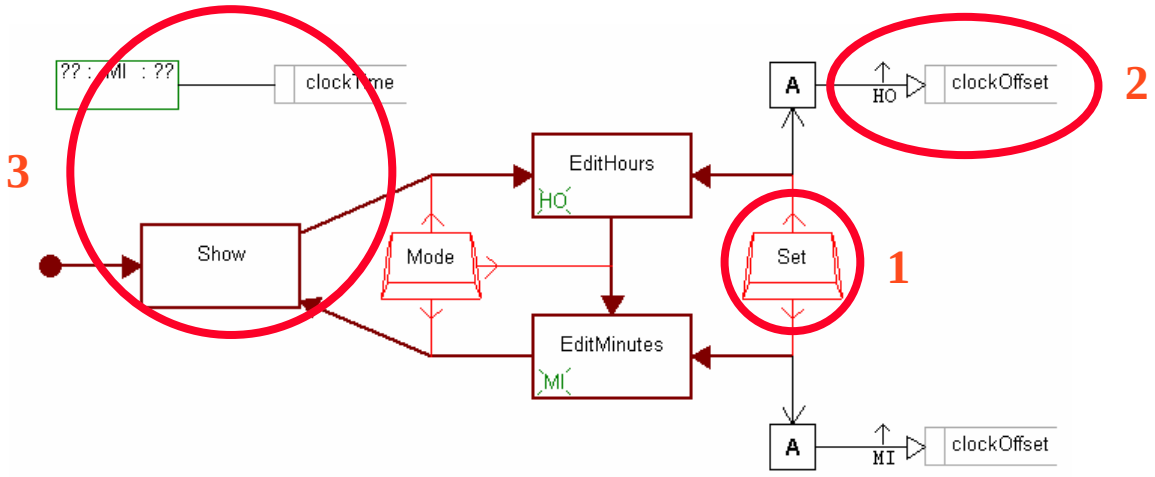
Modelling language for watch family

- Watch application as a state machine
  - V2: Actions executed in state transitions
  - V3: State transitions
  - V5: Number of state models
  - V6: Nested state models
- Display structure as a data model
  - V7: Display unit aggregation to zone
  - V8: Time unit property (selection list)
  - V9: Display unit aggregation to icon
  - V10: Icon property (selection list)
  - V11: Display unit aggregation to button
  - V12: Button property (selection list)
- TimeDisplay mappings as a data model
  - V1: DisplayUnits with Zones and shown time

© 2003 MetaCase

Language for logical watch [1/2]

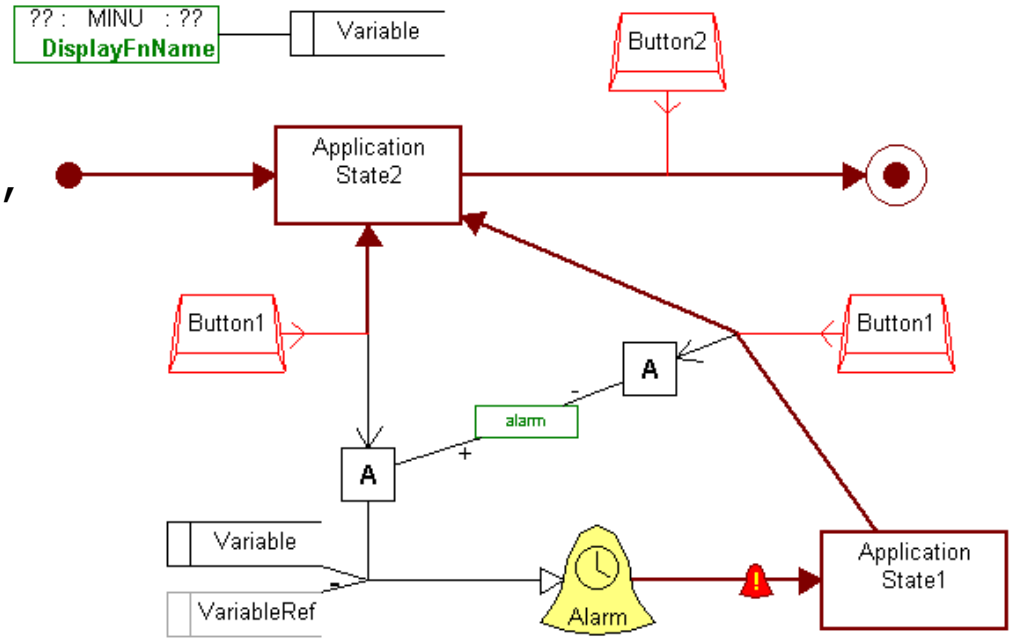
- State machine as a basis
- Extended with concepts and rules of watch domain, like:
  1. pressing button triggers transitions
  2. Actions in transitions operate only with time elements
  3. Each state describes time in specified way



© 2003 MetaCase

Language for logical watch [2/2]

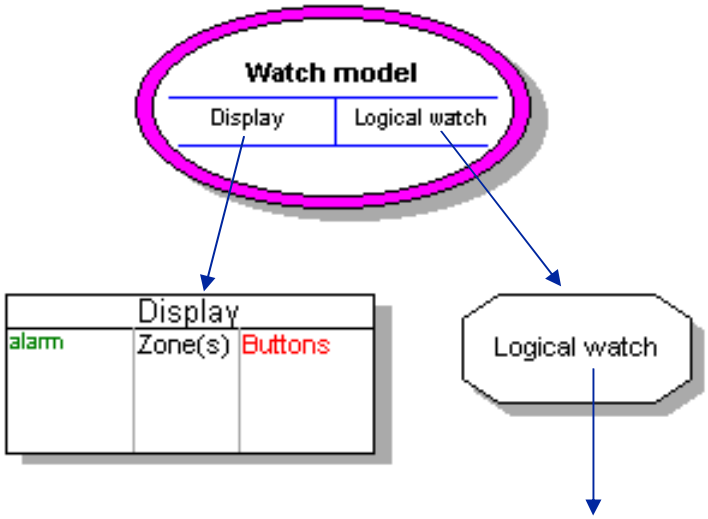
- Other extensions to state modelling concepts
  - Application state, Action, Alarm, Button, Icon, Displayed time, Variable
- To relationships
  - Action performs operations for icons,
  - Actions can set Alarm,
  - Alarm reads time to trigger application state,
  - Start state transition do not need triggers, etc.
- Figure: Metamodel by example



© 2003 MetaCase

Language for watch model and display structure

- Static structure in a data model
  - Data model describes one watch model
  - Modelling elements
    - Logical Watch, Display, Button, Icon, DisplayZone
  - Element structures and relations
    - Watch model <Logical Watch, Display>, Display <{Button}, {Icon}, {DisplayZone}>, Logical watch <Watch application>



Applications as state machine

© 2003 MetaCase



Implementing DSM

- To have tool support the metamodel must be defined in the form the selected tool accepts
- Tool implementation requires also other issues than conceptal metamodel (concepts and rules)
- Symbols
  - notation, icons in toolbars
- Dialogs
  - user interface
- Generators
- Connecitivity with other tools
- Help and tutorials

© 2003 MetaCase



Let's look at the watch metamodel...

- Concepts of Watch Model and State
  - definition
  - properties
  - rules
  - dialogs
  - symbols

© 2003 MetaCase



Exercise: extending the modelling language

- What if we would need a heartbeat functionality
  - have an external HW
  - HW abstraction layer with following services
    - start, stop
    - read
    - set monitoring interval

© 2003 MetaCase



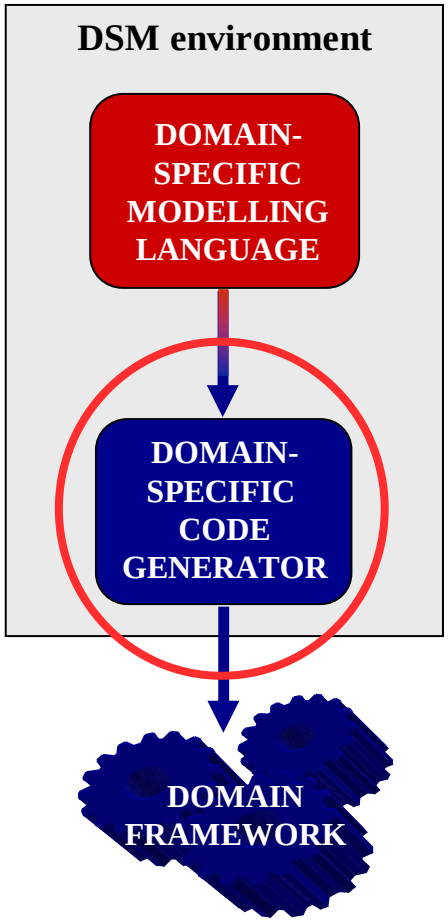
Outline

- Introduction
- What is domain-specific modelling (DSM)?
- Architecture for defining and using DSM
- Implementing modelling languages
- Implementing generators
  - Different generator approaches
  - The problems of fixed generators
  - Implementing domain-specific generators
  - Interfacing to generated code
  - Sample generators (C, Java, XML, Assembler)
  - Exercise: extending generator

© 2003 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
  - Customisable generators
  - Domain-Specific generators



© 2003 MetaCase



”Out-of-box” generators [1/2]

- Typical characteristics
  - Fixed metamodel (UML, EXPRESS, SA/SD, IDEF, etc.)
  - Limited possibilities to influence on generator execution (with parameters, input given in models, etc.)
  - Predefined platform support (e.g. ANSI C, certain processor model)
- Philosophy
  - Apply standard method and process
- E.g. generators included in traditional CASE tools
- It is questionable how much there is generation if class in model is transformed to a class in code?

© 2003 MetaCase



”Out-of-box” generators [2/2]

- Benefits
  - General purpose solution
  - Introduction and use requires relatively little resources
- Limitations
  - Generated code often incomplete (e.g. pure skeletons)
    → Requires manual changes
    - Models (other than pure code structure) and code not up-to-date
    - Often re-generation not possible without losing earlier changes
    - reverse-engineering works only for static structures
      → Rest of the designs (user view, dynamics, behavior, interaction etc.) maintained manually
  - Generated code not usable
    → Inefficient, need to be changed or rewritten
    - Typical in areas when only simulation is generated but not the product code
  - Works adequately in few cases only

© 2003 MetaCase



Customisable generators [1/2]

- Typical characteristics
  - Fixed metamodel (UML, etc.)
  - Developer can influence to the generator functionality (e.g. programming or scripting language, templates)
  - Platform support depends on the customisation mechanisms available
- Philosophy
  - Generator "intelligence" better because of improved capability to understand models (source)
  - Developer can extend variation handling by implementing it to the generator
    - as can't be put to the metamodel
  - Generators supported by raising abstraction in the platform
- E.g. generators in MDA
  - Models have profiles and marks to guide generators

© 2003 MetaCase



Customisable generators [2/2]

- Benefits
  - General-purpose solution
  - Better customisation support than ”out-of-box” generators
    - Works in more typical cases
    - Generated code more “compete” and “applicable”
  - ”Single source, multiple target” –approach possible
    → one model, multiple generators (for each platform)
- Limitations
  - Generators become easily large and complex (for each model instance variant situation)
  - Bigger investment in resource use than "out-of-box”–generators
  - Customisation limited because fixed source (metamodel)
    - Generator has limited possibilities to “invent” solutions that are not described in models

© 2003 MetaCase



Domain-specific generators [1/2]

- Typical characteristics
  - Customisable metamodel
  - Customisable generator definitions
  - Platform support dependent on the degree of customisation
- Philosophy
  - Modelling language enriched with domain concepts (types) supporting generator
    → variation managed with modelling concepts
  - Generators supported by raising abstraction in the platform
  - Generator definitions kept simple
- E.g. generators in metaCASE tools

© 2003 MetaCase



Domain-specific generators [2/2]

- Benefits
  - Support more complete customisation for given situation
  - 100% automated generation
    - No manual coding afterwards → models and produced code up-to-date
  - ”Single source, multiple target” –approach
  - Good support for reuse in domain-specific architectures
- Limitations
  - Requires expertise
  - Need resources for definition and maintenance
    - Fewer tools available

© 2003 MetaCase



Implementing domain-specific generators

- Generation is influenced by
  - Applied modelling language (i.e. metamodel)
    - concepts
    - rules
    - semantics
  - Chosen implementation platform
    - programming language + components (subroutines and interfaces)
    - Operating system
    - Hardware
- Generator should operate directly with domain concepts
  - Shift abstraction (not a class in model to a class in code)
  - Also implementation can refer to metatypes, instead of types or instances

© 2003 MetaCase



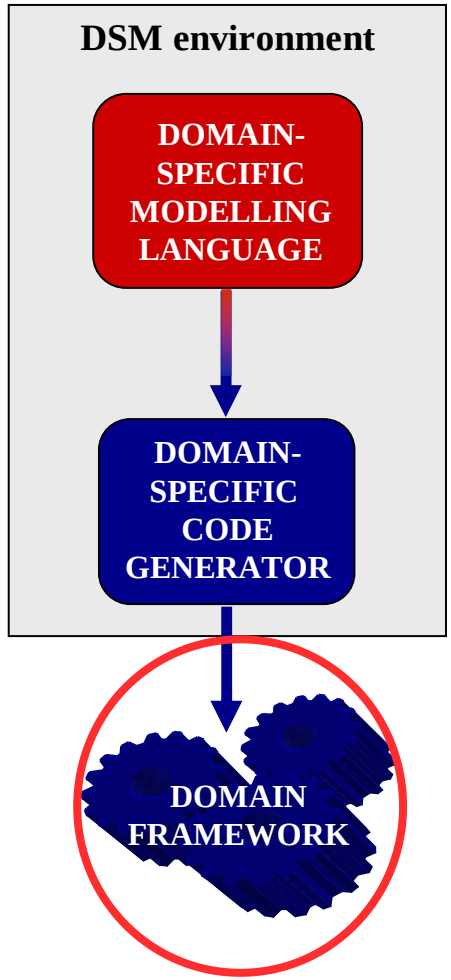
Implementing code generator [1/3]

- Keep generator (and generation process) as simple as possible
  - Raise variation handling into the modelling language (as data)
  - Push low-level implementation issues down to the framework
- Try to generate as little code as possible
  - Glue code only
- 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

© 2003 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)



© 2003 MetaCase



Implementing code generator [2/3]

- 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 modelling tool perspective

© 2003 MetaCase



Implementing code generator [3/3]

- Extensions to the generator
  - Checking completeness and uniformity
  - Autobuild → automating compile and execution
- Generators can be used also for other purposes
  - Configuration
  - Testing and analysis
  - Automated build
  - Documentation and review

© 2003 MetaCase



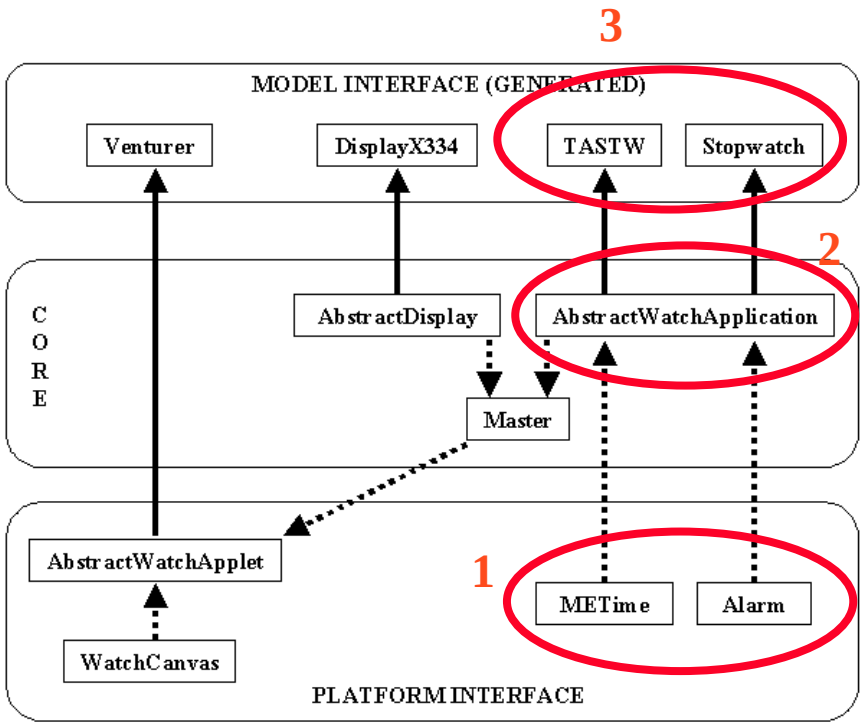
Example: watch framework [1/2]

- Domain framework provides
  - Expected code generation output interface
  - Basic building blocks in the form of abstract superclass 'templates'
  - Java classes to interface with the target platform
- Platform includes:
  - Hardware and OS
    - Windows
    - Linux
  - Run-Time Environment
    - Browser
    - MIDP
  - Java & RTE widgets

© 2003 MetaCase

Example: watch framework [2/2]

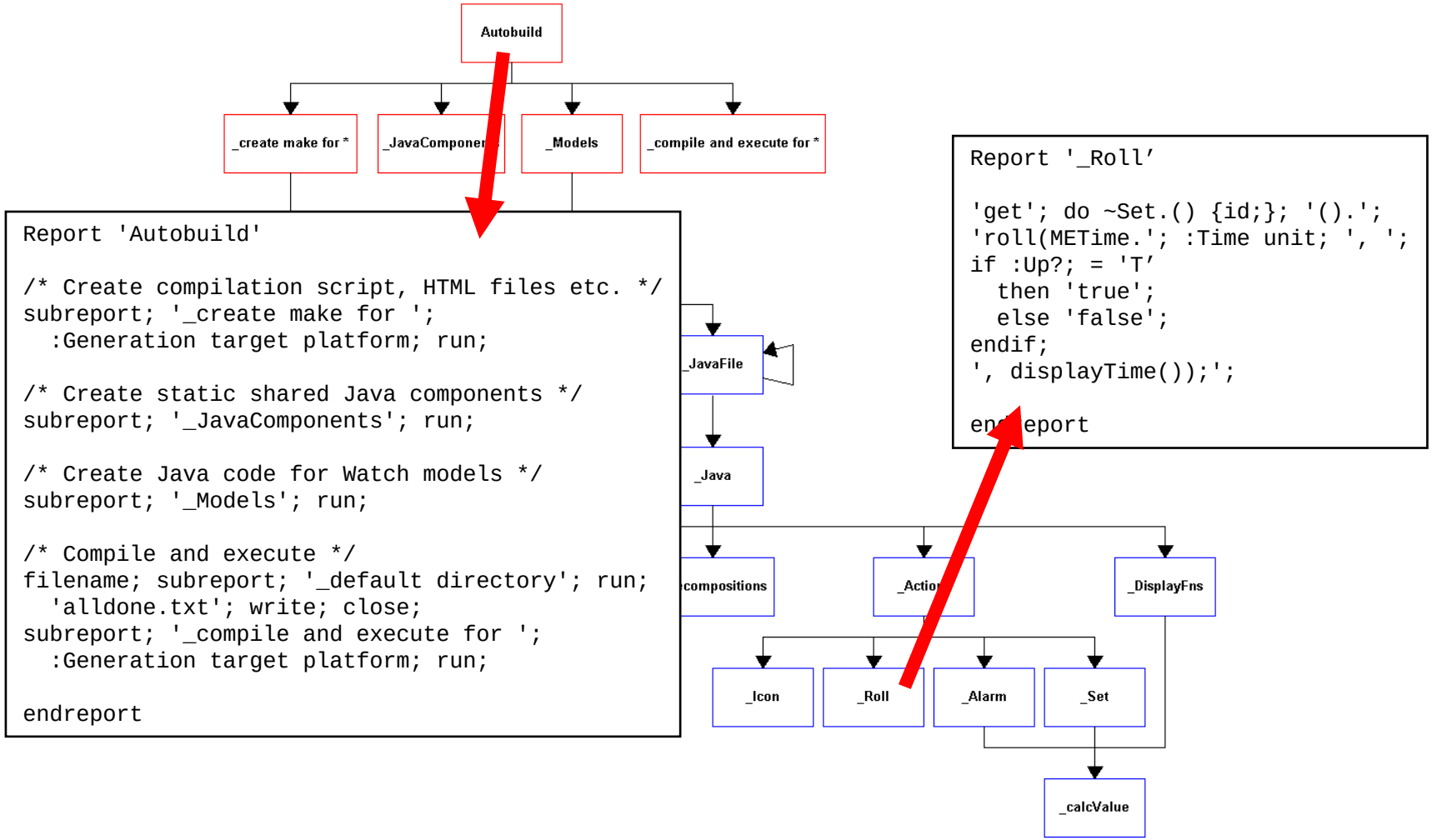
1. Platform interface: Implementing Time datatypes and Alarm services
2. Core: template implementation of the watch application as an abstract class
3. Expected generator output: Specification for creating and extending watch application by inheriting from abstract class



© 2003 MetaCase

Code generator structure

- Modular implementation to manage complexity

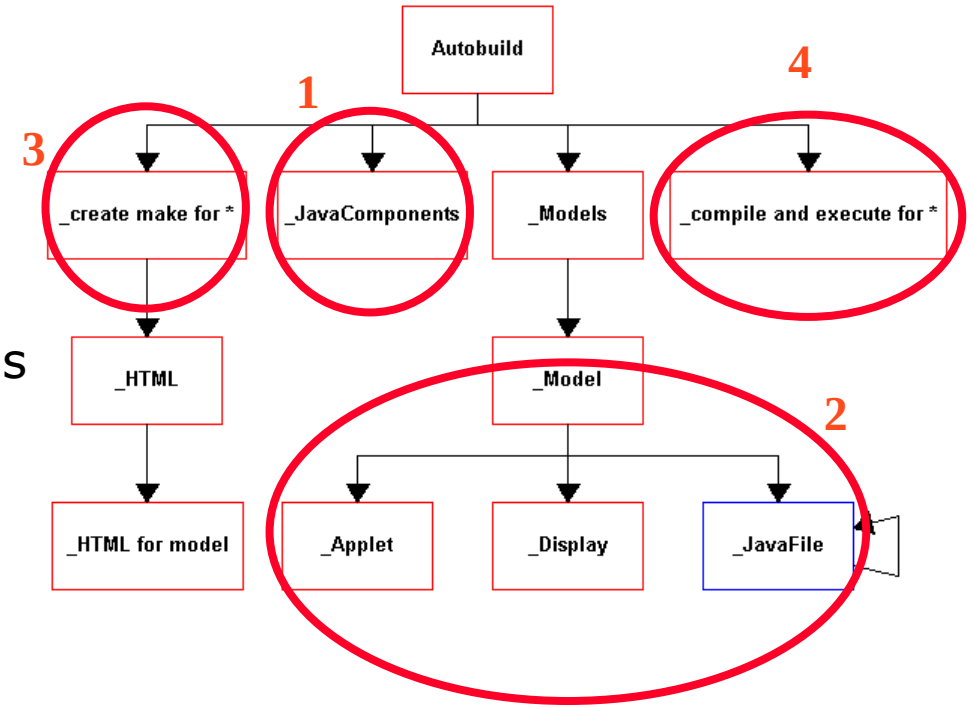


© 2003 MetaCase

Code generator example [1/3]

- General structure of generator for watch domain

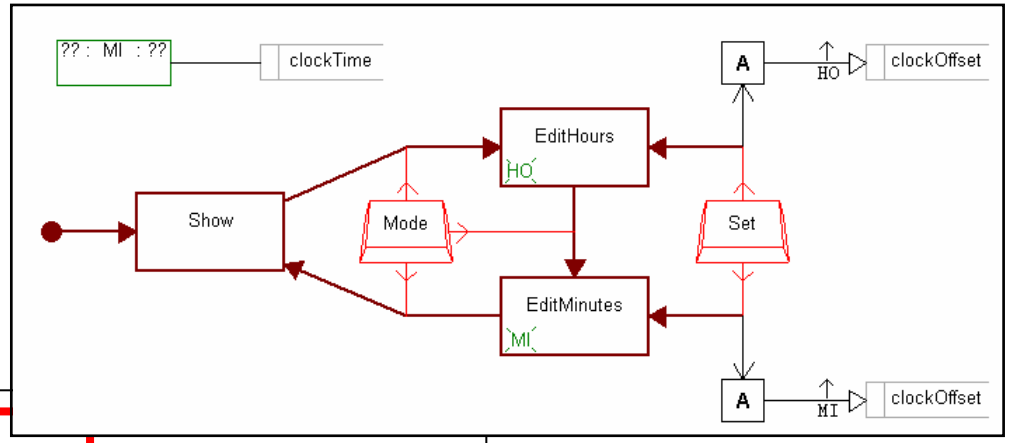
1. Generate framework components
2. Generate each watch model
   - UI
   - Display
   - Applications
3. Generate additional files for compiling and executing
4. Compile and execute



© 2003 MetaCase

Code generator example [2/3]

- Generating watch applications

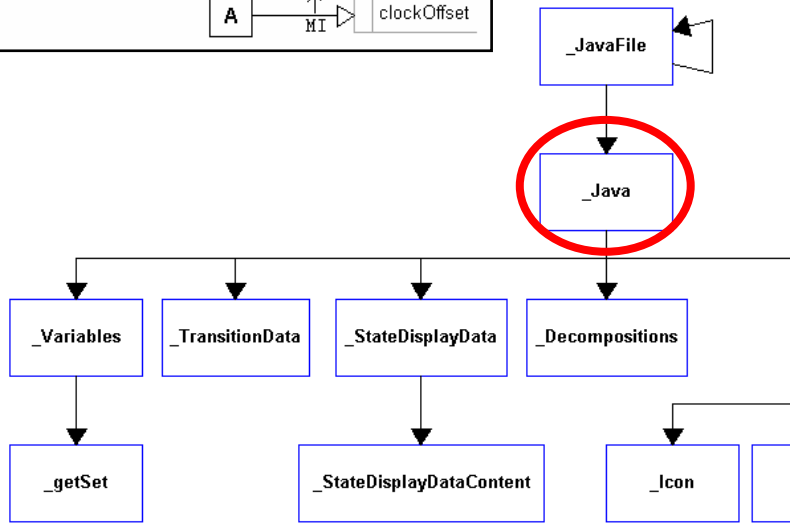


Generator ouput

```
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");

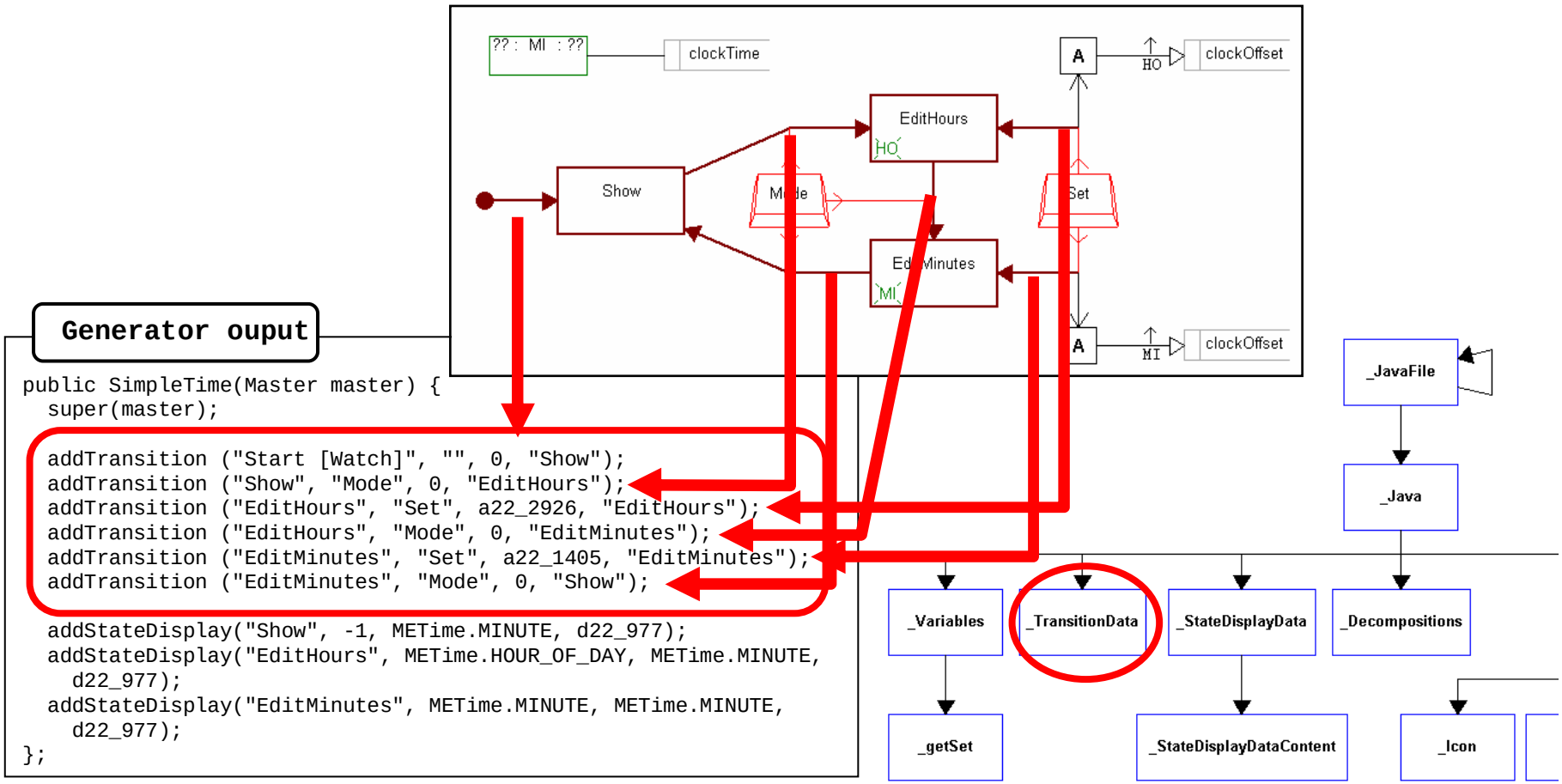
  addStateDisplay("Show", -1, METime.MINUTE, d22_977);
  addStateDisplay("EditHours", METime.HOUR_OF_DAY, METime.MINUTE,
    d22_977);
  addStateDisplay("EditMinutes", METime.MINUTE, METime.MINUTE,
    d22_977);
};
```



© 2003 MetaCase

Code generator example [2/3]

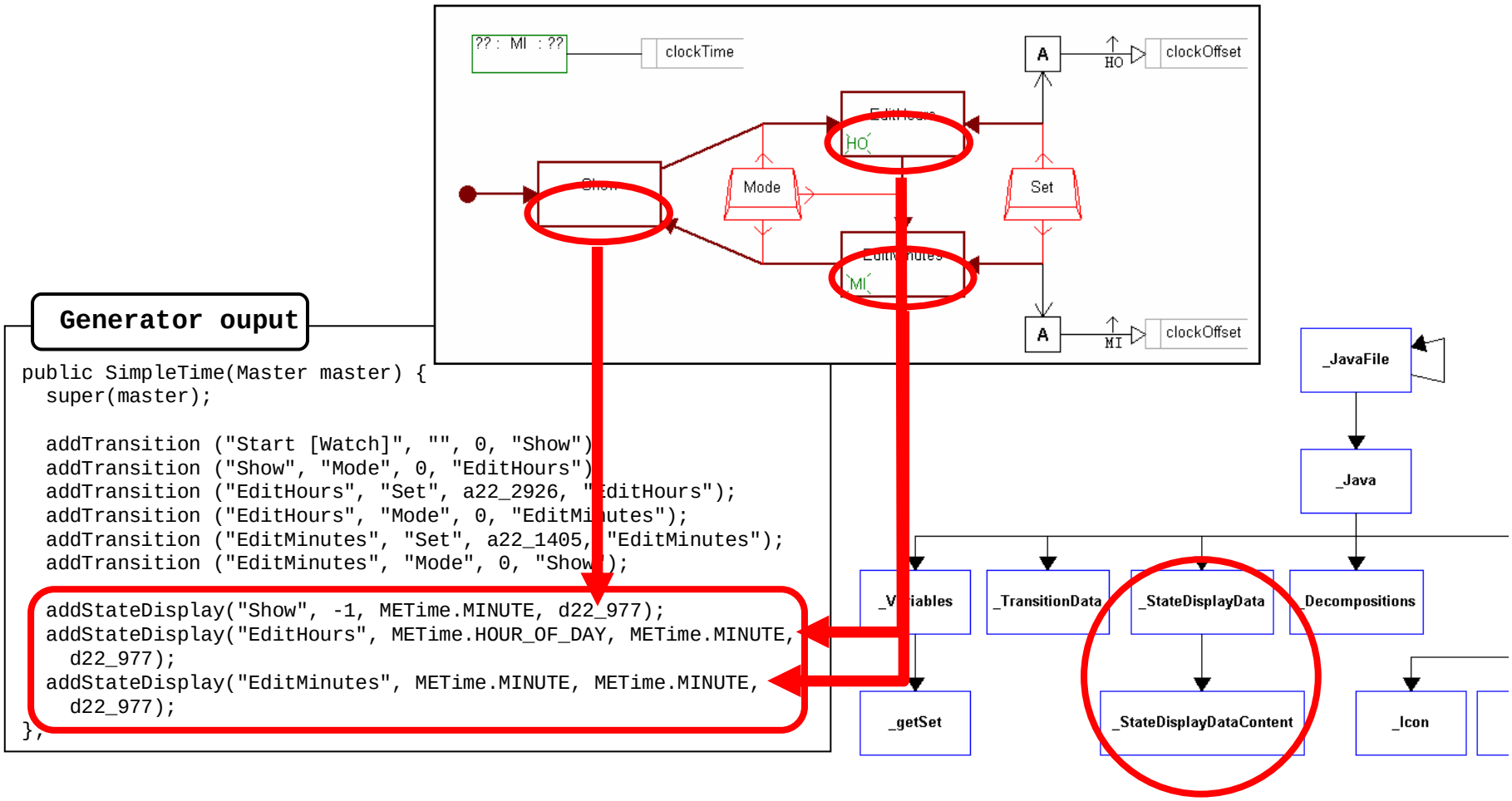
- Generating watch applications



© 2003 MetaCase

Code generator example [2/3]

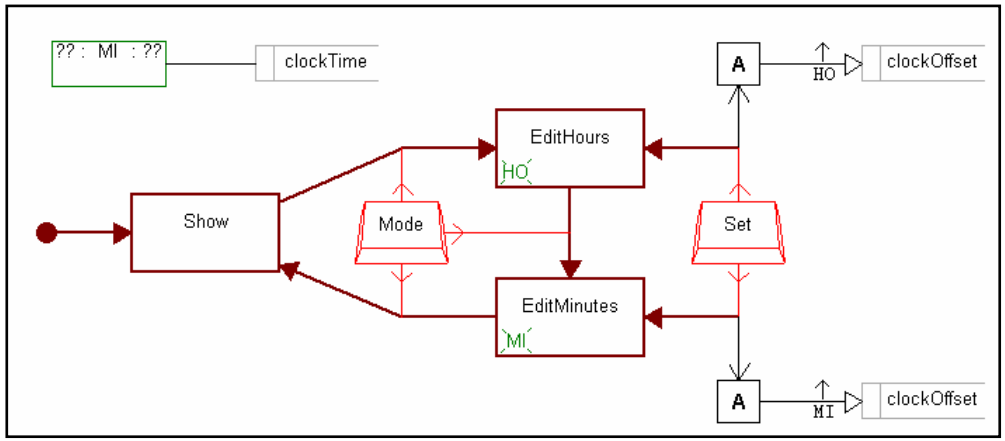
- Generating watch applications

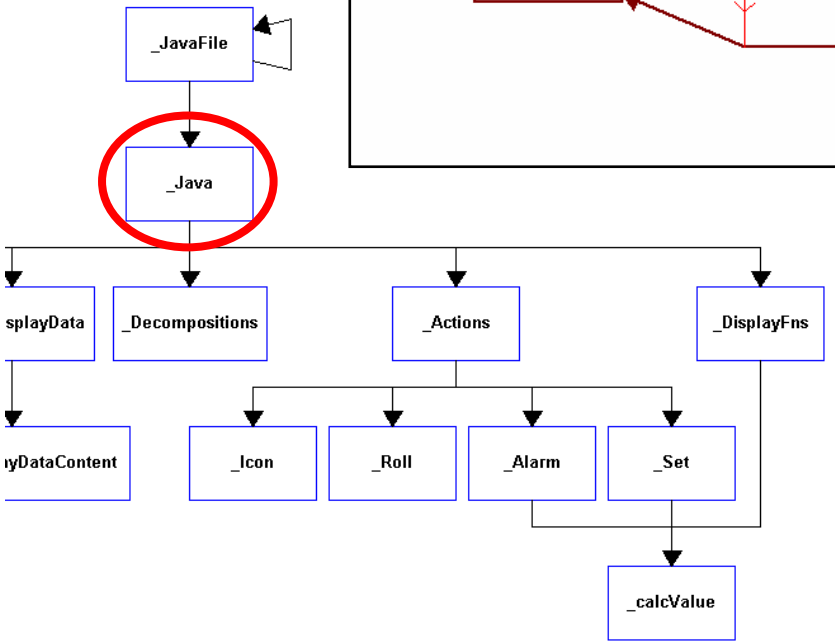


© 2003 MetaCase

Code generator example [3/3]

- Generating watch applications





Generator output

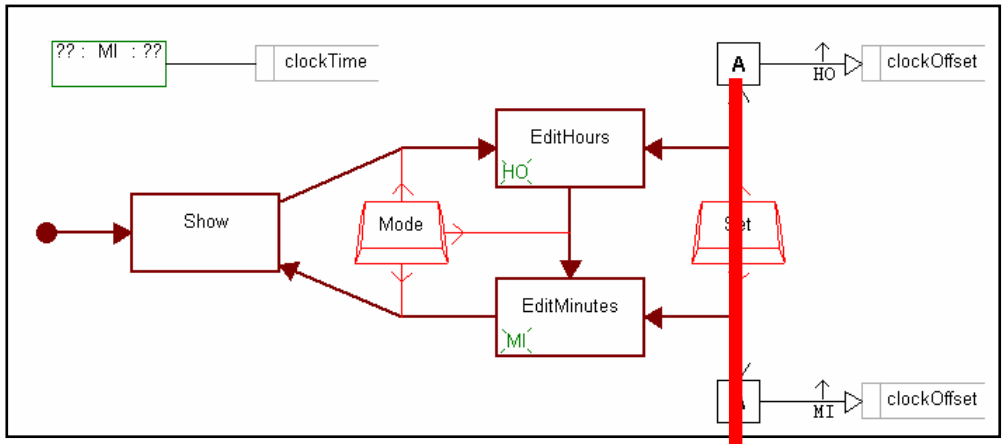
```
public Object perform(int methodId) {

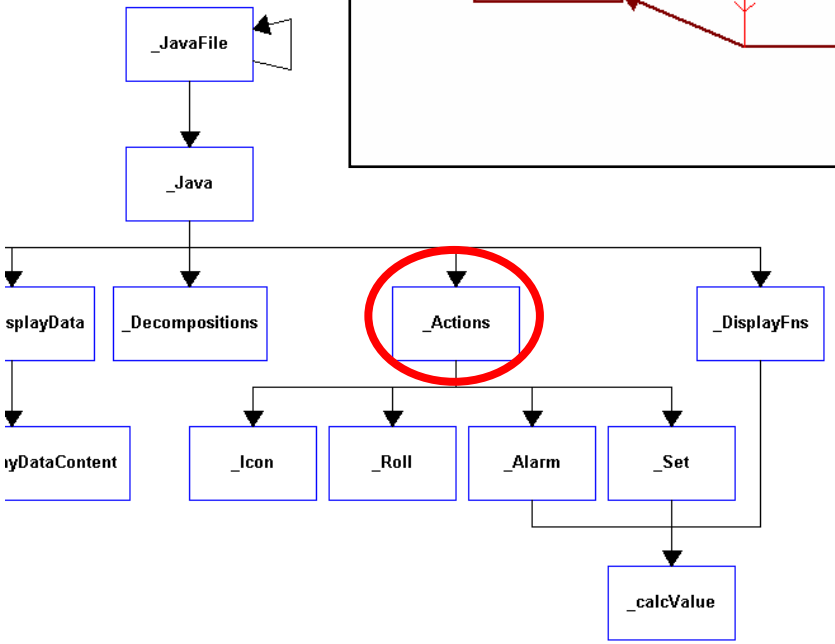
  switch (methodId) {
    case a22_2926:
      getclockOffset().roll(METime.HOUR_OF_DAY, true,
        displayTime());
      return null;
    case a22_1405:
      getclockOffset().roll(METime.MINUTE, true,
        displayTime());
      return null;
    case d22_977:
      return getclockTime();
  }
  return null;
}
```

© 2003 MetaCase

Code generator example [3/3]

- Generating watch applications





Generator output

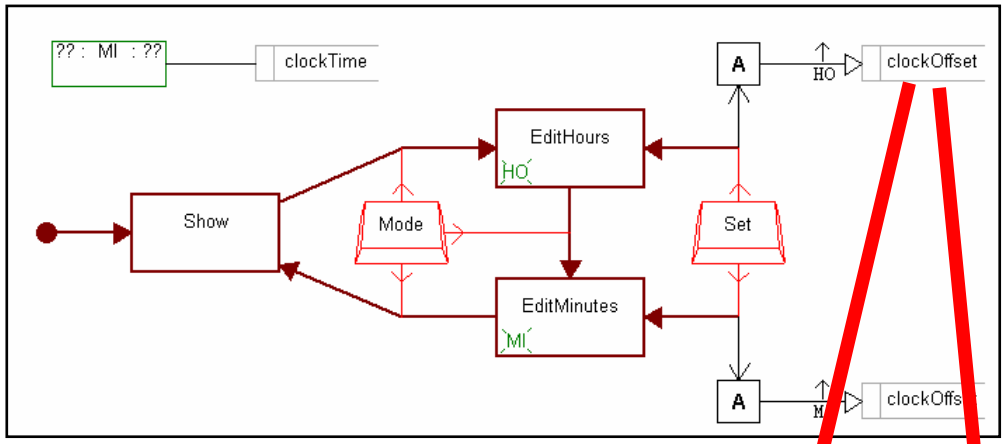
```
public Object perform(int methodId) {

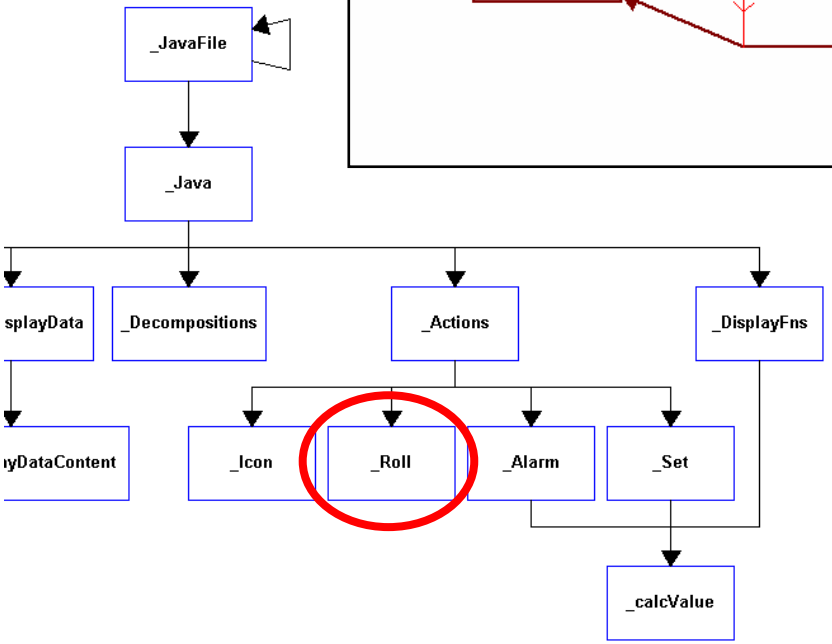
  switch (methodId) {
    case a22_2926:
      getclockOffset().roll(METime.HOUR_OF_DAY, true,
        displayTime());
      return null;
    case a22_1405:
      getclockOffset().roll(METime.MINUTE, true,
        displayTime());
      return null;
    case d22_977:
      return getclockTime();
  }
  return null;
}
```

© 2003 MetaCase

Code generator example [3/3]

- Generating watch applications





Generator output

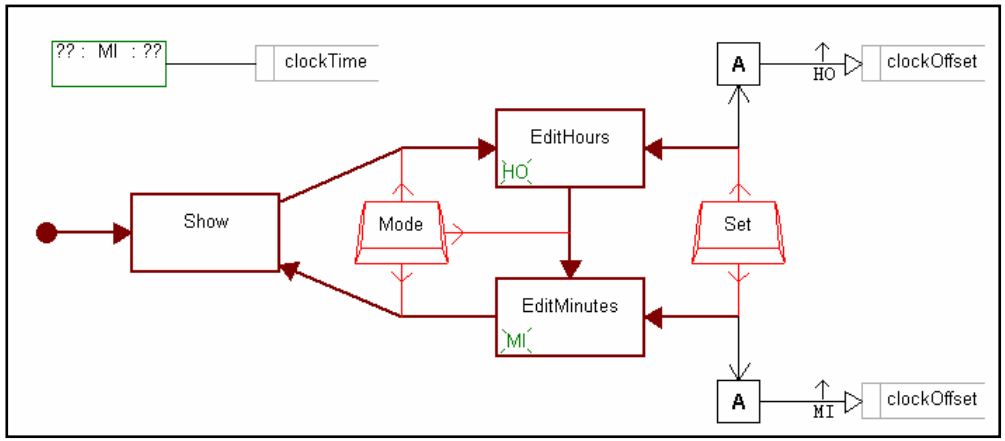
```
public Object perform(int methodId) {

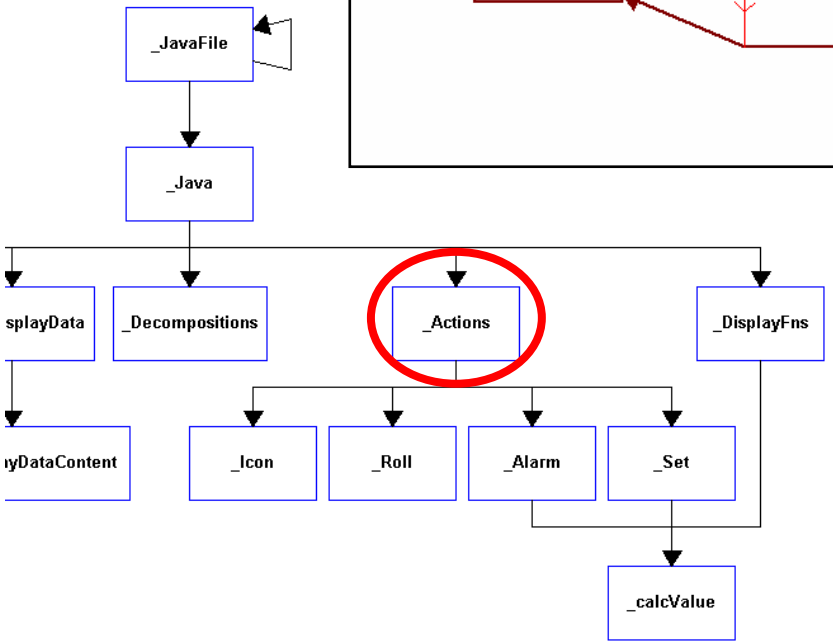
  switch (methodId) {
    case a22_2926:
      getclockOffset().roll(METime.HOUR_OF_DAY, true,
        displayTime());
      return null;
    case a22_1405:
      getclockOffset().roll(METime.MINUTE, true,
        displayTime());
      return null;
    case d22_977:
      return getclockTime();
  }
  return null;
}
```

© 2003 MetaCase

Code generator example [3/3]

- Generating watch applications





Generator output

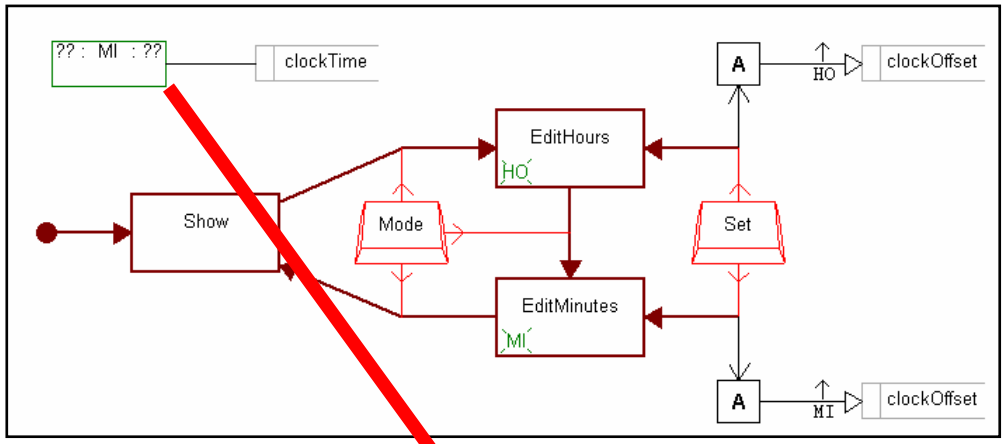
```
public Object perform(int methodId) {

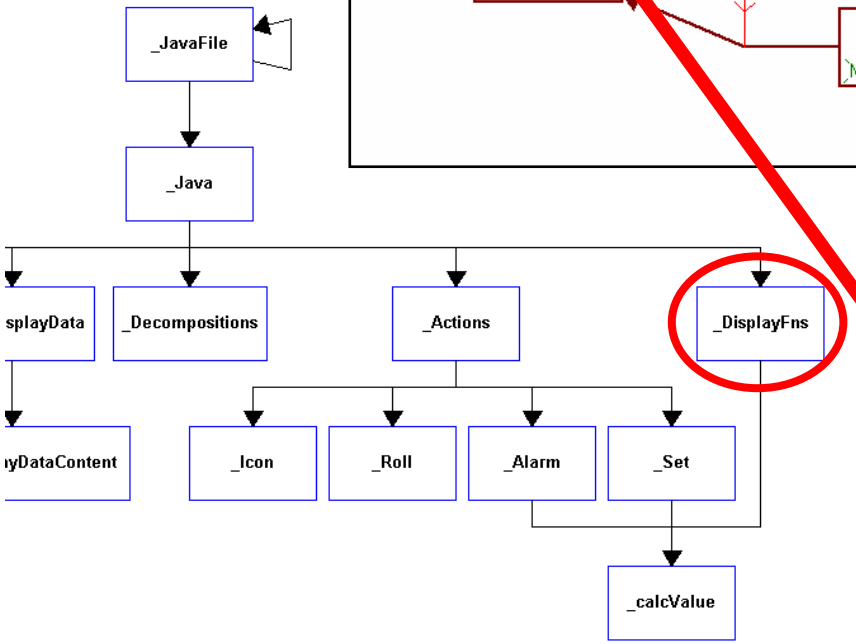
  switch (methodId) {
    case a22_2926:
      getclockOffset().roll(METime.HOUR_OF_DAY, true,
        displayTime());
      return null;
    case a22_1405:
      getclockOffset().roll(METime.MINUTE, true,
        displayTime());
      return null;
    case d22_977:
      return getclockTime();
  }
  return null;
}
```

© 2003 MetaCase

Code generator example [3/3]

- Generating watch applications





Generator output

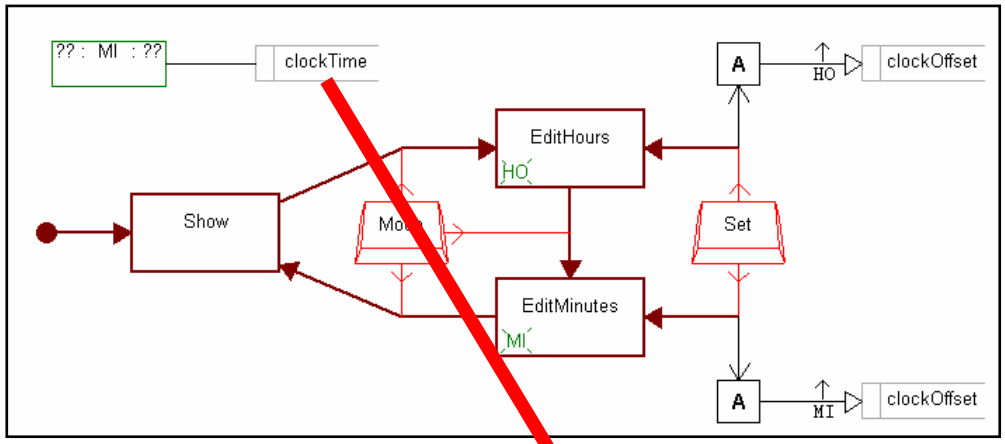
```
public Object perform(int methodId) {

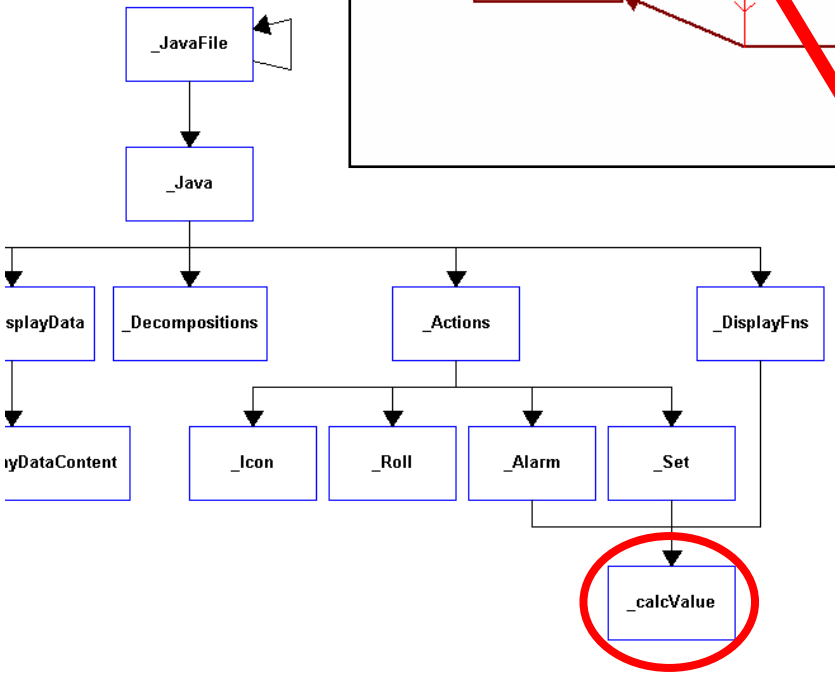
  switch (methodId) {
    case a22_2926:
      getclockOffset().roll(METime.HOUR_OF_DAY, true,
        displayTime());
      return null;
    case a22_1405:
      getclockOffset().roll(METime.MINUTE, true,
        displayTime());
      return null;
    case d22_977:
      return getclockTime();
  }
  return null;
}
```

© 2003 MetaCase

Code generator example [3/3]

- Generating watch applications





Generator output

```
public Object perform(int methodId) {

  switch (methodId) {
    case a22_2926:
      getclockOffset().roll(METime.HOUR_OF_DAY, true,
        displayTime());
      return null;
    case a22_1405:
      getclockOffset().roll(METime.MINUTE, true,
        displayTime());
      return null;
    case d22_974:
      return getclockTime();
  }
  return null;
}
```

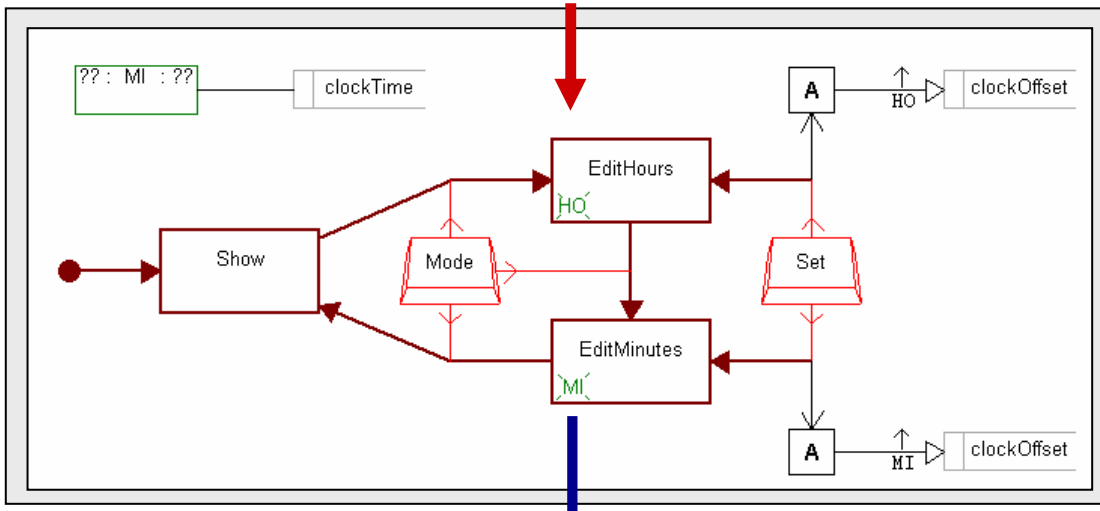
© 2003 MetaCase

All together: Watch DSM environment



PRODUCT IDEA

DSM TOOL



PRODUCT MODEL

Generated code

```
getclockOffset().roll(METime.MINUTE, true, displayTime());
...
```

PRODUCT CODE

+

Framework primitive

```
public void roll(int field, boolean up, METime displayTime)
{
   ...
}
```

+

DOMAIN FRAMEWORK

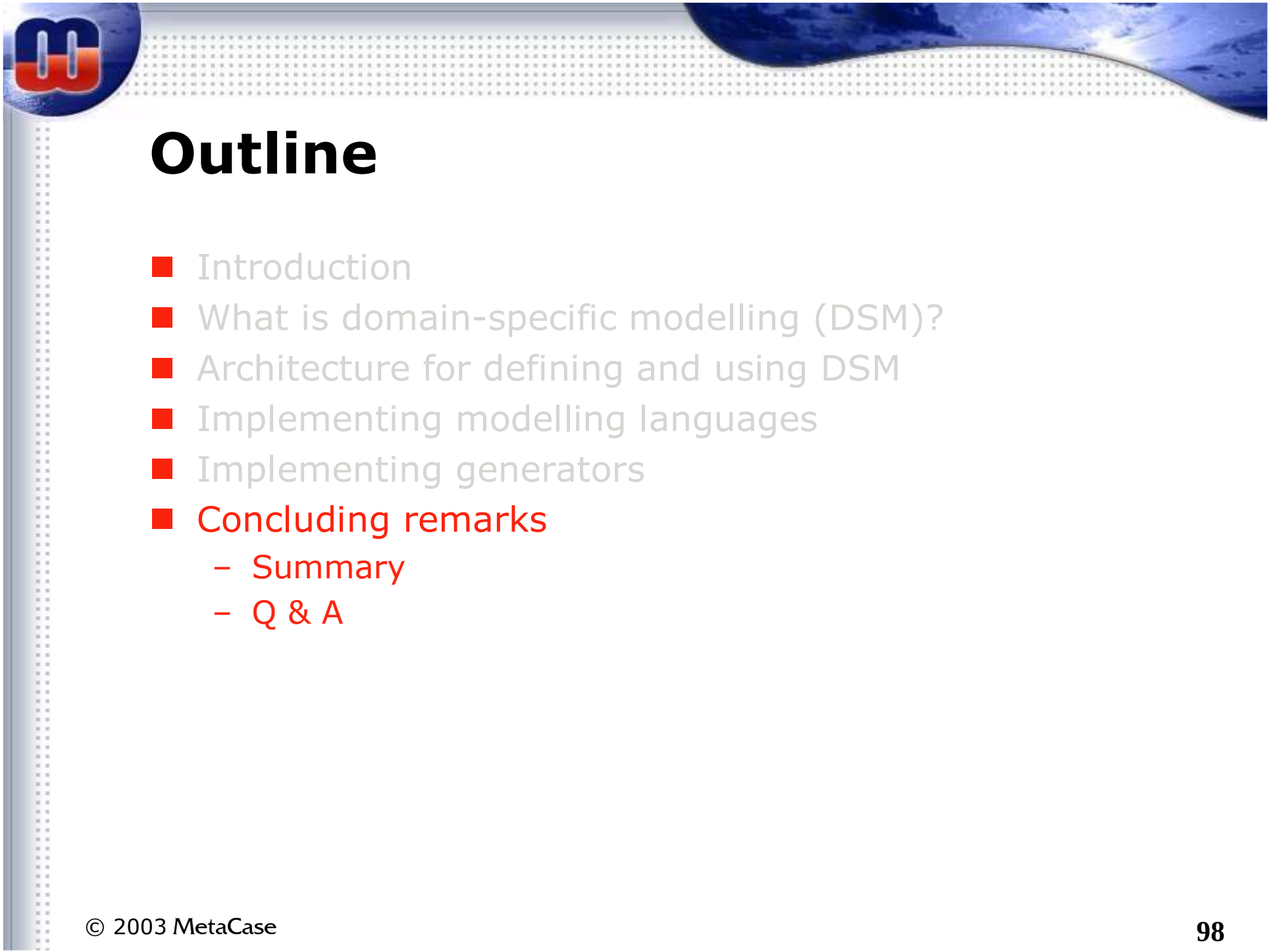
© 2003 MetaCase



Exercise: extending generators

- What if we would need heartbeat functionality
  - have external HW
  - HW abstraction layer with following services
    - start, stop
    - read
    - set monitoring interval

© 2003 MetaCase



Outline

- Introduction
- What is domain-specific modelling (DSM)?
- Architecture for defining and using DSM
- Implementing modelling languages
- Implementing generators
- Concluding remarks
  - Summary
  - Q & A

© 2003 MetaCase



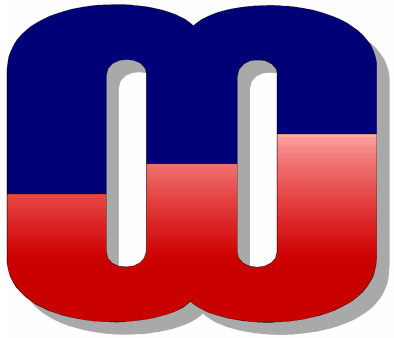
Summary

- Productivity can be improved by raising the abstraction beyond coding
- Generators can be applied effectively if both metamodel and generators can be customized
- Generator doesn't need to "do" everything
  - No single place to put automation rules
  - Use metamodel, apply platforms
- 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

© 2003 MetaCase

Thank you!

Question and comments?



MetaCase

Juha-Pekka Tolvanen, jpt@metacase.com
Ylistönmäentie 31
FIN - 40500 Jyväskylä, Finland
Phone +358 14 4451 400, Fax +358 14 4451 405
www.metacase.com

© 2003 MetaCase

References

- Batory, D., Chen, G., Robertson, E., Wang, T., (2000) Design Wizards and Visual Programming Environments for GenVoca Generators, IEEE Transactions on Software Engineering, Vol. 26, No. 5.
- 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.
- 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.
- MetaCase, Nokia case study, www.metacase.com
- Pohjonen, R., Kelly, S., Domain-Specific Modelling, Dr. Dobb's, 8, 2002
- Tolvanen, J.-P., Keeping it in the family, Application Development Advisor, July-August, 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.
- Tolvanen, J-.P., J. Gray, S. Kelly, K. Lyytinen, Procs of OOPSLA workshop on Domain-Specific Visual Languages, Tampa Bay, Florida, USA, University of Jyväskylä, Techical Reports, TR-26, Finland, 2001.
- Weiss, D., Lai, C. T. R., Software Product-line Engineering, Addison Wesley Longman, 1999.
- White, S., (1996) Software Architecture Design Domain, *Proceedings of* Second Integrated Design and Process Technology Conf., Austin, TX., Dec. 1-4, 1: 283-90.

© 2003 MetaCase



DSM related events

- Workshop on Domain-Specific Modeling (OOPSLA 2003)
- IEEE Symposium on Visual Languages and Formal Methods (VLFM '03)
- Engineering Methods to Support Information Systems Evolution' (EMSISE'03)
- 2nd Workshop on Domain-Specific Visual Languages, OOPSLA'02
- International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT '02)
- IEEE Symposia on Human Centric Computing Languages and Environments (HCC'02)
- Workshop on Domain Specific Visual Languages, OOPSLA'01
- IEEE Symposium on Human-Centric Computing Languages and Environments, HCC '01
- International Workshop on Model Engineering, ECOOP'00

© 2003 MetaCase