Domain Specific Languages

Implementation Techniques

Markus Völter
voelter@acm.org
www.voelter.de

 © 2007/8 Markus Völter

About me

Markus Völter
voelter@acm.org
www.voelter.de

- Independent Consultant
- Based out of Goeppingen, Germany
- Focus on
  - Model-Driven Software Development
  - Software Architecture
  - Product Line Engineering

Founder and Editor of

 © 2007/8 Markus Völter

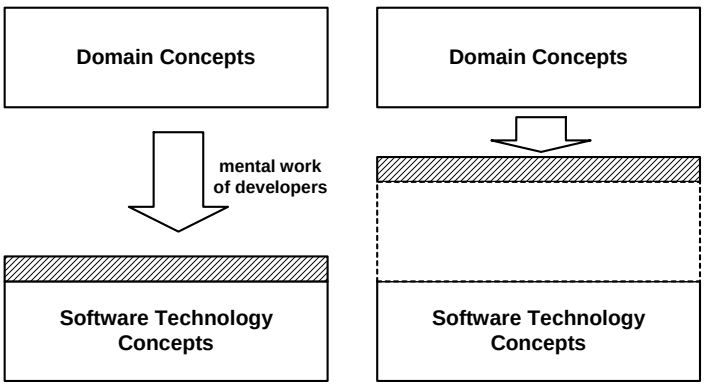
CONTENTS

- Why DSLs
- DSL Categorization
- External DSLs
  - Generative: Eclipse & Co
  - Interpreted: MetaEdit+
- Dynamic Internal DSLs
  - Runtime Metaprogramming: Ruby
- Compiled Internal DSLs
  - Compile Time Metaprogramming: Converge
  - Functional Programming: Scala
- Language Workbenches
  - Intentional Software Domain Workbench
- Summary & Further Reading

 ©2007/8 Markus Völter

Core Concepts

- DSLs are about making software development more **domain-related** as opposed to **computing related**. It is also about making software development in a certain domain **more efficient**.



 ©2007/8 Markus Völter

CONTENTS

- **Why DSLs**
- DSL Categorization
- External DSLs
  - Generative: Eclipse & Co
  - Interpreted: MetaEdit+
- Dynamic Internal DSLs
  - Runtime Metaprogramming: Ruby
- Compiled Internal DSLs
  - Compile Time Metaprogramming: Converge
  - Functional Programming: Scala
- Language Workbenches
  - Intentional Software Domain Workbench
- Summary & Further Reading

 ©2007/8 Markus Völter

What is a DSL?

A DSL is a **concise**, **processable language** for describing a **specific concern** when building a **system** in a specific **domain**. The **abstractions** and **notations** used are **tailored** to the **stakeholders** who specify that particular concern.

 ©2007/8 Markus Völter

What is a DSL?

A DSL is a **concise**, **processable language** for describing a **specific concern** when building a **system** in a specific **domain**. The **abstractions** and **notations** used are **tailored** to the **stakeholders** who specify that particular concern.

- Specifying something with a DSL rather than in implementation code **must be more concise** otherwise what's the point.
- **Details** that can automatically be derived are not described.

© 2007/8 Markus Völter

What is a DSL?

A DSL is a **concise**, **processable language** for describing a **specific concern** when building a **system** in a specific **domain**. The **abstractions** and **notations** used are **tailored** to the **stakeholders** who specify that particular concern.

- We want to **process** the "DSL program" **with a tool**: analyser, transformer, generator, interpreter, meta program
- Hence the DSL must be **formal enough** for the tool to know what to do with a specific DSL program
- It need **not (and should not!) be turing complete**, it need not be executable (**executable != precise**)

© 2007/8 Markus Völter

What is a DSL?

A DSL is a **concise**, **processable language** for describing a **specific concern** when building a **system** in a specific **domain**. The **abstractions** and **notations** used are **tailored** to the **stakeholders** who specify that particular concern.

- It is not simply an API. It must have a **language feel** to it.
- The **grammar** is important.
- It has a well-defined **notation**.

©2007/8 Markus Völter

What is a DSL?

A DSL is a **concise**, **processable language** for describing a **specific concern** when building a **system** in a specific **domain**. The **abstractions** and **notations** used are **tailored** to the **stakeholders** who specify that particular concern.

- Typically a single DSL does not describe the complete system, it rather specifies only a particular **concern**, **viewpoint** or **aspect**.
- The overall system is described with m**odels built with various DSLs** and/or 3GLs– one for each relevant concern. DSLs need to be **integratable**.

©2007/8 Markus Völter

What is a DSL?

A DSL is a **concise**, **processable language** for describing a **specific concern** when building a **system** in a specific **domain**. The **abstractions** and **notations** used are **tailored** to the **stakeholders** who specify that particular concern.

- In our context, we consider DSLs to be not any arbitrary language in a domain used by domain experts.
- We limit our use to languages that are used in the context of **building software systems**.

What is a DSL?

A DSL is a **concise**, **processable language** for describing a **specific concern** when building a **system** in a specific **domain**. The **abstractions** and **notations** used are **tailored** to the **stakeholders** who specify that particular concern.

- A domain is a **bounded** area of **knowledge or interest**.
- A domain often has a specific **culture, formalism or language**.
- Concerns can be **technical** (architecture, persistence, …) or **business-related** (insurance, banking, …)
- A DSL can be built **bottom up** („framework DSL“) or **top down** („business DSL“)

What is a DSL?

A DSL is a **concise**, **processable language** for describing a **specific concern** when building a **system** in a specific **domain**. The **abstractions** and **notations** used are **tailored** to the **stakeholders** who specify that particular concern.

- Core to a DSL are the **abstractions** that are relevant to the concern. They are formalized via a **meta model.**
- The abstractions (the meta model) is what the **processing tool** works with mainly.

 ©2007/8 Markus Völter

What is a DSL?

A DSL is a **concise**, **processable language** for describing a **specific concern** when building a **system** in a specific **domain**. The **abstractions** and **notations** used are **tailored** to the **stakeholders** who specify that particular concern.

- The notations are important for a DSL – they are what the **stakeholder** works with.
- Notation is the defining criterion that **distinguishes a DSL** from an API.
- Common notations are **graphical** diagrams, **textual** notations or tables/matrices

 ©2007/8 Markus Völter

What is a DSL?

A DSL is a **concise**, **processable language** for describing a **specific concern** when building a **system** in a specific **domain**. The **abstractions** and **notations** used are **tailored** to the **stakeholders** who specify that particular concern.

- Standardising DSLs in a large scope does not necessarily make a lot of sense – they need to be **tailored** to the domain in which they are used.
- It makes sense to build families/product lines of DSLs so you can **quickly create a variant** of an existing DSL for a given (sub) domain.
- The **effort** required to **build / tailor** a DSL is very **relevant** in the overall considerations.

 ©2007/8 Markus Völter

What is a DSL?

A DSL is a **concise**, **processable language** for describing a **specific concern** when building a **system** in a specific **domain**. The **abstractions** and **notations** used are **tailored** to the **stakeholders** who specify that particular concern.

- Just as there are different concerns in a system, there are **different stakeholders**
- Different **notations** might be appropriate for those stakeholders
- Different stakeholders specify their concern **at different times during** the development process.

 ©2007/8 Markus Völter

CONTENTS

- Why DSLs
- **DSL Categorization**
- External DSLs
  - Generative: Eclipse & Co
  - Interpreted: MetaEdit+
- Dynamic Internal DSLs
  - Runtime Metaprogramming: Ruby
- Compiled Internal DSLs
  - Compile Time Metaprogramming: Converge
  - Functional Programming: Scala
- Language Workbenches
  - Intentional Software Domain Workbench
- Summary & Further Reading

 ©2007/8 Markus Völter

DSL

- There is a large variety of **DSL flavours.**
- Today we will learn about several of them.
- This intro tries to **categorize** these approaches along the following dimensions:
  - Domain Selection
  - Expressive Power
  - Concrete Syntax
  - Execution
  - Integration
  - Tool Support

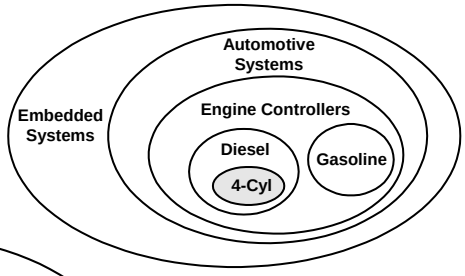
 ©2007/8 Markus Völter

Domain: Technical vs. Functional

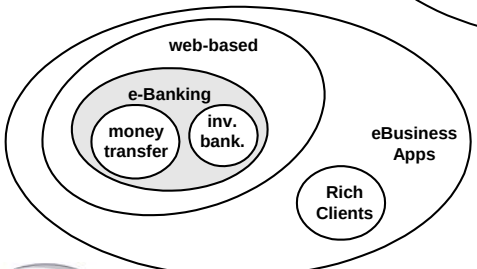
- In the context of software development it is useful to distinguish (at least) **two kinds of domains:**
  - **Technical Domains** adress key technical issues related to software development such as
    - Distribution, Failover and Load-Balancing
    - Persistence and Transactions
    - GUI Design
    - Concurreny and Realtime
  - **Functional Domains** represent the business/professional issues; examples include
    - Banking
    - Human resource management
    - Insurance
    - Engine Controllers
    - Astronomical Telescope Control

Specific: Wide vs. Narrow

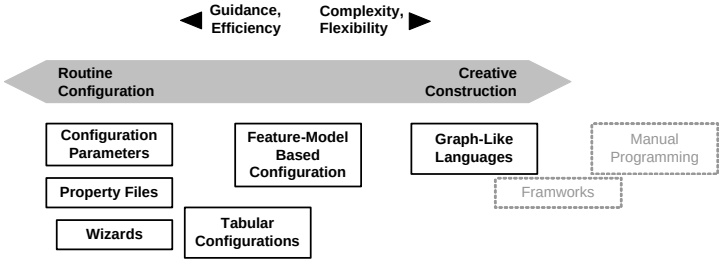
- Since Domains can be of any size or granularity, it is useful to **structure domains hierarchically**.
- Automotive Example:



- eBanking Example:



Expressive Power

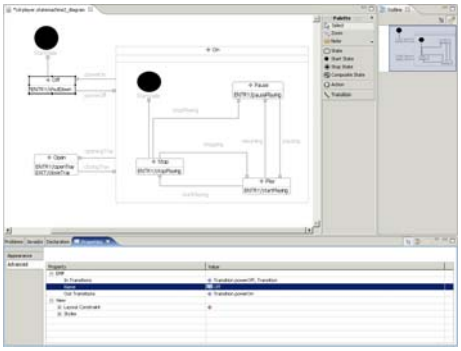


- The more you can move your DSL „form“ to the **configuration** side, the **simpler** it typically gets.
- Mature domains often (but not always) are described by configuration

 ©2007/8 Markus Völter

Syntax: Graphical vs. Textual vs. Tables vs. Forms vs ...

- Textual integrates nicely with **current infrastructure**
- Graphical is a good **sales argument** and the right decision for certain domains.
- Textual DSLs should be used more often
  (note: graphical rendering is different from graphical editing).



 ©2007/8 Markus Völter

Execution

- You can either **interpret** the model („Virtual Machine“)
- … or **transform** it into some executable artifact
  - Transformation
  - Generation
  - Compilation
- Each has various **tradeoffs** wrt.
  - Performance
  - Code size
  - (Runtime) Flexibility
  - Reflective features
- You can also **combine** things
  - E.g. generate something that is then going to be interpreted

 ©2007/8 Markus Völter

Integration

- A DSL can either be **separate** from „normal“ programming languages …
- … or it can be **embedded** …
- **External DSLs** are more flexible wrt. to concrete syntax
- **Internal DSLs** simplify symbolic integration
- How easy is to (symbolically) integrate several external DSLs?
- Often (but by no means always), internal DSLs are interpreted, external DSLs are often compiled/generated

 ©2007/8 Markus Völter

Tool Support

- Do you just have a language (and compiler/interpreter) or also **additional infrastructure**, such as
  - Nice, code-completing and syntax-highlighting editor?
  - Debugger
  - Static (edit time) constraint validation
- This aspect is not a core characteristic of the DSL, but it is certainly an important consideration when selecting a DSL „flavour“ in practice.

 ©2007/8 Markus Völter

A note on the scope of this presentation

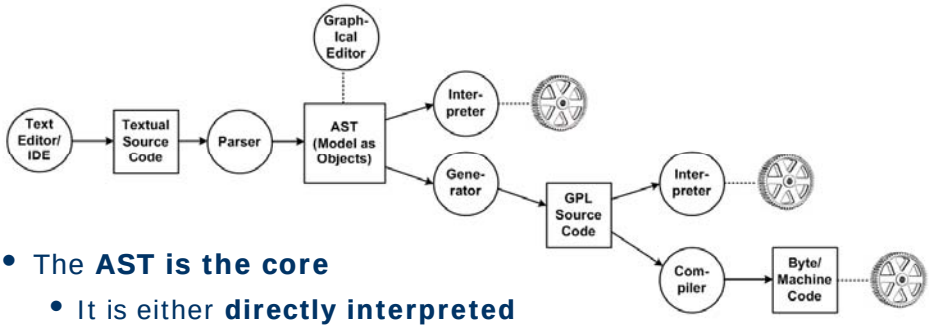
- In this presentation, we look at tools and technologies that allow you to **build your own DSLs.**
- We do not look at tools that **have a specific DSL built-in** in order for you to use it and build programs
- Examples of such tools include
  - Enterprise 4GLs
  - Engineering tools such as LabView, Matlab/Simulink, Mathematica

 ©2007/8 Markus Völter

CONTENTS

- Why DSLs
- DSL Categorization
- **External DSLs**
  - Generative: Eclipse & Co
  - Interpreted: MetaEdit+
- Dynamic Internal DSLs
  - Runtime Metaprogramming: Ruby
- Compiled Internal DSLs
  - Compile Time Metaprogramming: Converge
  - Functional Programming: Scala
- Language Workbenches
  - Intentional Software Domain Workbench
- Summary & Further Reading

External DSLs: Implementation



- The **AST is the core**
  - It is either **directly interpreted**
  - Or **transformed** into GPL code (which is then interpreted or further transformed, i.e. compiled)

Examples:
oAW, GMF, xText, etc.
(„classic“ MDSD)

- AST can be **created** by
  - **Direct editing** (typically via a graphical editor)
  - Or **via a parser** from a typically textual representation

CONTENTS

- Why DSLs
- DSL Categorization
- External DSLs
  - **Generative: Eclipse & Co**
  - Interpreted: MetaEdit+
- Dynamic Internal DSLs
  - Runtime Metaprogramming: Ruby
- Compiled Internal DSLs
  - Compile Time Metaprogramming: Converge
  - Functional Programming: Scala
- Language Workbenches
  - Intentional Software Domain Workbench
- Summary & Further Reading

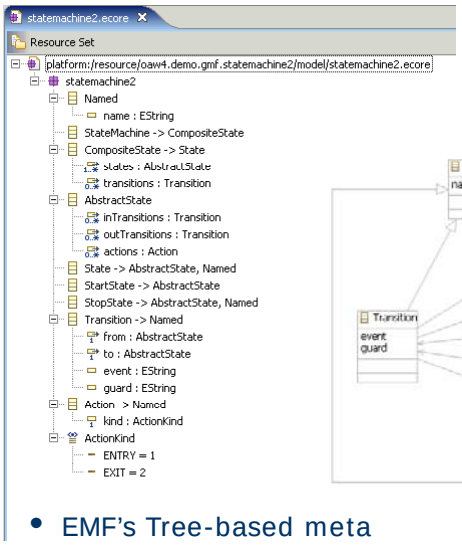
 ©2007/8 Markus Völter

External DSLs: Classic Approach

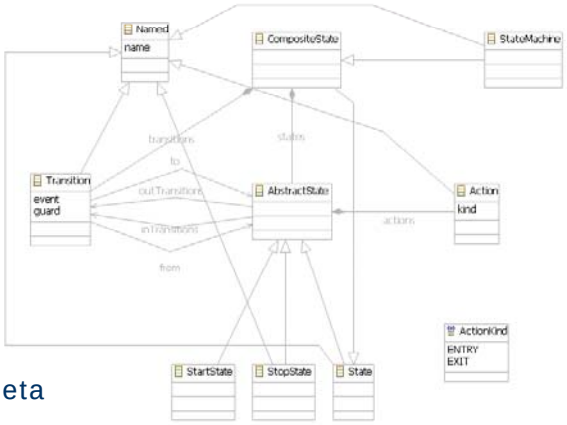
- You start by defining a **meta model**.
- You then define a **concrete syntax** and an **editor** (and often a parser to deserialize a model file)
  - This results in an **object graph** representing the model
- And finally, you process the model by
  - Defining a set of **transformations** or **generators** that map the model to some kind of **executable representation**.
  - Or by building an **interpreter** that directly executes **side effects** as it processes the model
- There are various „implementations“ of this approach, among them
  - Eclipse/EMF/GMF/oAW
  - Metacase‘s MetaEdit+
  - ...

 ©2007/8 Markus Völter

External DSLs: Classic Approach I: Meta Model



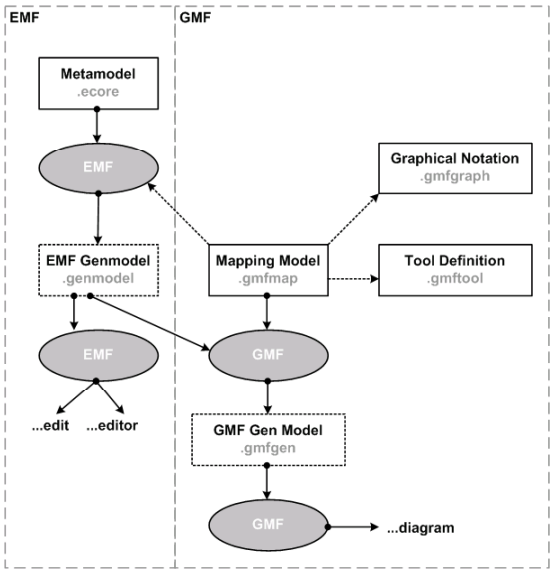
- A graphical, GMF-based editor



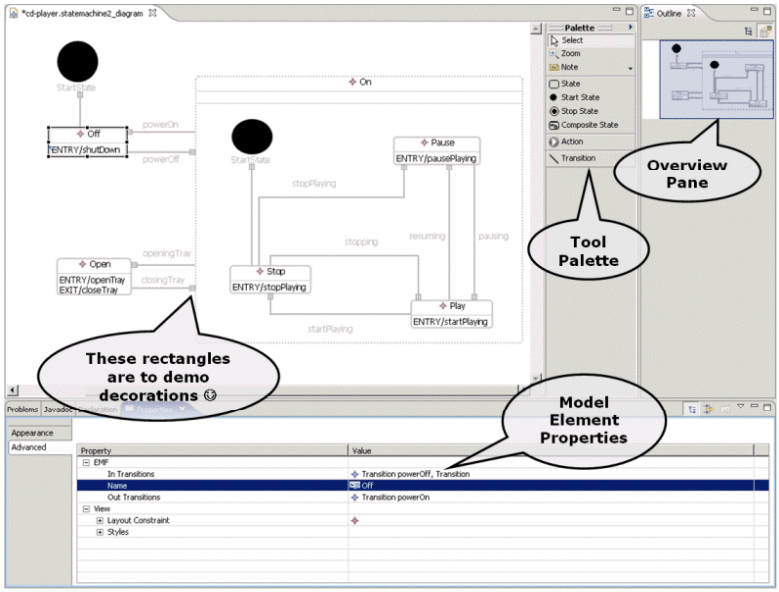
- EMF's Tree-based meta model editor

External DSLs: Classic Approach II: Concrete Syntax

- In Eclipse GMF, you use a number of **additional models** that map the domain meta model to graphical concrete syntax elements.
- These models, together with the domain model, are used by the **GMF generator** to build the editor plugin.

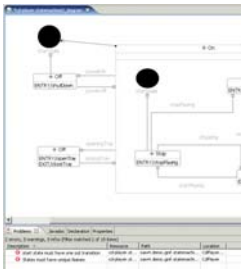


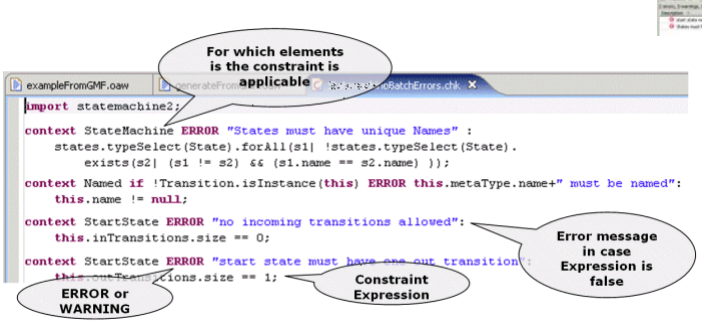
External DSLs: Classic Approach III: Concrete Syntax III



External DSLs: Classic Approach IV: Constraints

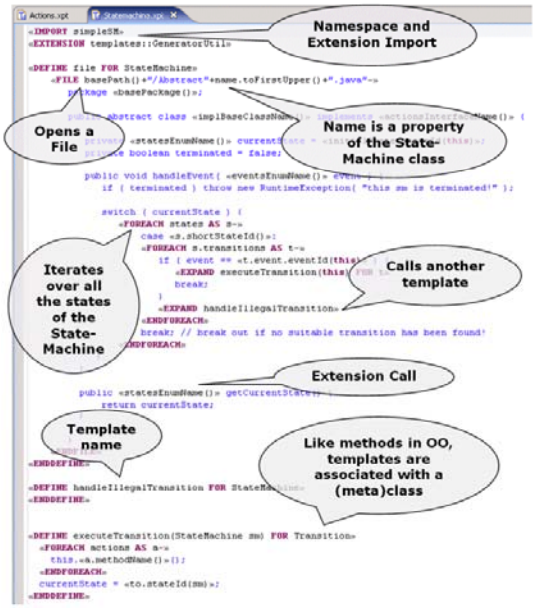
- **Additional constraints** can be defined to validate the model.
  - Typically, some OCL-like language is used (here: oAW Checks)





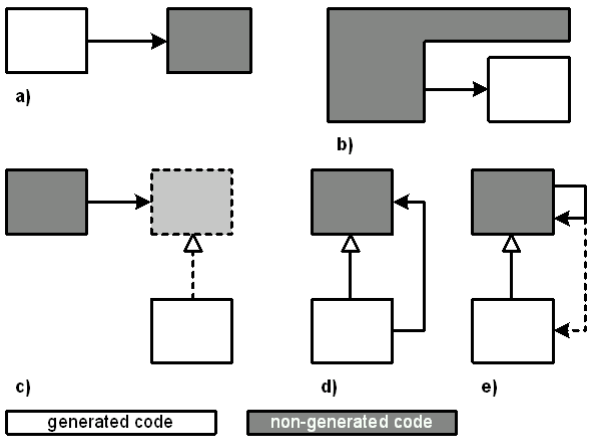
External DSLs: Classic Approach V: Code Generation

- Code Generation is typically done using **template language**
- These contain **template control code**, **model access code** as well as **target language code**.
- Special **escape characters** distinguish between them



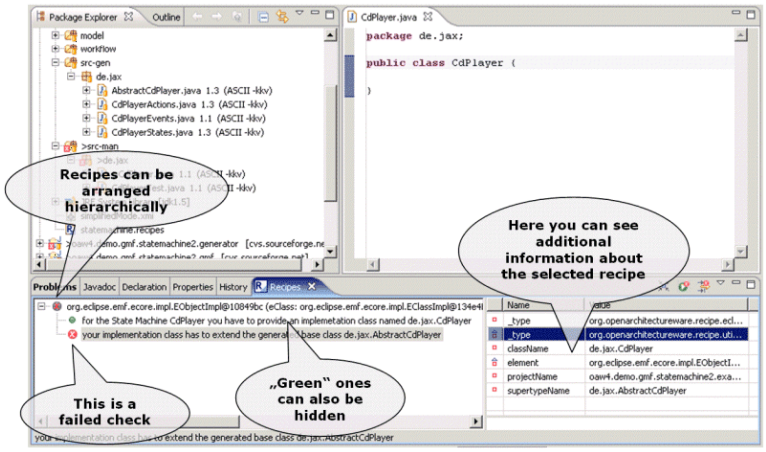
External DSLs: Classic Approach V: Code Integration

- **Integration of generated code and manually written code** needs to be taken care of explicitly, eg. using design patterns



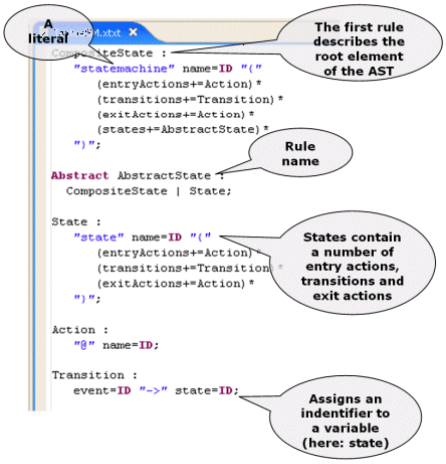
External DSLs: Classic Approach VI: Code Integration II

- **Recipe Frameworks** can also help. They check the sum of the code (generated and manually written) wrt. to user-defined consistency rules.



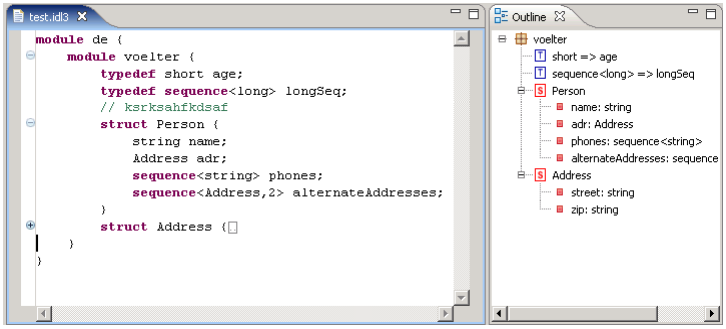
External DSLs: Classic Approach VI: Textual Editor I

- The syntax is defined
  - Either as some kind of **EBNF-like** structure
  - Or as an **annotation** of the domain meta model
- **Additional descriptions** let you define
  - **Outline** view labels and icons
  - Custom **constraints**
  - Code completion hints
  - Etc.



External DSLs: Classic Approach VI: Textual Editor II

- The editor is then generated as an Eclipse plugin, e.g.



- Typically, the tool also generates a **parser** that allows you to parse text files of the appropriate format in a backend code generator.

 ©2007/8 Markus Völter

I could have talked about...

- **Microsoft DSL tools**
  - allows you to build graphical DSLs (just like GMF, although with a somewhat friendlier tooling)
  - Generate code with a not-so-powerful transformation language
- All the **MDA** stuff... and **Executable UML** ...
- A whole bunch of other Open Source and commercial modeling, code generation and transformation tools.

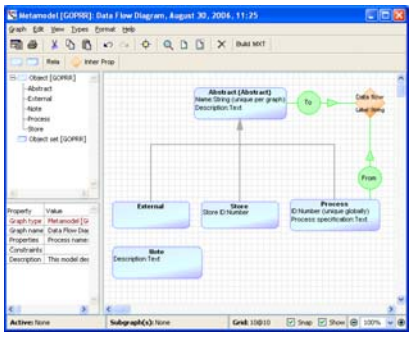
 ©2007/8 Markus Völter

CONTENTS

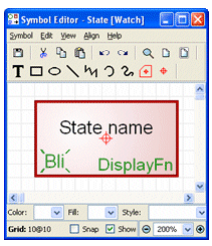
- Why DSLs
- DSL Categorization
- External DSLs
  - Generative: Eclipse & Co
  - **Interpreted: MetaEdit+**
- Dynamic Internal DSLs
  - Runtime Metaprogramming: Ruby
- Compiled Internal DSLs
  - Compile Time Metaprogramming: Converge
  - Functional Programming: Scala
- Language Workbenches
  - Intentional Software Domain Workbench
- Summary & Further Reading

MetaEdit+

- **MetaEdit+** from Metacase, Finland, is a tool to build graphical DSLs in a complete tool:
  - It comes with its own meta meta model
- Define a **meta model** (using dialogs, or graphically ➔)

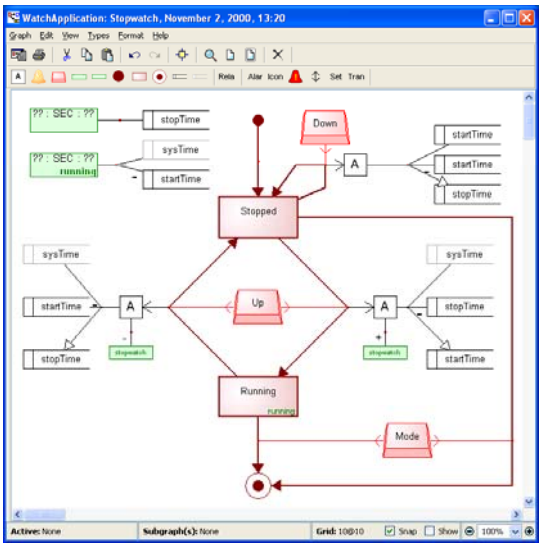


- You can then **draw the symbols** and associate them with the meta classes ➔



MetaEdit+ II

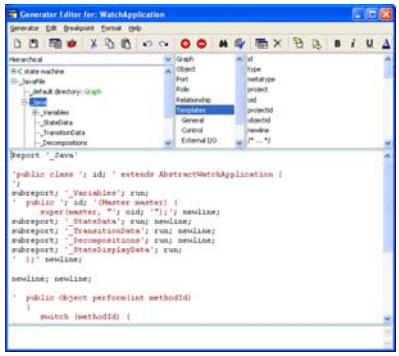
- MetaEdit then provides a **graphical editor for building models**
  - note that this editor is not generated, rather it is "interpreted" inside the tool itself



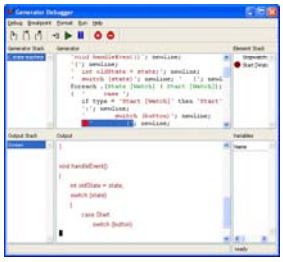
©2007/8 Markus Völter

MetaEdit+ III

- Finally, you can **define code-generation templates** in order to generate code from your models



- Including a **template debugger**:



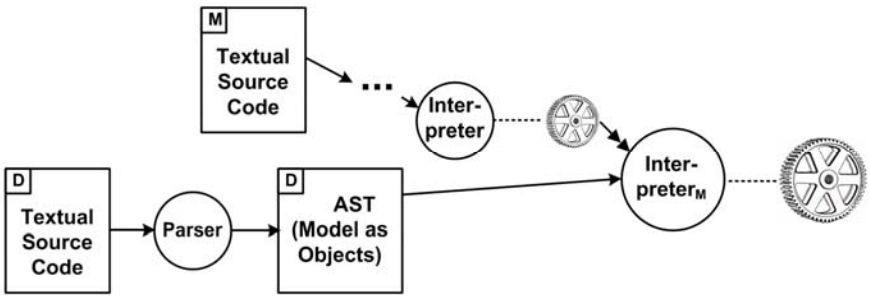
©2007/8 Markus Völter

CONTENTS

- Why DSLs
- DSL Categorization
- External DSLs
  - Generative: Eclipse & Co
  - Interpreted: MetaEdit+
- **Dynamic Internal DSLs**
  - Runtime Metaprogramming: Ruby
- Compiled Internal DSLs
  - Compile Time Metaprogramming: Converge
  - Functional Programming: Scala
- Language Workbenches
  - Intentional Software Domain Workbench
- Summary & Further Reading

 © 2007/8 Markus Völter

Internal DSLs: Interpreted I: Metaprogramming II

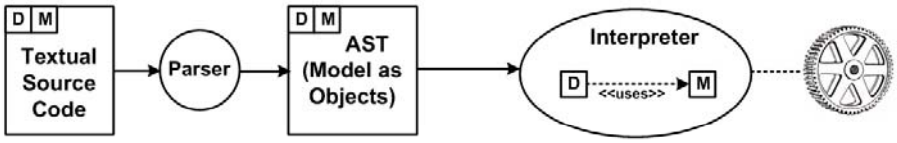


- The (often separate) **metaprogram M modifies the interpreter** effectively producing a custom interpreter that knows about M and can interpret DSL programs D
- The modified interpreter **interprets the DSL program D** as part of the host program

Example: CLOS

 © 2007/8 Markus Völter

Internal DSLs: Interpreted I: Metaprogramming



- Source code contains the **metaprogram** (M) defining the DSL as well as a **program** in the DSL (D)
- After parsing, the **AST contains the metaprogram and the program** (this is possible, since D is syntactically compatible with the host language)

Examples:
Lisp, Ruby

- In the interpreter, the **DSL program D uses the metaprogram M** and produces the desired effect

©2007/8 Markus Völter

CONTENTS

- Why DSLs
- DSL Categorization
- External DSLs
  - Generative: Eclipse & Co
  - Interpreted: MetaEdit+
- Dynamic Internal DSLs
  - **Runtime Metaprogramming: Ruby**
- Compiled Internal DSLs
  - Compile Time Metaprogramming: Converge
  - Functional Programming: Scala
- Language Workbenches
  - Intentional Software Domain Workbench
- Summary & Further Reading

©2007/8 Markus Völter

Internal DSLs in Ruby I: Simple Example

- Simple Example: **Ordering Coffee**

```
include StarbucksDSL

order = latte venti, half_caf, non_fat, no_foam, no_whip
print order.prepare
```

- Ruby **Syntax** that helps in building DSLs
  - Optional parentheses
  - Symbols
  - Blocks
  - Literal arrays and hashes
  - Variable-length arguments

Ruby Examples taken with permission from
a presentation by **Obie Fernandez** at
http://obiefernandez.com/presentations/obie_fernandez-agile_dsl_development_in_ruby.pdf

Internal DSLs in Ruby II: Process

- Ruby DSL development is **syntax-oriented:**
  - Don't try to do an abstract metamodel first
  - Capture your **DSL concepts in valid Ruby syntax**, but don't worry about implementation
  - Iterate over your Ruby DSL syntax until authorsagree that it faithfully represents the domain,then work on the implementation
- This approach is necessary primarily because
  - there *are* **limits** to what you can do with Ruby syntax, you have to approximate the syntax iteratively
  - And often you **won't even build an explicit meta model**, you'll interpret the DSL on the fly

Internal DSLs in Ruby III: Rails

- **Ruby on Rails** is an internal Ruby DSL for building web applications.

```
class Story < Goal

  acts_as_state_machine :initial => :unscheduled

  state :unscheduled
  state :stalled
  state :scheduled
  state :in_progress
  state :untested
  state :accepted

  belongs_to :scheduled_for, :class_name=> 'Iteration'
  belongs_to :finished_in, :class_name=> 'Iteration'

  validates_presence_of :body
  validates_numericality_of :points
```


Internal DSLs in Ruby IV: Rails II

- Rails uses many **advanced DSL-building features** of Ruby.

```
class Project < ActiveRecord::Base

  after_create  :setup, :create_backlog, :create_progress, :create_done

  has_many :goals
  has_many :stages, :order => :position
  has_many :iterations, :order => 'date DESC'
  has_many :milestones, :order => 'date DESC'

  has_many :stories, :order => :position do
    %w(unscheduled scheduled in_progress untested accepted).each do |method|
      define_method(method) do |*args|
        find_all_by_state(method, *args)
      end
    end
  end

  has_and_belongs_to_many :people

  def backlog
    stories.unscheduled
  end

  def done
    stages.last
```

Internal DSLs in Ruby V: Instantiation

- **Instantiation:** Building DSLs by simply instantiating objects and calling methods, exploiting Ruby's flexible syntax.

```
require 'builder'

my_faves = {
  'candy' => 'Neccos',
  'novel' => 'Empire of the Sun',
  'holiday' => 'Easter'
}

xml = Builder::XmlMarkup.new( :target => $stdout, :indent => 2 )

xml.instruct! :xml, :version => "1.1", :encoding => "US-ASCII"

xml.favorites do
 my_faves.each do | name, choice |
  xml.favorite( choice, :item => name )
 end
end
```

 ©2007/8 Markus Völter

Internal DSLs in Ruby VI: Class Macros

- **Class Macros:** DSL as (static) methods on some ancestor class, subclasses use those methods to tweak the behavior/structure of themselves

```
class Ruleset < ActiveRecord::Base

  belongs_to  :global_criteria
  belongs_to  :partner
  has_many    :comments, :dependent => :delete_all
  has_many    :ruleset_runs

  after_create :set_global_criteria

  acts_as_state_machine :initial => :draft
  state :draft
  state :pending_approval
  state :approved
  state :denied
  state :active           # eligible for inclusion in run
  state :expired

  event :clone_ruleset do
    transitions :to => :draft
  end
```

 ©2007/8 Markus Völter

Internal DSLs in Ruby VII: Top Level Methods

- **Top Level Methods**

```
namespace :schema do
  desc "Wipe Database"
  task :clean => :environment do
    connection = ActiveRecord::Base.connection
    connection.tables.each do |table|
      puts "* Dropping table '#{table}'"
      connection.execute "drop table #{table}"
    end
    if connection.respond_to? :sequences
      connection.sequences.each do |sequence|
        puts "* Dropping sequence '#{sequence}'"
        connection.execute "drop sequence #{sequence}"
      end
    end
    if ActiveRecord::Base.connection.class.to_s == 'Acti
      begin
        connection.execute "DROP FUNCTION months_betweer
      rescue Exception
      end
    end
  end
```

 ©2007/8 Markus Völter

Internal DSLs in Ruby VIII: Contexts

- **Contexts:** Your DSL is defined as methods of some object, but that object is really just a “sandbox”. Interacting with the object’s methods modify some state in the sandbox, which is then queried by the application.

```
desc "Transactional deployment plus restart"
task :deploy do
  transaction do
    update_code
    disable_web
    symlink
    migrate
  end

  restart
  enable_web
end
```

 ©2007/8 Markus Völter

Internal DSLs in Ruby IX: Meta Programming Facilities

- Ruby provides a number of **meta programming** facilities that are used to implement DSLs:
  - **Symbols:** less noisy than strings
  - **Blocks** enabling delayed evaluation and passing around of behaviour
  - **eval**, **instance_eval**, and **class_eval** to dynamically evaluate strings as code in various contexts
  - **define_method** to dynamically define new methods
  - **methodMissing:** callback that is invoked whenever you invoke a method that is not available (and then you can react accordingly)

 ©2007/8 Markus Völter

I could have talked about...

- **Smalltalk**
  - Smalltalk's dynamic object model let's you do some of the same things
- **CLOS**, as mentioned before...

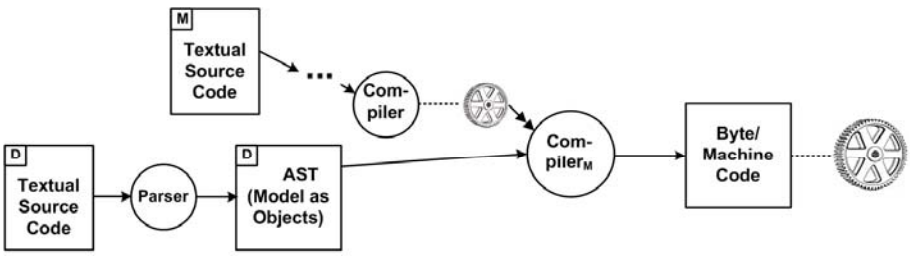
 ©2007/8 Markus Völter

CONTENTS

- Why DSLs
- DSL Categorization
- External DSLs
  - Generative: Eclipse & Co
  - Interpreted: MetaEdit+
- Dynamic Internal DSLs
  - Runtime Metaprogramming: Ruby
- **Compiled Internal DSLs**
  - Compile Time Metaprogramming: Converge
  - Functional Programming: Scala
- Language Workbenches
  - Intentional Software Domain Workbench
- Summary & Further Reading

 ©2007/8 Markus Völter

Internal DSLs: Compiled I

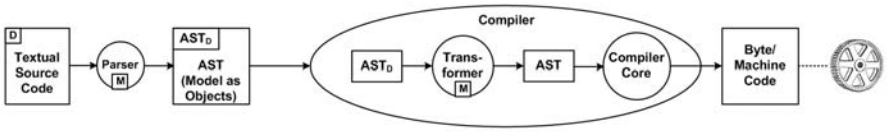


- The **metaprogram modifies the Compiler** to understand D programs (aka open compilers, Compile-Time MOP)
- The **Compiler$_M$ now understands D** – depending on how far the modification goes, D can have specific syntax or not
- In **homogenous systems**, the language for implementing M are the same as the host language (program and metaprograms can be mixed, too).

Example: OpenC++

 ©2007/8 Markus Völter

Internal DSLs: Compiled II



- The program contains **host code and DSL code** D.
- A **parser that knows about D** builds an AST with a part that is specific to M (AST_D).
- Inside the compiler, a **special transformer** (M-specific) transforms AST_D into a regular AST which then compiled with the compiler code of the host language.
- In **homogenous systems**, the language for implementing $Parser_M$ and $Transformer_M$ are the same as the host language (program and metaprograms can be mixed, too).

Example: Converge

© 2007/8 Markus Völter

CONTENTS

- Why DSLs
- DSL Categorization
- External DSLs
  - Generative: Eclipse & Co
  - Interpreted: MetaEdit+
- Dynamic Internal DSLs
  - Runtime Metaprogramming: Ruby
- Compiled Internal DSLs
  - **Compile Time Metaprogramming: Converge**
  - Functional Programming: Scala
- Language Workbenches
  - Intentional Software Domain Workbench
- Summary & Further Reading

© 2007/8 Markus Völter

Converge I: Macro System

- Converge's **Macro** facility:

```
2 + 3
```

- ... evaluates to 5

Converge examples taken with permission from a presentation by **Laurence Tratt.** More details at tratt.net and convergepl.org

```
$<x>
```

- **Splice:** evaluates x at compile-time; the AST returned overwrites the splice.

```
[| 2 + 3 |]
```

- **Quasi-quote** evaluates to hygienic AST representing 2 + 3.

```
[| 2 + ${x} |]
```

- **Insertion** 'inserts' the AST x into the AST being created by the quasi-quotes.

 ©2007/8 Markus Völter

Converge II: Macro System Example

- Consider the following example

```
func expand_power(n, x):
  if n == 0: return [| 1 |]
  else: return [| ${x} * ${expand_power(n - 1, x)} |]
func mk_power(n):
  return [|
    func (x): return ${expand_power(n, [| x |])}
  |]
power3 := $<mk_power(3)>
```

- The function *power3* looks like:

```
power3 := func (x): return x * x * x * 1
```

- This happens **during the compilation phase**; i.e. the latter definition is compiled into byte code

 ©2007/8 Markus Völter

Converge III: Buiding DSLs I

- To build DSLs in converge, the previously explained macro system is used to “translate and inject” the DSL program.
- In addition, a concept called the *DSL Block* is used, a special **area that can contain any arbitrary string** (here: a timetable)

```
func is_valid(train, start_station, destination_station, ticket_type, day):

    for time, station, valid_tickets := train.iterate():
        if station == start_station:
            if valid_tickets(destination_station, day, null).find(ticket_type):
                return "valid"
            else:
                return "invalid"


$<<timetable>>:
    8:25 "Exeter St. Davids"
        if day == "Saturday":
            ["Cheap", "Premium"]
        else:
            ["Premium"]
    10:20 "Salisbury"
        ["Cheap", "Premium"]
    11:15 "Woking"
        ["Cheap", "Premium"]
    11:49 "London Waterloo"
```

 ©2007/8 Markus Völter

Converge IV: The DSL implementation function

- The purpose of the DSL implementation function is to **somehow translate the DSL** text into a (converge) AST. (this function has the same name as the DSL block).
- A generic **utility function** (*CEI::dsl_parse*) builds a **parse tree** from any kind of textual syntax (using an Earley parser).
  - This is then passed to a translator that is specific to the DSL (see below)

```
func timetable(dsl_block, src_info):
    parse_tree := CEI::dsl_parse(dsl_block, src_info, ["ticket_type", "train"], [], GRAMMAR, "timetable")
    return Timetable_Translator.new().generate(parse_tree)
```

- It does need a **language spec** (the GRAMMAR), though…

 ©2007/8 Markus Völter

Converge V: The Grammar

- The grammar specifies the **concrete syntax** of the DSL.

```
GRAMMAR := CV_Parser::GRAMMAR + """
timetable ::= station ( "NEWLINE" station )*

station ::= "INT" ":" "INT" "STRING" ticket_check

ticket_check ::= "INDENT" expr "DEDENT"
ticket_check ::=
"""
```

- In order to do the transformation into a Converge AST, you have to write a **function for each of the syntax tokens**; you use the same name to associate the function with the token

Converge V: The Translator

- Note how the **translator uses the macro system** to assemble a “piece of AST” this is then compiled down to byte code.

```
class Timetable_Translator:
    func _t_timetable(node):
        stations := []
        i := 1
        while i < node.len():
            stations.append(self._preorder(node[i]))
            i += 2
        return CEI::ilist(stations)

    func _t_station(node):
        time := Strings::format("%s:%s", node[1].value, node[3].value)
        return [| [${CEI::lift(time)}, ${CEI::lift(node[4].value)}, ${self._preorder(node[5])}] |]

    func _t_ticket_check(node):
        if node.len() == 1:
            tickets_expr := [| &tickets := [ "Cheap" ] |]
            renames := [| pass |]
        else:
            expr := self._preorder(node[2])
            expr, renames := CEI::mk_hygienic_itree(expr, Set{"destination", "day", "date"})
            tickets_expr := [| &tickets := ${expr} |]
        return [|
            func (&destination, &day, &date := null):
                ${renames}
                ${tickets_expr}
                return &tickets
        |]
```

CONTENTS

- Why DSLs
- DSL Categorization
- External DSLs
  - Generative: Eclipse & Co
  - Interpreted: MetaEdit+
- Dynamic Internal DSLs
  - Runtime Metaprogramming: Ruby
- Compiled Internal DSLs
  - Compile Time Metaprogramming: Converge
  - **Functional Programming: Scala**
- Language Workbenches
  - Intentional Software Domain Workbench
- Summary & Further Reading

Scala: Syntax Extension via Closures & Operator Syntax

- Scala has **two important features** that allow you (to some extent) do define DSLs:
  - Automatic closure construction
  - Operator syntax
- **Methods that take one parameter** can be used as an infix operator. Methods without parameters can be used as postfix operators.

```
class MyBool(x: Boolean) {
  def and(that: MyBool): MyBool = if (x) that else this
  def or(that: MyBool): MyBool = if (x) this else that
  def negate: MyBool = new MyBool(!x)
}
def not(x: MyBool) = x negate; // semicolon required here
def xor(x: MyBool, y: MyBool) = (x or y) and not(x and y)
```

Scala examples taken with permission from the Scala tutorial.
More details at scala-lang.org

Scala II

- You can use **parameterless function names** as parameters of methods.
  - This is parameterless function name syntax:

```
(cond: => Boolean) // a function (name) that evaluates to Boolean
```

- Once such a method is called,
  - the **actual parameters are NOT evaluated!**
  - A **nullary function** is automatically defined (which encapsulates the computation of the parameter evaluation, aka call-by-name)
  - This function is only evaluated when its accessed in the function

```
object TargetTest1 extends Application {
  def whileLoop(cond: => Boolean)
               (body: => Unit): Unit =

   if (cond) {
     body
     whileLoop(cond)(body)
   }

   var i = 10
   whileLoop (i > 0) {
     Console.println(i)
     i = i 1
   }
}
```


©2007/8 Markus Völter

Scala III: A more complex example

- Using automatic closure construction and operator syntax, you can easily create **new syntactic forms**.
  - Note how **intermediate objects** are created to which you then subsequently apply an operator!

```
object TargetTest2 extends Application {
  def loop(body: => Unit): LoopUnlessCond =
    new LoopUnlessCond(body);
  private class LoopUnlessCond(body: => Unit) {
    def unless(cond: => Boolean): Unit = {
      body
      if (!cond) unless(cond);
    }
  }

  var i = 10;

  loop {
    Console.println("i = " + i)
    i = i 1
  } unless (i == 0)

}
```


©2007/8 Markus Völter

I could have talked about...

- **C++ Template Meta Programming**
  - Uses the template facility to write compile-time meta programs that are „interpreted by the compiler“ in order to generate executable (machine) code.
  - However, this is too awkward, and I don‘t really consider this DSLs
- **C/C++ makros**
  - They don‘t really define a type system or any other constraints, which makes using them as a DSL relatively error prone and cumbersome
- Other compile time meta programming facilities ...
  - Such as **Template Haskell**
  - ... but I don‘t know much about them ☺

 ©2007/8 Markus Völter

CONTENTS

- Why DSLs
- DSL Categorization
- External DSLs
  - Generative: Eclipse & Co
  - Interpreted: MetaEdit+
- Dynamic Internal DSLs
  - Runtime Metaprogramming: Ruby
- Compiled Internal DSLs
  - Compile Time Metaprogramming: Converge
  - Functional Programming: Scala
- **Language Workbenches**
  - Intentional Software Domain Workbench
- Summary & Further Reading

 ©2007/8 Markus Völter

Language Workbench

- **Martin Fowler's** definition of a language workbench:
  - Define DSLs which are **fully integrated with each other.**
  - The primary source is a **persistent abstract data structure.**
  - DSL == **schema**, **editor(s)**, and **generator(s)**.
  - Programs/models are manipulated w/ **projectional editor.**
  - A language workbench can persist **incomplete or contradictory information**.
- I would like to add:
  - DSLs can be integrated with (or directly support) additional services such as **debuggers**, **team development** (diff/merge) etc.
- The red stuff is **not widely available** today but coming along slowly.

http://www.martinfowler.com/articles/languageWorkbench.html

©2007/8 Markus Völter

CONTENTS

- Why DSLs
- DSL Categorization
- External DSLs
  - Generative: Eclipse & Co
  - Interpreted: MetaEdit+
- Dynamic Internal DSLs
  - Runtime Metaprogramming: Ruby
- Compiled Internal DSLs
  - Compile Time Metaprogramming: Converge
  - Functional Programming: Scala
- Language Workbenches
  - **Intentional Software Domain Workbench**
- Summary & Further Reading

©2007/8 Markus Völter

Intentional Software

- Intentional Software is building a **Domain Workbench**, which is a language workbench that uses DSLs very broadly**.**
- The name *Intentional* hints at the fact that the Domain Workbench allows developers and business users to **capture the *intent*** of a program uncluttered.
  - Intentional has the explicit goal of having domain experts (and not programmers!) use the DSLs.
- In order to do this, you need to **define notations to capture this intent** – thus, building languages (in a broad sense) is a core part of intentional software.
  - Of course the ultimate benefit is in using these languages!
- Conceptually, this is based on the **Intentional Programming research project** at Microsoft 1993-2002
  - In fact, the same person is behind both projects. Intentional Software was founded by Charles Simonyi.

© 2007/8 Markus Völter

Intentional II: The intentional tree and projections

- At the core of the domain workbench is a data structure called the **intentional tree**.
- Here is a piece of code (actually, it isn't really. See below)

```
return a = b / (c + 1);
```

- The **intentional tree representation** can be displayed something like this ➔
- This tree is not the result of parsing the source code above.
- Rather, the **tree is the master**, and the textual syntax is a **projection**.
  - Here is another projection:

$$\text{return a} = \frac{b}{c + 1};$$

- The syntax **need not be parsable** at all!

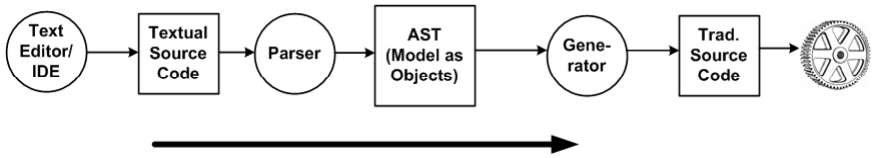
```
Return
(
   Assign
   (
      a,
      Div
      (
         b,
         Plus
         (
            c,
            1
         )
      )
   )
)
```

Intentional Examples taken from
the OOPSLA 2006 paper by **Simonyi/Christerson/Clifford**
http://intentsoft.com/technology/IS_OOPSLA_2006_paper.pdf

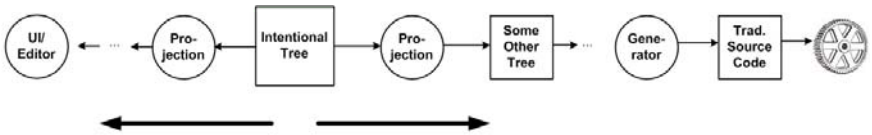
© 2007/8 Markus Völter

Intentional III: Projections

- Traditionally, the **AST (Abstract Syntax Tree) is the result** of a parsing process – ascii text is the master



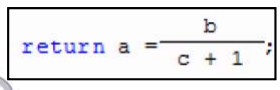
- In the Domain Workbench, the **Intentional Tree is the master**, and the editor, as well as the (potentially) generated code follows from **projections**.

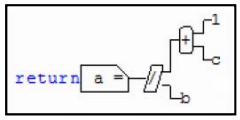


© 2007/8 Markus Völter

Intentional IV: The intentional tree II

- The intentional tree can thus be considered a **domain model** in the sense that it does not represent the (abstract) syntax of the input, but rather the “cleaned up” domain concepts.
- The domain model is used as code, called **domain code**, that conforms to a schema, the **domain schema**
  - The domain schema is conceptually equivalent to a schema in SQL or XML, a meta model for modeling approaches or a grammar for a programming language
  - It is however, important to note that the **domain model can be captured even if it is wrong, inconsistent or incomplete** wrt. to the schema. Correctness is in the eye of the user (i.e. projection).
- Since the „source code“ is a projection, **any number of projections** are possible.

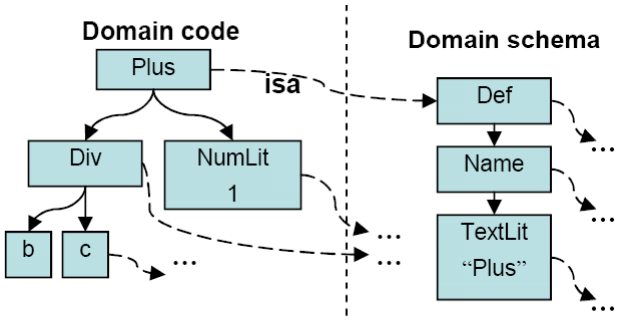




© 2007/8 Markus Völter

Intentional V: The intentional tree III

- Every node in the Intentional Tree has a **reference to its type**; the type nodes make up an intentional tree themselves (meta → meta → meta …)



© 2007/8 Markus Völter

Intentional VI: Projections II

- Since every "syntax" is just a projection, **syntactic forms, and languages, can be mixed** ("symbolic integration").
  a term coined by Martin Fowler in his Language Workbench article
- Example: Mix of C# and SQL:

```
private SqlDataReader GetAddresses(int ADDRESSID)
{
  int SELECT;
  SqlConnection Connection = new SqlConnection("server=localhost;uid=sa;database=db_cust");
  SqlCommand Command = new SqlCommand(
    SELECT
        *

      FROM
        ADDRESS AS addr

      WHERE
        addr.ADDRESS_ID = ADDRESSID
  , Connection);
  Connection.Open( );
  return Command.ExecuteReader(CommandBehavio
}
```

```
procedure double Sum3(double j, double k, double m)
{
   double result;
   result = j + (m + k)
```

	A	B	C	D	E
1	j	m	k	expected	result
2	1	15	2	18	18
3	3	15	4	22	22
4	4	5	6	14	15
5	.	.	.	.	.

```
   ;
   return result;
}
```

- Example: Test Data as Spreadsheet

© 2007/8 Markus Völter

CONTENTS

- Why DSLs
- DSL Categorization
- External DSLs
  - Generative: Eclipse & Co
  - Interpreted: MetaEdit+
- Dynamic Internal DSLs
  - Runtime Metaprogramming: Ruby
- Compiled Internal DSLs
  - Compile Time Metaprogramming: Converge
  - Functional Programming: Scala
- Language Workbenches
  - Intentional Software Domain Workbench
- **Summary & Further Reading**

 ©2007/8 Markus Völter

More Details

- **Eclipse & oAW**
  - eclipse.org & eclipse.org/gmt/oaw
- **Ruby DSLs:**
  - Obiefernandez.com
  - jayfields.com
  - weblog.jamisbuck.org
  - onestepback.org
- **Converge**
  - convergepl.org
- **MetaEdit**
  - metacase.com
- **Intentional Software**
  - intentsoft.com

Episode 16: MDSD Hands-on

Episode 52: Interview with Obie Fernandez

Episode 56: Interview with Laurence Tratt

 ©2007/8 Markus Völter

CONTENTS

- Why DSLs
- DSL Categorization
- External DSLs
  - Generative: Eclipse & Co
  - Interpreted: MetaEdit+
- Dynamic Internal DSLs
  - Runtime Metaprogramming: Ruby
- Compiled Internal DSLs
  - Compile Time Metaprogramming: Converge
  - Functional Programming: Scala
- Language Workbenches
  - Intentional Software Domain Workbench
- Summary & Further Reading

THE END.

QUESTIONS?

©2007/8 Markus Völter