Keep it Sweet & Simple

The Third KISS Workshop
25 October 2009 @ OOPSLA
Orlando FL, USA
08:30 - 08:45
Introduction round

08:45 - 09:00
Summary of KISS results to date, Jorn Bettin

09:00 - 09:15
Model-based interoperability, Tony Clark

09:30 - 12:00
Presentation of position papers, authors

13:00 - 16:40
Articulating the rationale of principles & guidelines in terms of examples

16:40 - 17:00
Conclusion & logistics of next steps
Building software intensive systems has taught us that domain specific languages represent the key to capturing, preserving, and exploiting knowledge in virtually all industries.

When designing and using domain specific languages we keep it sweet & simple (KISS). Most importantly, we are committed to the following values:

- We strive to **automate** software construction from domain models
- We work with **domain-specific assets**
- We support the emergence of **supply chains** for software services
- We see **Open standards, as well as Open Source** as driving interoperability
- We use methodologies that conform with the values of the **Agile Manifesto**

*Peter Bell, Jorn Bettin, Tony Clark, Keith Duddy, Scott Finnie, Matthew Fowler, Steven Kelly, Jack Kennedy, Frank Sauer, Laurence Tratt, Markus Voelter, Jos Warmer*
Hype Driven Software Developers

The challenges

• Those who have been disillusioned by UML and those who are not yet disillusioned by UML

• Those who have discovered the power of model driven generation in the last five years

★ focus on amount of code generated, not quality

★ low quality input specifications (models, config files, ...)

★ ignorance of best practices for modelling language design

★ some of the same people sold objects as a silver-bullet in the 90s ...

• Love affair of software professionals with complexity & technologies

• Underestimating the value of true domain expertise
Status Update

- www.industrialized-software.org/kiss-initiative
  - Reaching a strong consensus on fundamental values and principles for designing and using DSLs
  - Progress towards interoperability between tools

- KISS conference events to date
  - 14 April, **Australian SW Engineering 09**, Gold Coast, Australia
  - 16 June, **Code Generation 09**, Cambridge, UK
  - 25 or 26 October, **OOPSLA 09**, Orlando, Florida

- Upcoming KISS conference events
  - 16 or 17 November, **Automated Software Engineering 09**, Auckland, NZ

- For a powerful message on simplicity, visit [www.spinellis.gr/blog/20090203](http://www.spinellis.gr/blog/20090203)
Fundamental Principles for Designing DSLs

1. There **must** be an economic imperative for the development of a DSL
2. The DSL **must** be meaningful to users of the DSL
3. The DSL **must** be appropriate for the intended machine based processing
4. The DSL definition **must** always be available when processing a model
5. The DSL **should** be cognitively efficient for the users of the DSL
6. The DSL **should** have multiple notations where necessary
7. DSLs **should** be kept small through modularization and integration
8. DSLs **should** offer mechanisms for modularizing and integrating models
9. The DSL **should** be supported by appropriate tooling for DSL users
Guidelines

1. DSL development must assemble sufficient expertise in:
   ★ Problem Domain
   ★ Solution Domain
   ★ Designing DSLs
   ★ Language Implementation Technologies

2. DSL development must anticipate evolution of the DSL

3. DSL modularization should take into account DSL user roles

4. DSL design and validation should use an iterative approach

5. DSL design should involve the construction of concrete examples

6. DSL notations should take into account established domain notations
More examples
to underscore
  • Fundamental Principles
  • Guidelines

Counter-examples
that illustrate the limits of
  • Fundamental Principles
  • Guidelines
Goals beyond OOPSLA 2009

Publish the Foundations for DSSD

1. Confirm signatories for the official wording
2. Publish content on home page of industrialized-software.org
3. Promote KISS on signatory company web sites, refer to KISS principles and guidelines in our daily work with customers, and include appropriate references to industrialized-software.org in articles and case studies

Identification of open challenges

1. Positioning of domain analysis in relation to traditional requirements engineering
2. Avoiding “my model is right and yours is wrong”
3. Avoiding the design by committee effect
4. Concrete steps towards model based interoperability
Presentation of Position Papers

How to Choose a Metamodeling Approach  
Arnon Sturm

The Relationships between Domain Specific & General-Purpose Languages  
Oded Kramer, Arnon Sturm

A Survey on Determining Factors for Modeling Reference Architectures  
Camelia Maga, Nasser Jazdi

Industrialized Software, open questions or consensus?  
Jorn Bettin

30 minutes per paper
Lunch

12:00 - 13:00
Feedback on examples, further examples, counter examples

Five Values
Nine Fundamental Principles
Six Guidelines

10 minutes per item
The Values Explained

• We strive to automate software construction from domain models
• We work with domain-specific assets
• We support the emergence of supply chains for software services
• We see Open standards, as well as Open Source, as driving interoperability
• The methodologies we use conform with the values of the Agile Manifesto
The Values Explained

• We strive to **automate** software construction from domain models therefore we consciously distinguish between building software factories and building software applications

Rationale: Domain models expressed in a domain specific language are significantly more compact than corresponding specifications expressed in a language that does not exploit domain specific abstractions. Often the amount of information that needs to be maintained manually shrinks by a factor of three to five when switching to a domain specific language.

Grouping items that change for related reasons (and consequently at similar rates) is a fundamental architectural principle. This principle motivates the separation of domain models from the underlying language definition (which evolves at a slower rate) and the patterns used to map to implementation technologies (which may change for reasons that are unrelated to the domain).

The separation facilitates modularization, and it creates a sufficiently stable working environment for teams in the context of large systems.
The Values Explained

- We strive to **automate** software construction from domain models
- We work with **domain-specific assets**

which can be anything from models, components, frameworks, generators, to languages and techniques

Rationale: There is increasing evidence that formal, domain-specific, and compact modeling notations are the most elegant vehicle to replace mysterious rituals by accessible domain knowledge.

In contrast, hastily articulated software solutions in general purpose programming languages provide only limited opportunity for uncovering deep domain knowledge, and lead to spurious complexity in the form of verbose and obscure code.

However, domain models alone are not sufficient for the production of software. Models need to be mapped to appropriate implementation technologies, and they need to be supported by appropriate techniques and tooling for analysis, debugging, and maintenance.
The Values Explained

- We strive to **automate** software construction from domain models
- We work with **domain-specific assets**
- We support the emergence of **supply chains** for software services which implies domain-specific specialization and enables mass customization

Rationale: Any state-of-the-art software development and deployment method needs to be able to take advantage of specialized knowledge and resources. The web and web based services have opened the door for approaches that were inconceivable a decade ago. In particular value added software services can increasingly be built and delivered by tapping into available lower level infrastructure services developed by other members of a non-trivial supply chain.
The Values Explained

- We strive to **automate** software construction from domain models
- We work with **domain-specific assets**
- We support the emergence of **supply chains** for software services
- We see **Open standards, as well as Open Source** as driving interoperability (test beds and reference implementations)

Rationale: Durable, functionally rich, and maintainable infrastructure software can only be the result of Darwinian software evolution. We can design software-in-the-small, but we can probably only guide evolution of software-in-the-large.

The most economical way to achieve software evolution is by exploiting and contributing to the mind-boggling soup of working implementations, including both commercial and Open Source software. The size of the community building working implementations means that very real Darwinian selective forces are at work and sometimes lead to surprising results - often in the form of emerging de-facto standards.

In contrast, consortium-based standardization is anachronistic. Bruce Perens (the creator the Open Source Definition) puts it very nicely: “In the consortium projects, there's always the handshake with one hand and a dagger in the other.”
The Values Explained

- We strive to **automate** software construction from domain models
- We work with **domain-specific assets**
- We support the emergence of **supply chains** for software services
- We see **Open standards, as well as Open Source** as driving interoperability
- We use methodologies that conform with the values of the **Agile Manifesto**

Rationale: Without sufficient agility the risk of developing a solution for the wrong problem is unacceptably high. The use of techniques that honor the agile principles provides a collaborative environment where the interests of software users and software developers are aligned.
The Fundamental Principles Explained

1. There must be an economic imperative for the development of a DSL.
2. The DSL must be meaningful to users of the DSL.
3. The DSL must be appropriate for the intended processing.
4. The DSL definition must always be available when processing a model.
5. The DSL should be cognitively efficient for the users of the DSL.
6. The DSL should have multiple notations where necessary.
7. DSLs should be kept small through modularization and integration.
8. DSLs should offer mechanisms for modularizing and integrating models.
9. The DSL should be supported by appropriate tooling for DSL users.
The Fundamental Principles Explained

1. There **must** be an economic imperative for the development of a DSL

The DSL must assist in reducing time to market, development costs, or quality issues in the product line. **Examples:**

- Reducing the size of configuration artifacts in an industrial automation system by a factor of 5
- Systematically eliminating entire classes of configuration errors
- **Enabling domain experts to configure** the functionality of a large software system
- Improving overall productivity in defining, implementing, and testing insurance products by a factor of 2
- Model based modularization of functional specifications of sensor/actuator networks, and drastically increasing test coverage via automatic generation of test cases
- Formalizing the design patterns contained in a product line for building control automation technologies and reducing the size of component specifications by a factor of 6
The Fundamental Principles Explained

1. There **must** be an economic imperative for the development of a DSL
2. The DSL **must** be meaningful to users of the DSL

If DSL is to be useful, the concepts it contains must be intuitive for users to understand. **Examples:**

- ★ Content providers and editors working with the content management system of an international news site relate to articles and ad spaces, **and not to numerical coded object types**
- ★ Insurance product designers relate to risk events, risk factors, and scalar functions, **and not to JAVA classes and J2EE technologies**
- ★ An implementation consultant in the domain of building control automation relates to specific device types (products) and hardware product lines, **and not to the byte codes used in communication protocols** within sensor/actuator networks
The Fundamental Principles Explained

1. There must be an economic imperative for the development of a DSL
2. The DSL must be meaningful to users of the DSL
3. The DSL must be appropriate for the intended machine based processing

Models expressed in the DSL must contain all domain-related information needed to automatically derive the required implementation artifacts, and it must not contain information at the wrong level of abstraction. Examples:

★ A DSL for component models may need to be integrated with a DSL for type set models and a DSL for state machine models in order to allow the derivation of executable code. At the same time the DSL should not be polluted with implementation concepts from the solution space (such as J2EE or MS DotNET specific framework concepts).
The Fundamental Principles Explained

1. There **must** be an economic imperative for the development of a DSL
2. The DSL **must** be meaningful to users of the DSL
3. The DSL **must** be appropriate for the intended machine based processing
4. The DSL definition **must** always be available when processing a model

Without access to the DSL definition, models remain purely conceptual, and cannot be navigated by tools such as model validators, transformers, and generators. **Examples:**

★ Without the DSL definition a model validator can’t confirm compliance of a model with the definition (meta model).

★ Transformations are expressed using the concepts contained in the DSL definition. Of course a transformation may be “compiled” into an executable that has no need to dynamically read the DSL definition, but the DSL definition still needs to be available for the compilation of the transformation, and subsequently it is implicit in the executable.

★ A model based generator is simply a special case of a model transformer, and the same needs for availability of the DSL definition applies.
The Fundamental Principles Explained

1. There **must** be an economic imperative for the development of a DSL
2. The DSL **must** be meaningful to users of the DSL
3. The DSL **must** be appropriate for the intended machine based processing
4. The DSL definition **must** always be available when processing a model
5. The DSL **should** be cognitively efficient for the users of the DSL

The user should have no need to go through a process of mental gymnastics when articulating or reading the details of a model. **Examples:**

★ The DSL should rely on clear terminology and symbols, that is a DSL should avoid using nearly identical names or symbols for unrelated concepts.

★ The different notational aspects that are exposed to the user (say a tree browser, a diagram that shows concepts and their links in the form of boxes and lines, and a property sheet that provides details on the selected element in the diagram) need to relate to each other in an obvious way.
The Fundamental Principles Explained

1. There **must** be an economic imperative for the development of a DSL
2. The DSL **must** be meaningful to users of the DSL
3. The DSL **must** be appropriate for the intended machine based processing
4. The DSL definition **must** always be available when processing a model
5. The DSL **should** be cognitively efficient for the users of the DSL
6. The DSL **should** have multiple notations where necessary

Different users of a DSL may have different notational preferences. **Examples:**

- ★ A regular user may want an emphasis on easy editing (minimal number of required user gestures).
- ★ A user who only accesses models from other models (expressed in a different DSL), may want to be shielded from all those concepts of the first DSL that do not apply to his/her context.
- ★ A DSL may provide a “technical short-hand” notation for experienced modelers, and a more verbose notation with more syntactic sugar for the casual or novice user.
- ★ MS Excel accommodated support for the Lotus 1-2-3 syntax in order to entice users to switch.
The Fundamental Principles Explained

1. There **must** be an economic imperative for the development of a DSL
2. The DSL **must** be meaningful to users of the DSL
3. The DSL **must** be appropriate for the intended machine based processing
4. The DSL definition **must** always be available when processing a model
5. The DSL **should** be cognitively efficient for the users of the DSL
6. The DSL **should** have multiple notations where necessary
7. DSLs **should** be kept small through modularization and integration

If a DSL is not sufficiently modularized, there is a real risk of models becoming large as well. The DSL designer should set a positive example for the DSL user.

**Examples:**

- ★ The UML meta model contains 250+ elements. In a way this signals that complexity is okay or even desirable. Consequently it should not surprise that UML users often create huge monolithic models.

- ★ The meta model used in the Eclipse BPMN Modeler contains about 25 elements.
The Fundamental Principles Explained

1. There **must** be an economic imperative for the development of a DSL
2. The DSL **must** be meaningful to users of the DSL
3. The DSL **must** be appropriate for the intended machine based processing
4. The DSL definition **must** always be available when processing a model
5. The DSL **should** be cognitively efficient for the users of the DSL
6. The DSL **should** have multiple notations where necessary
7. DSLs **should** be kept small through modularization and integration
8. DSLs **should** offer mechanisms for modularizing and integrating models

Even a tiny DSL may allow the creation of arbitrarily large models. **Examples:**

- The DSL definition required for representing unlimited tree structures only needs to contain one or two elements. Hence such a “tree structure” DSL needs to offer a mechanism for modularizing trees.
The Fundamental Principles Explained

1. There **must** be an economic imperative for the development of a DSL
2. The DSL **must** be meaningful to users of the DSL
3. The DSL **must** be appropriate for the intended machine based processing
4. The DSL definition **must** always be available when processing a model
5. The DSL **should** be cognitively efficient for the users of the DSL
6. The DSL **should** have multiple notations where necessary
7. DSLs **should** be kept small through modularization and integration
8. DSLs **should** offer mechanisms for modularizing and integrating models
9. The DSL **should** be supported by appropriate tooling for DSL users

The modeler using a DSL may work with it on a daily basis. **Examples:**

★★ Professional users of a DSL will compare the tooling against products such as MS Excel!
1. DSL development must assemble sufficient expertise in:
   - Problem Domain
   - Solution Domain
   - Designing DSLs
   - Language Implementation Technologies

2. DSL development must anticipate evolution of the DSL.

3. DSL modularization should take into account DSL user roles.

4. DSL design and validation should use an iterative approach.

5. DSL design should involve the construction of concrete examples.

6. DSL notations should take into account established domain notations.
The Guidelines Explained

1. DSL development **must** assemble sufficient expertise in:
   - Problem Domain
   - Solution Domain
   - Designing DSLs
   - Language Implementation Technologies

Since all four areas of expertise are essential it is easy to run into a bottleneck.

**Examples:**

- The biggest bottleneck tends to be access to truly deep problem domain expertise, as domain experts often are in hot demand by all corners of the organization.
- Sufficiently deep expertise in relevant implementation technologies is required to build reference implementations. Expertise may need to be sourced externally if the technology is new to the team.
- DSL design requires someone who is talented at identifying useful abstractions, is experienced in validating DSLs for completeness, and in creating sufficiently modular sets of integrated DSLs.
- A DSL implementation always includes development of code templates or DSL execution engines.
The Guidelines Explained

1. DSL development **must** assemble sufficient expertise in:
   - Problem Domain
   - Solution Domain
   - Designing DSLs
   - Language Implementation Technologies

2. DSL development **must** anticipate evolution of the DSL

Domain knowledge and the context of the domain always evolves. **Examples:**

- The development process of a DSL tends to be incremental, in particular during the development of the first version. As soon as valuable test and production models have been created in a DSL, any further DSL evolution needs to consider the migration of model content.

- Appropriate care should be taken only to include sufficiently stable concepts in the first version of a DSL. A DSL can relatively easily be extended with further concepts later, but refactoring extensive amounts of model content expressed in a poorly designed DSL into a refactored DSL is much harder.
1. DSL development must assemble sufficient expertise in:
   - Problem Domain
   - Solution Domain
   - Designing DSLs
   - Language Implementation Technologies

2. DSL development must anticipate evolution of the DSL

3. DSL modularization should take into account DSL user roles

Roles provide clues for DSL modularization & usability requirements. Examples:

- A DSL used by a software developer will typically need to be integrated into an IDE such as Eclipse or MS Visual Studio. A DSL for an insurance product designer needs no IDE integration, but may require smooth integration into a legacy application environment.

- In the domain of automated building control, implementation consultants, device designers, and hardware product line architects may all require dedicated DSLs.
The Guidelines Explained

1. DSL development **must** assemble sufficient expertise in:
   - ★ Problem Domain
   - ★ Solution Domain
   - ★ Designing DSLs
   - ★ Language Implementation Technologies

2. DSL development **must** anticipate evolution of the DSL

3. DSL modularization **should** take into account DSL user roles

4. DSL design and validation **should** use an iterative approach

DSL designs regularly need to be validated with realistic models. **Examples:**

★ The applications resulting from using the DSL need to be validated by future users. User feedback typically leads to new insights and usability requirements, hence an iterative approach is essential.
The Guidelines Explained

1. DSL development **must** assemble sufficient expertise in:
   - ★ Problem Domain
   - ★ Solution Domain
   - ★ Designing DSLs
   - ★ Language Implementation Technologies

2. DSL development **must** anticipate evolution of the DSL

3. DSL modularization **should** take into account DSL user roles

4. DSL design and validation **should** use an iterative approach

5. DSL design **should** involve the construction of concrete examples
   - ★ Domain experts need to validate that the DSL allows them to express all their requirements.
   - ★ Platform architects and transformation developers need to validate that the DSL is sufficiently complete to allow automatic derivation of all required implementation artifacts.
The Guidelines Explained

1. DSL development **must** assemble sufficient expertise in:
   - ★ Problem Domain
   - ★ Solution Domain
   - ★ Designing DSLs
   - ★ Language Implementation Technologies

2. DSL development **must** anticipate evolution of the DSL

3. DSL modularization **should** take into account DSL user roles

4. DSL design and validation **should** use an iterative approach

5. DSL design **should** involve the construction of concrete examples

6. DSL notations **should** take into account established domain notations

**Examples:**
- ★ MS Excel accommodated support for the Lotus 1-2-3 syntax in order to entice users to switch.
Conclusion & logistics of next steps
Thank you!

Jorn Bettin
jbe @ sofismo.ch
Skype jorn_bettin
+41 62 891 0987