Model-Driven Architecture
vs.
Domain-Specific Modeling

14 Feb 2008

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
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?

- Model alone should be sufficient in most cases
  - No need to look at code
Modeling domain vs. modeling code

Domain Idea
- Solve problem in domain terms
- Map to code, implement
- Map to UML
- No need to map!

Code
- Generates code
- Generates domain model

Product
- Assembler
- Code
- CIM
- PIM
- PSM

Let’s inspect an example

Conference registration
- Early pm: 995, - USD
- So 10:45: T1: Smith
  - T2: Jackson
  - T3: Holland
- So 11:30: T4: Cohen
  - T7: Dawson

Ok  Cancel
Traditional way: some modeling and then coding

- **Step 1**: User view

  - Conference system
  - SMS sending
  - Register
  - Cancel registration
  - View program
  - View pricing
  - Receive registrations
  - See program in

Development with UML...

- **Step 2**: Describe static structure

  - Can we apply here model-to-model transformation?
Step 3: Specify interaction

Step 4: Logic
+ user navigation
+ behavior
+ exceptions
+ etc.
+ In steps 5...N

Can we apply here model-to-model transformation?
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

```python
... def Query25_931():
# Query: Your name?
    global PersonNamed
    PersonNamed = appuifw.query(u"Your name?", 'text')
    if PersonNamed:
        return (List25_275, True)
    else: # Cancel selected
        return ((call_stack.pop()), False)

def SendSMS25_692():
# Sending SMS Cancel_registration
    # Use of global variables
    string = u"Cancel_registration 
    appuifw.note(string, 'info')
    messaging.sms_send('+358400648606', string)
    return (Note25_649, False)
...```

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

**How is DSM different from MDA?**

- Same idea on using models and transformations, but...
- DSM is always full code direct from models
  - Not OMG MDA (elaborationist)
  - Simpler in terms of versioning and management
- DSM = domain-specific language and generators
  - MDA is UML-based* (or MOF based)
- No reverse- or round-trip engineering in DSM
  - We want a real lift in the level of abstraction
  - How often do you reverse engineer assembler to code?
- Separation of concerns
  - You are the experts in your domain and code (not the vendor)
- DSM is agile: as much or as little as you want
  * official definition, www.omg.org
Works in any domain
(not on phone only)

<table>
<thead>
<tr>
<th>Problem domain</th>
<th>Solution domain/ generation target</th>
</tr>
</thead>
<tbody>
<tr>
<td>Telecom services</td>
<td>Configuration scripts</td>
</tr>
<tr>
<td>Insurance products</td>
<td>J2EE</td>
</tr>
<tr>
<td>Business processes</td>
<td>Rule engine language</td>
</tr>
<tr>
<td>Industrial automation</td>
<td>3 GL</td>
</tr>
<tr>
<td>Platform installation</td>
<td>XML</td>
</tr>
<tr>
<td>Medical-device configuration</td>
<td>XML</td>
</tr>
<tr>
<td>Machine control</td>
<td>3 GL</td>
</tr>
<tr>
<td>Call processing</td>
<td>CPL</td>
</tr>
<tr>
<td>Geographic Information System</td>
<td>3 GL, propriety rule language, data structures</td>
</tr>
<tr>
<td>SIM card profiles</td>
<td>Configuration scripts and parameters</td>
</tr>
<tr>
<td>Phone switch services</td>
<td>CPL, Voice XML, 3 GL</td>
</tr>
<tr>
<td>eCommerce marketplaces</td>
<td>J2EE, XML</td>
</tr>
<tr>
<td>SIM card applications</td>
<td>3 GL</td>
</tr>
<tr>
<td>Applications in microcontroller</td>
<td>8-bit assembler</td>
</tr>
<tr>
<td>Household appliance features</td>
<td>3 GL</td>
</tr>
<tr>
<td>Automotive infotainment</td>
<td>Java, C</td>
</tr>
<tr>
<td>ERP configuration</td>
<td>3 GL</td>
</tr>
<tr>
<td>Handheld device applications</td>
<td>3 GL</td>
</tr>
<tr>
<td>Phone UI applications</td>
<td>C++ (C, Python, Java)</td>
</tr>
<tr>
<td>SIP services</td>
<td>XML</td>
</tr>
</tbody>
</table>

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
Case: IMS Service Creation*

- Rapid creation, deployment and provisioning of IP-based services
- Modeling language centralizes service concepts
- Generate all required artifacts from a single design
  - Code, configuration, documentation
- Uses a service enabling framework
  - runs on top of off-the-shelf application servers
  - industry standard SIP-servlet (JSR 116)
- Services can be created easier and faster because of the higher abstraction level
  - without the usual cross-cutting concerns seen in SIP and HTTP servlet development.

* Implemented by ICT Automation

A basic sample. More complex uses include region and time based routing for e.g. Helpdesk applications.
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

- Repetition:
  - # of product variants
  - # of similar features
  - # of developers
  - “outsourcing” to domain experts

How to implement automation...

- Domain Idea
  - Expert (few)
  - Model in DSM language
  - Code generator
  - Framework code

- Normal (many)
  - Done a few times before!
  - Generate code
  - Domain Framework

Finished Product
The four levels

Envisaged application + Problem Domain + Solution Domain

Model and application level

User's world

Running application
The metalevel

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
Metamodeling 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

Generator definition

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

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

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

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

Generator output

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

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

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

def main():
    # 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

Manual coding vs Wizards vs DSM
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 sync 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!

info@metacase.com
www.metacase.com

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

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

© 2008 JPT/MetaCase