On the Evolution of Tools and Languages

Dr. Axel Uhl
Chief Development Architect
SAP, Office of the CTO
Language History
“Moore’s Law” of Platform Complexity

Platform complexity “doubles” every few years. Languages and Paradigms hardly keep up.

- 50’s: hard-wired
- 60’s: assembly
- 70’s: 3GLs, 4GLs, OO
- 80’s: “Modeling,” DSLs
- 90’s:
- 2000’s:
Benefits of Adequate Abstractions

Enabling / cost reduction for
- architecture / platform evolution
- optimization across layers

stack of platform components and languages (examples only)

complete, deployed system

migration effort with specification provided in platform-specific ways

replace platform components 2&1

migration effort with specification provided at abstract, portable levels

amount of specification content

complete specification

sketches

Business Model
Technical Model
Code using frameworks
Framework implementation
**Improved Development Efficiency**

**Take path of least effort**
- Detailing at too low an abstraction level causes extra effort and errors.
- Example: write an object-oriented business application in assembler
Approaches to Adequate Abstractions

“We could embed a DSL into a suitable host language.”
- Works for textual and graphical languages (e.g., Ruby, UML)
- Are tooling concerns addressed appropriately?
- How do you restrict the host language infrastructure to use only your DSL?

“Let’s build a new scripting language, and we’ll be doing fine.”
- But what distinguishes scripting in the first place?

“Ok, so we’re going to use a model-driven approach.”
- But what’s the difference between an executable model and a piece of code?
- And where is a graphical syntax more appropriate than a textual one?
- Is a model transformation or a code generator different from a compiler?
- Can I debug my models when the system doesn’t behave as expected?
- Will I remain agile, with powerful refactoring support?

Let’s take a closer look...
Learning from the “Scripting Trend”

Scripting is about
- eliminating the compilation step, leading to quick round trips
- ease of learning
- ease of change
- good integration capabilities
- using flexible type systems to make developer more productive

Blurring boundaries
- short compile/run/debug cycles even for compiled languages
- JIT compilation (Java byte code → native; JSP to Java to byte code; ...)
- type system qualities (static vs. dynamic vs. duck typing; inference)
- memory management and bounds checking in compiled languages
- lifecycle management requirements for scripting solutions
- difficult auto-completion and refactorings in IDEs due to lack of type information
There are many commonalities in what we call programming language and modeling language. Both

- have abstract and concrete syntax
- can be of rather declarative or imperative nature
- can use different types of representation
  (though we usually think of programming language artifacts as text strings)
- strive for adequate abstractions, concern separation and aspect localization

Many issues of classical “programming” also exist for “modeling”

- physical partitioning of artifacts
- dependencies
- teamwork aspects (change management, versioning, ...)

What’s the difference between

- a code generator / model transformer and a compiler?
- a piece of C++ code and a sequence chart?
What’s “Modeling?”

Herbert Stachowiak, *Allgemeine Modelltheorie*

- homomorphic representation
  - A model represents some *thing*.
  - Model and *thing* are connected by a morphism.

- abstraction
  - The model suppresses irrelevant detail and focuses on important aspects.

- pragmatics
  - The model is created for a purpose.

Let’s take a look at some examples...
Model Example: London Tube Map
Model Example: Crash Test Dummy
*Misunderstood Pragmatics...*
Homomorphisms at Work
Model Example: Mannequin
...because *this* would be impractical...
Partial Views
Model Example: Scale Models
...and the Wiring Plan as another Partial View
Model Example: Computer Programs

Pragmatics
- easily and correctly describe (parts of) a program that a machine can execute

Abstraction
- omit unnecessary technical detail of underlying platform

Homomorphism
- language semantics preserved across compilation or during interpretation

Examples
- UML model
- BPMN model
- program in some 3GL
- program in an assembly language
Informal Definition of “Modeling”

Intuitively, we can characterize models as
- supporting multiple partial views
- that are connected and kept in sync
- and which can be mapped, refined and woven into an executable system

The way we tend to design modeling languages and their tools...
- with an integrating, underlying model repository
- with a separation of concrete and abstract syntax
- with separate generated and sometimes manually refined artifacts
- where all have their own life cycle

...several challenges emerge for model-driven development
Lifecycle Issues of Multiple (Graphical) Views

Changing a model through one view may update another
- views may be versioned and access-controlled artifacts
- extensions to models may be provided in multiple layers of the system

Changes in graphical views may be for viewing only...
- toggle expanded/collapsed setting on a diagram entity
- change the zoom level and panning position

...but should not necessitate checkout/versioning operation
- user may not have the permissions required
- creation of a new version not justified by minor changes of settings

Logically atomic changes may affect multiple conflict/merge units
- cross-partition link addition/removal that is stored on both ends
- delete propagation along composition hierarchy
- needs to be considered during merge operations

How to display differences and merge conflicts of abstract model in concrete syntax (and in which)?
Understood for Text Syntaxes
Where to Display Conflicts for Partial Views?
Textual Syntaxes for Models

Overrated graphical notations

- difficult to implement and manage
- much semantics in “secondary notation” (grouping, ordering, alignment)
- turns out to work best “from expert to expert”

We see approaches promoting textual concrete model syntax

- with a model repository in the back
- “HUTN revived”

Leads to a different set of challenges

- managing “textual layout” (whitespace etc.)
- storing and editing unparsable text
- incrementally parsing while maintaining model element identity

Diff/merge remains difficult for cross-view changes
Other Challenges

Refactoring support
- Where is a decent UML tool with good refactoring for integrated OCL?
- Where is a decent model transformation tool that helps me refactor my hand-written code after refactoring the model?

Model migration after language evolution / metamodel change
- for some reason, language evolution seems to be perceived more necessary for “modeling” languages than for others (“language agility”)
- complicated by the dependency between abstract and concrete syntax which may vary independently

Still largely no industrial-strength “model component ware”
- the best we see is the open source development around and on top of EMF
- MOF fragmentation hasn’t helped
- standard workbench paradigms don’t support modeling very well yet
The “Babylonian” Problem

Software factories, language workbenches and all...
- make it easier to create and evolve certain aspects of languages,
- but we move away from language standards, using only standard meta-languages

A new set of languages for each project?
- often lacking good specification, documentation, community and support

What about changing teams, maintenance, specification longevity?
Tools & Languages: Factors to Keep in Mind

Number of developers working on a software product
- may require good collaborative features (e.g., partitioning, diff/merge)

Size of the software
- may require good modularization, as well as flexible delivery and deployment options

Supporting tool features enabled by the language design
- auto-completion ("intellisense"), refactoring, static checks, type inference
- debugging capabilities

Don’t forget about model debugging

Learn from the "Scripting Trend"

Life cycle of a piece of software
- from small to large
- from local to central
- from prototype to product
- from one-off to commercial success
The Good News

Language workbenches are gradually “getting there”
- language design and tool building is getting easier and easier
- workbench environments (e.g., Eclipse) work towards improved modeling support
- generic capabilities go hand in hand with the trend towards smaller DSLs

Abstract syntax specification converges on MOF+OCL
- even though EMF/EMOF/CMOF confuses, fundamentally it's about MOF
- OCL is ubiquitous, with improving infrastructure
  - editing
  - incremental evaluation
  - compiling into an OCL model for better refactoring

Decreasing hype opens our eyes for the essentials
- finally, we can start working on the core issues
Conclusions

*Modeling* is not too different from *coding*.

Increasing complexity of UML standard lets small DSLs thrive.

Relevance of graphical notations is overrated.

Generic qualities of “language workbenches” are increasingly relevant.

Language “agility” reduces attention to important quality attributes

- artifact longevity (migration)
- source/model-level debugging
- refactoring support
- team development support

By and large we’re moving in the right direction. Let’s keep going!
Thank You