Domain-Specific Modeling, Model-Driven Architecture

14 February 2010

Juha-Pekka Tolvanen

MetaCase

Outline

- Why Model-Driven Development?
- What is Domain-Specific Modeling?
- What is Model Driven Architecture?
- Examples and experiences from the industry
- How to define own languages and generators
- Q&A
Why modeling?

- Complex
  - overwhelming amount of detail
  - many levels of change
  - different views
- Uncertain
  - why to change
  - what to change
  - how to change?
- Contextual and contingent
  - past history
  - development group
  - domain
  - technology

Modeling procedures

- Requirements
  - effectivity (effectiveness)
  - efficiency
  - completeness
  - consistency
  - accuracy
  - well-defined products
  - determinism
  - relevance
  - formalisability
  - communicable
  - reducing complexity
  - stepwise
  - integrated
How productivity has improved?

- "The entire history of software engineering is that of the rise in levels of abstraction"
- Newer programming languages have not increased productivity
- UML and visualization of code have not increased productivity
- Abstraction of development can be still raised when move from solution domain to problem domain
  - Inside one company, product family, business area etc.

How do we use models?

<table>
<thead>
<tr>
<th>Separate model &amp; code</th>
<th>CASE</th>
<th>Code visualization</th>
<th>Roundtrip</th>
<th>MDA®</th>
<th>DSM/DSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>Model</td>
<td>'Model'</td>
<td>'Model'</td>
<td>Model</td>
<td>Model</td>
</tr>
</tbody>
</table>

- Model alone should be sufficient in most cases
  - No need to look at code
Reverse engineering?

Types of models

- State models
- Data models
- Process models
- Object models
- Interaction models
- Flow models
- Use Case models
- Collaboration models
- Component models
- Deployment models
- etc.
Use Case models

Data models
Object models

Data flow models
State models

- Quantity checking (Check quality)
  - Move to quality control

- After ordering (Make an order)
  - Place order

- Delivering product
  - Send

Modeling domain vs. modeling code

Domain Idea
- Map to code, implement
- Solve problem in domain terms

Assembler
- Map to code, implement

Finished Product
- Code
  - Generate, Add bodies
  - Map to UML

Domain Framework
- Generates code
  - Map to UML

UML Model
- CIM
  - PIM
  - PSM
Pedagogical example: Let’s design a mobile app for event registration ...

How would you design and implement the application?

- Prototyping (to finalize or throw away)?
- Start with making test cases (Test-Driven Dev)?
- Make design models and then implement?
- Model and generate (all) code?
- ...or something else?
  - Agile modeling, Feature Driven Development, eXtreme Proramming, Domain-Driven Design...?
- The selection of the suitable approach depends on a number of contingencies!
  - There is no single best approach...
  - ...but several good principles: DRY, KISS, avoid copying,
    low coupling & high cohesion, modularization
Traditional way: some modeling and then coding

- **Step 1: User view**

  ![Diagram of conference system and user view]

  - User
  - Conference system
  - SMS sending
  - Register
  - Cancel registration
  - View program
  - View pricing
  - See program on the web

**Development with UML...**

- **Step 2: Describe static structure**

  ![Diagram of application and SMS sending]

  - Application
  - SMS sending
  - Form
  - Field
  - Parameter
  - Class

Can we apply here generators to produce the code?
Visualizing code

- Modeling anti-pattern from code generation point of view: 1 symbol => 1 keyword

```java
public class Alarm extends Thread {
    String name;
    boolean localTimeAwareness;
    String alarmState;
    int sleeptime;
    static final int dayMs;
    Master master;
    AbstractWatchApplication alarmApplication;
    volatile boolean isLive=false;
    public Alarm(String alarmName, boolean aware, public void run() public void stopAlarm()
```
Development with UML...

- Step 4: Logic
  - user navigation
  - behavior
  - exceptions
  - etc.

- In steps 5...N

Can we apply here model-to-model transformation?

Development with UML+code

And finally we start coding!

- Implement the functions, access to APIs, remember the exceptions, architectural rules, UI guidelines etc.
- and throw models away as they are not anymore in sync

```cpp
// First we create a new one to handle this message (in case our own was in use)
CMSVSelectionOrdering sort; // we want to include also the invisible entries
sort.SetActiveSelectionKind(KMsvEntry); // we want to handle just the invisible entries
CMsvSelection* entries = iSession->Entries(sort); // Take a handle to the entries...
// We schedule sending of the message...
CBaseMtm* smsMtm = iMtmReg->NewMtm(sort); // we want to handle also the invisible entries
smsMtm->SwitchCurrentEntry(); // we want to handle also the invisible entries
this->SendMessageL(smsMtm); // Send message...
// Release allocated memory
delete smsMtm;
```

© 2010 MetaCase
Domain-Specific Modeling solution

after running the generator...
**DSM: 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
  - Narrow down the design space
  - Focus on single range of products

- Uses generators to produce the 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.

**MDA: Model Driven Architecture**

- Hard to pin down: all things to all men
- Strong lock-in to OMG (standards: XMI, OCL, QVT ...)
  - Initially "you must use UML"
  - But later, in MDA manifesto, Booch et al. say: "The full value of MDA is only achieved when the modelling concepts map directly to domain concepts rather than computer technology concepts"
  - Now: "you can have any language you like, as long as it's like UML" – only allowed to build languages with MOF

- Schism into two schools of thought:
  - Elaborationist (OMG): Model a bit, transform, edit transformed models, generate, edit generated code
    - Computationally Independent Model, Platform Independent Model, Platform Specific Model
  - Translationist (XUML): Generate directly from high level UML-like models
Integrating models with M2M transformations?

- MDA "theory": PIM -> (PIM | PSM) -> (PSM | Code)

  E.g. add get, set

  Copying the same thing to multiple places is a bad thing

  // MovingObject is an abstract class
  public abstract class MovingObject
  { public static Point position = 100.0@100.0;
    public Point velocity = 0.0@2.0;
    public Point extent;
    public ColorValue colour
    // Displays the object at its current position. The default is
    public void display() {
      // MEPMD5 16_9042 protected block for method body
      // Enter code here
      // MEPMD5}
    // Move changes the location and direction of the moving object
    public void move(Point
      // MEPMD5 16_9088 protected block for method body
      // Enter code here
      // MEPMD5 e3d38755fd3f68647fbe5d3a33c6fd8f
    }

Extending current language: a case of (UML) profiles

- Principle: extend the current language
  - Rule example (OCL): A 'blow' can be associated with a direction only to one 'move' action
Extending current language: a case of (UML) profiles

- Large number of rules makes twisting the original syntax and semantics expensive and difficult
  - How about domain rules in OCL:
    - "After sending SMS only one UI element can be triggered"
    - "Forms can’t have cancel action", etc.
- With profiles we can’t take anything way
  - Do we want to build something bigger than UML?
- How to avoid using the unnecessary parts of the language?
  - In our example: e.g. properties of association, inheritance etc.
  - Define OCL for all?
    - "Don’t give names for associations" etc.
- Hard to raise the level of abstraction with UML concepts

MDA Pros & Cons

+ OMG: Some claim to vendor-independence (IBM?)
  - Standard is missing major areas
    - Based on UML, largest and most bug-ridden standard
  - Large number of other coupled standards
    - MOF, XMI, OCL, QVT – all moving targets, unproven
+ Focused on one domain anyway
  - Business apps with db and web or GUI front-end
  - Largely an accident: just didn't know other domains
+ Vendors will make something work
  - But you won’t be able to make your own language
- Productivity gains minimal
  - E.g. +30% in vendor-sponsored test
**DSM Pros & Cons**

+ Fundamental productivity and quality improvements
  - 300% faster in academic study, 1000% reported by companies
  - 50% less errors in academic study
+ Gives full control to the company
  - Experienced developers are sitting in the driver’s seat
  - Requires expertise and resources from the company
+ Minimal vendor lock
  - Metamodel-driven tools are open
  - You can translate & transform models to other tools and formats
+ Only few industry strength tools available
  - Scalability to a larger number of developers
  - Do not handle evolution and maintenance

**Case:**

**Business Process Modeling for XPDL**

- Defining business processes to be executed in a workflow engine
- Modeling language for 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
Case: Insurance products & eCommerce

- Developing portal for insurances and financial products
- Need to specify several hundred financial products
- Insurance experts visually specify insurance products and generate code to the portal

- Comparison to hand-writing Java after first 30 products = DSM at least 3 times faster, fewer errors
Case: Web application

- Web application for e-commerce:
  - catalogs, events, press releases, 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
Intershop example

- Visual Pipeline Manager
- Language to describe business flows using pipelines
- Separate aspects from business logic and presentation
  - Tasks
  - Flows
  - Interaction
  - Stop, error, calls, jumps
  - Decision, join, loop

Hänsgen, Model-Driven Software Development in Practice, MDD&PL, 2006
Economics of DSM

- **Gains:**
  - Productivity increase compared to earlier practice
  - Quality improvements compared to earlier practice
  - The number of expected users or implementations

- **Effort:**
  - Create and maintain the language and related code generators
  - Obtain and maintain tool support for the language
  - Learning the (formalized) language
    - Note: domain experts should already be familiar with the concepts!
Productivity increase with DSM

Comparing to earlier practice (typically compared to coding)

Percent Increase

0 % 100 % 200 % 300 % 400 % 500 % 600 % 700 % 800 % 900 % 1000 %

Embedded UI applications 1000 % Mobile phone software 750 % Phone switch features 600 % Call processing services 500 % Heart rate monitor 500 % J2EE web application 600 % Home automation 600 %

Domains

How to implement automation...

Done a few times before!

Domain Idea

Expert (few)

DSM language

Code generator

Framework code

Finished Product

Normal (many)

Easy!

Model in DSM language

Generate code

Domain Framework
Implementing modeling languages

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

Metamodelling example: part of class diagram

- Metamodel specifies language concepts and related rules
- Metamodel is instantiated when creating models
- Model can’t express other aspects that those defined in the metamodel
- Model can be instantiated once, metamodel twice!
Identifying DSM constructs [1/2]

- 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 [2/2]

- 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
Rules in the languages

- The domain concepts of a modeling language are bound together with rules
- Putting the rules into the language allows
  - preventing creation of illegal models
  - informing about missing data
  - keeping models consistent
  - make code generation possible
- Prefer having rules as part of metamodel to having separate checker
  - Support early error prevention and provide guidance
  - But going overboard can hinder flow of modeler

Defining notation

- Vital for acceptance and usability
- Symbols can vary from boxes to photorealism
  - Best to resemble closely the actual domain representation
  - Worst is having everything a box and special text to show the difference (cf. stereotypes)
  - Design information needs space: compromise
- Don’t create notation from scratch
  - Use known/existing elements (and, or, start, stop etc)
- Hint: ask users to define the notation
  - It is much easier to introduce their own language than something you created
  - Remember also model readers
    - managers, test engineers, customers, deployment, configuration, packaging and even sales
Generator

- Generator translates the models into the required output
  1. crawls through the models → navigation according to metamodel
  2. extracts required information → access data in models
  3. translates it into the code → translation semantics and rules
  4. using some output format → possibility to define output format

Model navigation and translation

- Multiple ways to navigate
  - Using some start elements
  - Based on types
    - Object types
    - Relationship types
    - Objects with certain connections
    - Objects with certain submodels
    - Relationships with submodels, etc.
  - Based on certain instance values
    - In any model element, or set of them

- Different computational implementations possible
  - Sequential
  - Function calls
  - Switch-case structure
  - Transition tables etc.
Function calls – Series 60/Python

```python
def Note3_2227():
    appuifw.note(u"Registration made", 'conf')
    return Stop3_983

def Note3_6109():
    appuifw.note(u"Registration cancelled", 'info')
    return Stop3_983

def Note3_2543():
    appuifw.note(u"Conference registration: Welcome", 'info')
    return Popup_menu3_2520

def Stop3_983():
    # This applications stops here
    return appuifw.app.set_exit

...```

Combining generated code and other code

- Different levels of code to integrate
  - Other generated code
  - Domain framework, components, platform functions
  - Hand-made code
- Separation of generated and non-generated code
  - Best to keep them in separate files (or sections in files)
- Generated code...
  - can call existing code, instantiate data structures
  - can be called from existing code
  - can be subclassed from existing code
  - can form base classes

Model

Generated code

Non-generated
How to design a generator [1/2]

- Make generator for your situation only
  - Trying to make general purpose generator often fails
- Make generation process complete, target 100% output
  - Never modify the generated code
  - Do you want to edit Assembler after compiling?
    - Correct the generator or framework instead
  - No round-trip-related problems
    - Do you want to edit Assembler and keep C in synch with it?
- Use modeling languages to raise abstraction
  - Don't visualize code
  - Generating a class from a diagram to a class in code helps very little, if at all...
- Put domain rules up-front to the language
  - Generator definition becomes easier when the input is correct
  - Models should be impossible to draw wrongly for generation
How to design a generator [2/2]

- Try to generate as little code as possible
  - Glue code only, rest in domain framework or platform

- Keep generator as simple as possible
  - Raise variation to the specification language
  - Push low-level common implementation issues to the framework

- Keep generator modular to reflect changes
  - e.g. structure generator based on modeling languages, generated files, modeling concepts
  - e.g. use common generator subroutines for common needs

- Make generated code readable ("good looking")
  - To be used later while debugging the code, executing it in a simulator, and while implementing the generator
  - Follow good coding standards, include comments, have data to link back to models (e.g. in comment or via e.g. simulator)

Thank you!