Summary

Specifically

- Design and implementation of a domain-specific language for building web applications

Generally

- A systematic approach to designing domain-specific languages?

Outline

- A bit of history
- The domain-specific language engineering experiment
- Iteratively building a DSL for web applications
- Today: CRUD from DSL equivalent to class diagrams
- Next time: extension with a DSL for presentation layer

This is work in progress!
Acknowledgements

- Jos Warmer - Tutorial on Ordina Web DSLs
- William Cook - Discussions on DSLs
- Rob Schellhoorn - Eclipse
- Jonathan Joubert - Seam
- Sander Mak - WebDSL design experiments
- Danny van Groenewegen - Access control
- Martin Bravenboer - Discussions
Part I

A Bit of History
Photography (AKI; 1985–1987)
Computer Science (University of Amsterdam; 1988–1993)

Project: 3D environment (1989)
- programming in Turbo Pascal on Mac
- observations
  - lot of repetitive code for conceptually simple things
  - not captured by abstraction mechanisms of language
- goal: better reuse

Term paper: 'Hergebruik van Programmatuur' (1991)
- Reuse and economics
- Reuse and abstraction
  - Program skeletons, components, principles of decomposition, abstraction spiral
- Modulair abstraction
  - Module algebra, reuse by inheritance in Eiffel
- Linguistic abstraction
  - Application generators, languages with variable syntax, programming environments
Research program: better reuse by program generation

- generative programming and domain-specific languages
- needed: tools for language design and implementation

SDF2: language for describing (the syntax of) languages

- Redesign and reimplemention of SDF(1)
  - revolutionary new idea: scannerless parsing
  - redesign the syntax of the language to be more orthogonal
  - while keeping the language mostly backwards compatible
- Recurring idea
  - small core language makes back-end implementation simple
  - rich syntactic sugar makes programming attractive
  - desugaring transformations to map sugar to core
- Implemented in ASF+SDF
  - first-order functional language based on term rewriting
Term rewriting

- good basis for transformation
- lack of control over application of rules

Programmable rewriting strategies (1998)

- Control application of rewrite rules
  - determine rules to apply
  - strategy to apply them with
- Generic traversal
  - concise and generic definition of traversals over trees
- A core language for rewriting
  - small language with essential ingredients for building a transformation language
Program Transformation (Utrecht University; 1998-2006)

**Stratego/XT**
- Language and infrastructure for program transformation
- Syntax definition: SDF
- Transformation: Stratego
- Tools: XT

**Innovations**
- Dynamic rewrite rules
  - context-sensitive transformations
  - typechecking, data-flow transformations, partial evaluation
- Mixing languages
  - concrete syntax for meta-programming
  - embedded domain-specific languages
  - syntax of composite languages (AspectJ)

Many applications
Developing software for real users is hard

- deployment
- release management
- can we reduce that effort and get it right?

**Nix** software deployment system

- correct deployment, exact dependencies
- concurrent versions, maximal sharing
- safe garbage collection (uninstall)
- ...

**Nix buildfarm**

- continuous integration: automatically and continuously build and test software projects in a clean environment
- continuous release: automatically produce releases

* Nix was developed by Eelco Dolstra in the Jacquard project TraCE
Model-Driven Software Evolution (MoDSE)

- Integrate domain-specific languages in software process
- Requires
  - easy design and implementation of DSLs
  - support evolution (at multiple dimensions) of DSLs
- NWO/Jacquard project

Transformations for Abstractions (TFA)

- Support development of new linguistic abstractions
- Extend syntax and transformations of a language
- Research: independent extensibility of transformations
- NWO Open Competition
Part II

Domain-Specific Language Engineering
Designing Domain-Specific Languages Systematically

The Momentum

• Domain-specific languages
• Model-driven architecture
• Software factories
• Language workbenches
• Intentional programming
• ...
• Whatever you call it, it has never been so hot

The Challenge

• Designing and evolving domain-specific languages
• How do you come up with a new DSL?
• Is there a systematic approach?
(My) DSL Design Experience

SDF2

- Incremental improvements to an existing language
- Well developed theory, (some) local expertise
- Parser implemented in plain C

Stratego

- New language
- Based on six years experience with term rewriting in ASF+SDF
- Inspired by strategies in ELAN
- ATerms for term representation and garbage collection
- Theory on term rewriting not really helpful

Nix

- New language, inspired by lazy functional programming and
- Research languages for software build management
- Using ATerm library for term representation
- Hardly any theory on software deployment
Some Observations

Domain should be well understood (by someone)

• designing a DSL is not a good method for domain analysis

Domain knowledge is important

• DSL design may be a way to explore the domain

Basic technology should be available

• libraries, frameworks, development experience, code base

Abstraction gap

• considerable gain in abstraction should be possible

Common themes

• concise core language capturing the essence
• extended with syntactic sugar and desugaring transformations
An Experiment: WebDSLs

**Experiment**

- Take a new domain: web applications
- Develop a DSL (set of DSLs) for this domain
- Observe elements for a standard process
- (Repeat in the future for other domains)

**Experience with domain**

- Using web applications: extensive
- Implementation
  - HTML, CSS
  - Maintenance of several wiki-based sites (since 2000?)
  - Tweaking TWiki (Perl)
  - Few experiments with servlets
- In summary: minimal experience
Contributions of this tutorial

- An experience report
- Introduction to a particular (meta) technology (Stratego/XT) from an application perspective
- Guidelines; in particular, an emerging method for developing DSLs in a more systematic way

What you should not expect

- a comparison of techniques and tools for DSL definition
- a comparison of visual vs textual languages
- a comparison of web programming languages and technologies

(That is future work, though)
Part III

Domain Analysis
Scope: what types of web applications?

- Content-management system
- Wiki-like
  - editable via browser
- Rich domain model
  - instead of generic text
  - objects in domain classes
  - generic queries, aggregations, etc.
- Example: web site of a research group
The Software Engineering Research Group

Mission

Software engineering is concerned with methods and techniques for building high quality software systems. This not only includes software construction, but also requirements analysis, design, system integration, testing, deployment, and making changes to software systems after their first release.

The mission of the Delft Software Engineering Research Group is

1. to develop a deep understanding of how people build and evolve software systems;
2. to develop novel methods, techniques and tools that advance the way in which software is built and adjusted; and
3. to offer students an education that prepares them to take a leading role in complex software development projects.

Research at the Delft Software Engineering Research Group is centered around two themes, software evolution and embedded software, which are studied separately as well as in combination in two laboratories:

- The Software Evolution Research Laboratory (SWERL), and
- The Embedded Software Laboratory (ESL)
2. to develop novel methods, techniques and tools that advance the way in which software is built and adjusted; and
3. to offer students an education that prepares them to take a leading role in complex software development projects.

Research at the Delft Software Engineering Research Group is centered around two themes, software evolution and embedded software, which are studied separately as well as in combination in two laboratories:

- The Software Evolution Research Laboratory (SWERL), and
- The Embedded Software Laboratory (ESL)

**News**

*2007-04-25*

Peter Kluit was elected Computer Science Teacher of the Year 2006/2007. The election was organised by the study society Christian Huygens [http://ch.tudelft.nl/index.php](http://ch.tudelft.nl/index.php).

*2007-04-20*


*2007-04-18*

Paper: *Understanding Execution Traces Using Massive Sequence and Circular Bundle Views* by Bas Cornelissen et al. accepted by *International Conference on Program Comprehension*. The paper proposes to gain an understanding of software behavior by means of a scalable trace visualization technique. See the [ExTravis homepage](http://www.extravis.org) for more information.
software engineering groups in the project.

2006-10-15

Eelco Visser, formerly at Utrecht University, has been appointed associate professor. He will be continuing his work on program transformation and generation and lead the McDSE and TFA projects. Martin Brevenboer joins him, for the time being as guest PhD student from Utrecht University.

2006-10-01

Andy Zaidman appointed postdoc in the Reconstructor Project.

Recent Publications

2007


- B. Graaf and A. van Deursen (2007). Visualization of Domain-Specific Modelling Languages
Eelco Visser

News
Looking for postdocs and PhD students in the following project

- Model-Driven Software Evolution (Jacquard 2006) - 2 postdocs + 2 PhD students

Coordinates

- Associate professor
- Software Engineering Research Group
- Department of Software Technology
- Electrical Engineering, Mathematics and Computer Science (EWI)
- Delft University of Technology
- Delft, The Netherlands (CEST/GET)

- Email: visser@acm.org
- http://www.et.ew.tudelft.nl/~eelco
- http://www.eelcovisser.net
- Blog

Recent Papers

- Model-driven software evolution: A research agenda (MoDSE’07)
- Declarative, Formal, and Extensible Syntax Definition for AspectJ (OOPSLA’06)
- Stratego/XT 0.16. Components for Transformation Systems (PEPM’06)
- Stratego/XT Manual (documentation)
- Transformations for Abstractions (Keynote SCAM’05)
- Generalized Type-Based Disambiguation of Meta Programs with Concrete Object Syntax (GPCE’05)
TUD-SERG Technical Report Series

Our technical report series, started in 2005, contains preprints of our Scientific Publications. They are listed in reverse chronological order.

2007

<table>
<thead>
<tr>
<th>Report ID</th>
<th>Author(s)</th>
<th>Title</th>
<th>Appeared as</th>
</tr>
</thead>
<tbody>
<tr>
<td>TUD-SERG-2007-011</td>
<td>Alex Feldman, Greg Provan, Arjan van Gemund</td>
<td>On the performance of SAFARI algorithms</td>
<td></td>
</tr>
<tr>
<td>TUD-SERG-2007-010</td>
<td>Marius Marin, Leon Moonen, Arie van Deursen</td>
<td>Documenting Typical Crosscutting Concerns</td>
<td></td>
</tr>
<tr>
<td>TUD-SERG-2007-005</td>
<td>Marius Marin, Leon Moonen, Arie van Deursen</td>
<td>Some Remarks on Significant Features of Our Research Program</td>
<td></td>
</tr>
</tbody>
</table>
Research Colloquium

The SERG group meets (at least) once in the two weeks to learn about and exchange ideas on recent research carried out by the group's researchers (Faculty members, Postdocs, PhD students). Occasionally researchers from other organizations are invited to present their latest work.

Time and Place

Thursday, 11:00 - 12:00
Room: 9.130 (EWI)

Upcoming Presentations

<table>
<thead>
<tr>
<th>Date</th>
<th>Speaker</th>
<th>Title</th>
<th>Extras</th>
</tr>
</thead>
<tbody>
<tr>
<td>24-05-2007</td>
<td>Eelco Visser</td>
<td>Domain-Specific Language Engineering</td>
<td>MoDSE Colloquium, 10:30-12:30, abstract</td>
</tr>
<tr>
<td>07-06-2007</td>
<td>Andy Zaidman</td>
<td>On How Developers Test Open Source Software Systems</td>
<td>12:45, abstract</td>
</tr>
<tr>
<td>14-06-2007</td>
<td>tba</td>
<td>tba</td>
<td>MoDSE Colloquium in Bordevijkzaal (19.130) 10:30-12:30</td>
</tr>
<tr>
<td>20-06-2007</td>
<td>tba</td>
<td>MoDSE workshop</td>
<td>all day in Snoijderszaal</td>
</tr>
</tbody>
</table>

See Also

* Past Presentations
## In Progress

<table>
<thead>
<tr>
<th>Name</th>
<th>Start</th>
<th>End</th>
<th>Supervisor(s)</th>
<th>Site</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zooger Lubson</td>
<td>2007</td>
<td>2008</td>
<td>Andy Zaidman</td>
<td>Software Improvement Group, Reconstructor Project</td>
<td>Co-evolution of test and production code</td>
</tr>
<tr>
<td>Jippe Holwerda</td>
<td>2007</td>
<td>2008</td>
<td>Leon Moonen</td>
<td>Compuware</td>
<td>Semi-automatic MDD Remodularization in OptimalJ</td>
</tr>
<tr>
<td>Denny Groenewegen</td>
<td>2006</td>
<td>2007</td>
<td>Elco Visser</td>
<td>TUD</td>
<td>Web-application security</td>
</tr>
<tr>
<td>Jonathan Joubert</td>
<td>2006</td>
<td>2007</td>
<td>Elco Visser</td>
<td>Finalist</td>
<td>Model-driven online development, deployment and maintenance of web applications</td>
</tr>
<tr>
<td>Gerardo Geest</td>
<td>2006</td>
<td>2007</td>
<td>Elco Visser</td>
<td>Avanade</td>
<td>Evolution of DSLs with an application to web services</td>
</tr>
<tr>
<td>Mulo Emmanuel</td>
<td>2007</td>
<td>2007</td>
<td>Andy Zaidman, Arie van Deursen</td>
<td>Philips Medical Systems</td>
<td>Architectural design for testability</td>
</tr>
<tr>
<td>Xia Chao</td>
<td>2007</td>
<td>2008</td>
<td>Gerr Gross</td>
<td>Imec, Leuven</td>
<td>Development of a new task scheduling component for multimedia platforms</td>
</tr>
</tbody>
</table>
IntraSE - An Intranet for the Software Engineering Research Group

This website aims at sharing internal information of the TU Delft Software Engineering Research Group.

Contents of this intranet site:

- **Serg Meetings**
- **Action Points**, including
  - SERG Web Site
  - Calendar
  - Research Colloquium
  - Technical Report Series
- **Howtos**
- Project Codes
- Research Output
- Teaching Howto's
- Coffee-Machine Maintenance Roster
- News-Events-Visitors Policy

Some useful links for (new) TWiki users

- The [TWiki](#) homepage which describes the idea behind wiki and this particular instance: TWiki.
- [Welcome Guest](#) and [Taste of Twiki](#) are two tutorials.
- A [sandbox](#) where you can safely play around to test things without messing up anything (NB: everything on this wiki is stored under a revision management system so there no need for worries anyway).
(screenshot of wiki edit page)
SERG: Domain Model

- Text
- News
- Conferences
- Publications
- Researchers
- Homepages
- Photos
- Research projects
- Software
- Technical reports
- Courses
- Students
- Thesis projects (workflow!)
- Travel (accounting)
- Meetings
- ...

...
Pure Domain Objects

Object-Oriented Utopia

- 'Objects are models of real-world concepts'
- Bertrand Meyer - Object-Oriented Programming (Eiffel)

Object-Oriented Practice

- Most objects are not domain objects
- Large part of software concerned with plumbing
- Complicated 'patterns'
- Example: web applications
  - a bit of domain (business) objects
  - a lot of servlets, session beans, ...

- Can we regain pure objects?
Domain Analysis: Deductive vs Inductive

**Deductive (top-down)**
- Analyze the problem domain
- Define abstract requirements
- Useful/necessary for new domains
- Risk: may be difficult to implement
- Example: QVT (although not strictly a new domain)

**Inductive (bottom-up)**
- Look at existing applications / systems in the domain
- Find common programming patterns
- Define abstractions that capture these patterns
- Risk: abstractions too close to existing practice
- Solution: iterative abstraction
- ’Don’t look for domain abstractions, let the abstractions find you’
Technology

- what technology is available?
- how is this technology typically used?
- one scenario is to come up with a complete new application domain for which the technology still has to be invented; in that case it is probably not a good idea to start with the design of a DSL; DSLs are useful where there is already a good idea for implementing applications in a domain; a DSL should allow to make it considerably easier to make applications in this domain
- thousands of virtual machines; each combination of languages, libraries, frameworks defines a virtual machine to target in software development; each enterprise system/application we develop may require a different combination; similar to situation in embedded systems, where the peculiarities of different hardware architectures have to be dealt with; if we’re developing the essence of an enterprise system, can we abstract from the details of the different virtual machines? ORM solutions such as Hibernate already help in abstracting from the peculiarities of database management systems; development of abstractions is driven by the technology they abstract from
- it would be nice to design an abstract language from a set of requirements and then think about ways to implement the language; this is an approach that is taken (examples: QVT, ...)
- however, the approach that I want to advocate in this paper is the incremental introduction of abstractions (inductively) that allow one to capture a set of common programming patterns in software development for a particular domain. This should allow a quick turn-around time for such abstractions. Since they are based on concrete programming patterns, there will be no question as to how to implement the constructs of the language to
- Considering the collection of patterns will hopefully lead to a deeper insight into how to make even better abstractions for the application domain.
My Technology Stack

I chose / stumbled upon the Java route

- Java: programming
- Servlets: handling web requests
- JSP: simple presentation layer
- SQL: database management
- JDBC: database connection
- Hibernate: object-relational mapping
- JSF: better presentation layer
- EJB3:
  - Seam: integration framework

Many alternatives available

- PHP, Ruby/Rails, .Net, ...
Granularity and Expressivity

Scale of granularity and expressivity

- Maximal expressivity/reuse, minimal flexibility
  - few constructs from which a complete application is 'generated'
  - coarse grained language provides good reuse
  - large chunk of code can be reused at once
  - provides (too) little flexibility

- Maximal flexibility, minimal expressivity/reuse
  - 1 construct corresponds to 1 GPL construct
  - the language now mimicks the GPL and no productivity gains are to be expected

Find a balance between these extremes
Part IV

Capturing Common Programming Patterns
Architecture of an EJB/Seam Web Application

Tiers

- Presentation layer
  - Java Server Faces (JSF)
- Session beans
  - Java objects that connect presentation and domain objects
- Domain objects
  - store persistent data
  - correspond to 'real-world' concepts

And

- A bit of configuration (XML)
Java persistence

- Java 5 annotations for declaration of object-relational mapping
- Vendor independent interface
- Hibernate provides implementation and special purpose annotations
- Entity class corresponds to table in database

Class annotated with @Entity, empty constructor

```java
@Entity
public class Publication {

    public Publication () { }

    // properties

}
```
Entities have an identifier as primary key

```java
@Id @GeneratedValue
private Long id;

public Long getId() {
    return id;
}

private void setId(Long id) {
    this.id = id;
}
```
Properties represent data (columns in database)

private String title;

public String getTitle() {
    return title;
}

public void setTitle(String title) {
    this.title = title;
}
Properties referring to other entities require annotations

@ManyToOne
@JoinColumn(name = "PublicationAuthor")
@org.hibernate.annotations.Cascade( {
    org.hibernate.annotations.CascadeType.PERSIST,
    org.hibernate.annotations.CascadeType.SAVE_UPDATE,
    org.hibernate.annotations.CascadeType.MERGE
})

private Person author = new Person();

public Person getAuthor() {
    return author;
}

public void setAuthor(Person author) {
    this.author = author;
}
A Domain Model DSL

The essence of an entity class is simple

- class name
- list of properties, i.e., (name, type) pairs

Example

Publication {
  title : String
  author : Person
  year : Int
  abstract : String
  pdf : String
}

Person {
  fullname : String
  email : String
  homepage : String
}
Implementing a DSL

- Definition of concrete syntax
- Parser
- Definition of abstract syntax
- Transformation of models to Java code
Implementing a DSL

- Definition of concrete syntax
  - using the syntax definition formalism SDF
- Parser

- Definition of abstract syntax

- Transformation of models to Java code
Implementing a DSL

- Definition of concrete syntax
  - using the syntax definition formalism SDF
- Parser
  - generate from syntax definition
- Definition of abstract syntax

- Transformation of models to Java code
Implementing a DSL

- Definition of concrete syntax
  - using the syntax definition formalism SDF
- Parser
  - generate from syntax definition
- Definition of abstract syntax
  - generate from syntax definition
- Transformation of models to Java code
Implementing a DSL

- Definition of concrete syntax
  - using the syntax definition formalism SDF
- Parser
  - generate from syntax definition
- Definition of abstract syntax
  - generate from syntax definition
- Transformation of models to Java code
  - implement using term rewrite rules
Implementing a DSL

- Definition of concrete syntax
  - using the syntax definition formalism SDF
- Parser
  - generate from syntax definition
- Definition of abstract syntax
  - generate from syntax definition
- Transformation of models to Java code
  - implement using term rewrite rules
  - use concrete syntax of target language to make rules readable
module DomainModel
exports

lexical syntax
[a-zA-Z][a-zA-Z0-9\_]* -> Id
[0-9]+ -> Int
"\"" ~[\"\n\]* "\"" -> String
[\t\n\r] -> LAYOUT
"/\"" ~[\n\r]* [\n\r] -> LAYOUT

context-free syntax
Entity -> Definition
Id "{" Property* "}" -> Entity {cons("Entity")}
Id ":" Sort -> Property {cons("Property")}
Id -> Sort {cons("SimpleSort")}

Generate abstract syntax definition from syntax definition

signature

constructors

SimpleSort : Id -> Sort
Property : Id * Sort -> Property
Entity : Id * List(Property) -> Entity

: Entity -> Definition
: String -> Id

sdf2rtg -i DomainModel.def -m DomainModel -o DomainModel.rtg
rtg2sig -i DomainModel.rtg -o DomainModel.str --module DomainModel
Generate parser from syntax definition

sdf2table -i DomainModel.def -o DomainModel.tbl -m DomainModel

Parsing gives abstract syntax terms

```java
Person {
    fullname : String
    email : String
    homepage : String
}
Entity("Person",
    [ Property("fullname", SimpleSort("String"))
    , Property("email", SimpleSort("String"))
    , Property("homepage", SimpleSort("String"))
    ]
)
```

glri -p DomainModel.tbl -i publication.dom | pp-aterm
@Entity
public class Publication {
    public Publication () { }
}

abstract syntax

ClassDec(
    ClassDecHead(
        [MarkerAnno(TypeName(Id("Entity"))), Public()]
        , Id("Publication")
        , None(), None(), None()),
    ClassBody(
        [ConstrDec(
            ConstrDecHead([Public()]),None(),Id("Publication"),[],None()),
            ConstrBody(None(), [])])
)
entity-to-class :
    Entity(x, prop*) ->
    ClassDec(
        ClassDecHead(
            [MarkerAnno(TypeName(Id("Entity"))), Public()],
            Id(x),
            None(), None(), None()),
        ClassBody(
            [ConstrDec(
                ConstrDecHead([Public()], None(), Id(x), [], None()),
                ConstrBody(None(), []))]
        ))
Use concrete syntax of Java in transformation rules

entity-to-class :
   Entity(x_Class, prop*) ->
   |
   @Entity
   public class x_Class {
      public x_Class () { } 
   }

Properties

• code fragment is parsed (syntax check)
• transformation produces term representation, not text
• generated code can easily be further transformed
Domain Model: Code Generation for Entity

domain-to-class:
Entity(x_Class, prop*) -> |

@E企ity public class x_Class {
    public x_Class () { }

    @Id @GeneratedValue private Long id;

    public Long getId() {
        return id;
    }
    private void setId(Long id) {
        this.id = id;
    }

    ~*cbd*
}

where cbd*: = <mapconcat(property-to-gettersetter)> prop*
property-to-gettersetter :
    Property(x_prop, s) -> class-body-dec*[
        private t x_prop;

        public t x_get() {
            return title;
        }
        public void x_set(t x) {
            this.x = x;
        }
    ]

where t := <builtin-java-type> s
    ; x_get := <property-getter> x_prop
    ; x_set := <property-setter> x_prop

builtin-java-type :
    SimpleSort("String") -> type[ String ]
property-to-property-code(|x_Class) :
  Property(x_prop, s) -> class-body-dec* |[
    @ManyToOne
    @org.hibernate.annotations.Cascade({
      org.hibernate.annotations.CascadeType.PERSIST,  
      org.hibernate.annotations.CascadeType.SAVE_UPDATE,  
      org.hibernate.annotations.CascadeType.MERGE  
    })
    private t x_prop;

    public t x_get() { return x_prop; }

    public void x_set(t x_prop) { this.x_prop = x_prop; }
  ]|

where t := <defined-java-type> s
  ; x_Prop := <capitalize-string> x_prop
  ; x_get := <property-getter> x_prop
  ; x_set := <property-setter> x_prop
  ; columnname := <concat-strings>[[x_Class, x_Prop]]
Propagate declared entities

\[
\text{declare-entity} = \\
\text{?Entity}(x_{\text{Class}}, \text{prop}*) \\
; \text{rules(} \\
\text{\quad defined-java-type :} \\
\quad \text{SimpleSort}(x_{\text{Class}}) \rightarrow \text{type}\|[x_{\text{Class}}]\| \\
\text{)}
\]

Dynamic rewrite rules

- add new rewrite rules at run-time
- rules inherit variable bindings from their definition context
- propagate context-sensitive information
- e.g., the Java type for a declared entity
Composing a code generator

domain-model-to-java =
   io-wrap(
      alltd(declare-entity)
      ; collect(entity-to-class)
   )

What it does

- read input
- define dynamic rules for all declared entities
- generate java code for each entity declaration

Ignoring some details

- Invoking parser (this definition assumes AST as input)
- Pretty-printing output (this definition produces AST as output)
- Java classes should be written to separate files
Part V

Capturing More Programming Patterns
Person {
  fullname : String
  email : String
  homepage : String
  photo : String
  address : Address
  user : User
}

Address {
  street : String
  city : String
  phone : String
}

User {
  username : String
  password : String
  person : Person
}
<h1> <h:outputText value="#{viewPerson.person.fullname}"/> </h1>
<table>
<tr> <td> <h:outputText value="Fullname"/> </td> <td> <h:outputText value="#{viewPerson.person.fullname}"/> </td> </tr>
<tr> <td> <h:outputText value="Address"/> </td> <td> </td> </tr>
<tr> <td> <h:outputText value="Street"/> </td> <td> <h:outputText value="#{viewPerson.person.address.street}"/> </td> </tr>
<tr> <td> <h:outputText value="User"/> </td> <td>
<s:link view="/viewUser.xhtml"
value="#{viewPerson.person.user.name}" propagation="none">
  <f:param name="user" value="#{viewPerson.person.user.id}"/>
</s:link>
</td> </tr>
</table>
Seam component can be approached directly from JSF page

```java
@Stateful // can keep state between requests
@Name("viewPerson") // component name
public class ViewPersonBean
    implements ViewPersonBeanInterface
{
    ...

    @Destroy @Remove // required for stateful beans
    public void destroy() {
    }
```
Necessary services are obtained by *injection*

```java
@Logger
private Log log;
// generating log messages

@PersistenceContext(type = EXTENDED)
private EntityManager em;
// interface to the database

@In
private FacesMessages facesMessages;
// generating screen messages
```

No need to pass parameters or use factories
Domain object made available to JSF via property

```java
private Person person;
public void setPerson(Person person) { this.person = person; }
public Person getPerson() { return person; }
```

Identity of object is passed to page via request parameter

```java
@RequestParameter("person")
private Long personId;
```

Initialization of object based on parameter identifier

```java
@Create @Begin
public void initialize() {
    if (personId == null) {
        person = new Person();
    } else {
        person = em.find(Person.class, personId);
    }
}
```
Generating JSF Pages: Entities

entity-to-xhtml-viewEntity :
Entity(x_Class, x_super, props) ->

<h1>
  <h:outputText value="<%=x_Class%> #{<%=x_class%>.name}"/>
</h1>
<table>
  <%= rows :::* %>
</table>
<%
  where x_class := <decapitalize-string> x_Class
  ; rows := <map(row-in-view-form(|x_class)> props

row-in-view-form(|x_class) :
prop@Property(x_prop, _, _, _) ->
<%
  <h:outputText value="<%=x_prop%>"/> <%= input %>
<%
  where input := <property-to-view-component(|x_class)> prop
Output of property

property-to-view-component(|x_class|) :
    Property(x_prop, SimpleSort("String")) ->
    <%>
        <h:outputText value="#{<%=x_class%>.<%=x_prop%}>"/>
    <%

Input of property

property-to-edit-component(|x_component|) :
    Property(x_prop, SimpleSort(s0)) ->
    <%>
        <h:inputText value="#{<%=x_component%>.<%=x_prop%}>"/>
    <%
Replacing names in boilerplate code

entity-to-session-bean :
    Entity(x_Class, prop*) -> [ [ @Stateful
                                      @Name("~viewX")
                          public class x_ViewBean implements x_ViewBeanInterface
                        {
                          ...
                          @Destroy @Remove
                          public void destroy() { }
                        }
                    ]]
    where x_ViewBean := ...
          ; x_ViewBeanInterface := ...
deriving interfaces

create-local-interface(|x_Interface) :
class ->
[[@Local
   public interface x_Interface { ~*methodsdecs
   }
]
where methodsdecs := <extract-method-signatures> class

extract-method-signatures =
collect(method-dec-to-abstract-method-dec)

method-dec-to-abstract-method-dec :
   MethodDecHead(mods, x , t, x_method, args, y) ->
   AbstractMethodDec(mods, x, t, x_method, args, y)
where <fetch(?Public())> mods
Part VI

Refining Programming Patterns
Special types allow to generate refined behaviour

Person {
    fullname : String
    email : Email
    homepage : URL
    photo : Image
    address : Address
    user : User
}

User {
    username : String
    password : Secret
    person : Person
}

property-to-edit-component(|x_component|):
    Property(x_prop, SimpleSort("Text")) ->
    %><h:inputTextarea value="#{<%=x_component%>.<%=x_prop%>}"/></>

property-to-edit-component(|x_component|):
    Property(x_prop, SimpleSort("Secret")) ->
    %><h:inputSecret value="#{<%=x_component%>.<%=x_prop%>}"/></>
Collections

Publication {
    title : String
    authors : List<Person>
    year : Int
    pubabstract : Text
    projects : Set<ResearchProject>
    pdf : URL
}

**Generate Many-to-Many Associations**

```java
property-to-property-code(|x_Class) :
    Property(x_prop, GenericSort("List", [y])) -> |
        @ManyToMany
        @org.hibernate.annotations.Cascade({
            org.hibernate.annotations.CascadeType.PERSIST,
            org.hibernate.annotations.CascadeType.SAVE_UPDATE,
            org.hibernate.annotations.CascadeType.MERGE
        })
        private List<t> x_prop = new LinkedList<t>();
```
Reining Associations

Value Types

- title :: String

Composite Associations

- address <> Address

Reference Associations

- authors -> List<Person>

Publication {
  title :: String
  authors -> List<Person>
  year :: Int
  pubabstract :: Text
  projects -> Set<ResearchProject>
  pdf :: URL
}

Person {
  fullname :: String
  email :: Email
  homepage :: URL
  photo :: Image
  address <> Address
  user -> User
}
Generating JSF Pages: Unfolding Entities

```java
Person {
    fullname :: String
    email :: Email
    homepage :: URL
    photo :: Image
    address <> Address
    user -> User
}

Address {
    street :: String
    city :: String
}

User {
    username :: String
    password :: Secret
    person -> Person
}
```
Show

```java
row-in-edit-form(x_component) :
    Property(x_prop, Comp(), s1@SimpleSort(s2), _) ->
    %>
    <h:outputText value="<%=x_prop%>"/>
    <%= row* ::%>
<% 
where <defined-java-type> s1
    ; prop* := <properties> s2
    ; x_sub_comp := <concat-strings>[x_component,".",x_prop]
    ; row* := <edit-form-rows(|x_sub_compt)> prop*
```
Summary

How far did we get?

- language for domain models
- with some refinements to support sophisticated crud operations
- generate entity classes, session beans, and JSF pages
- coarse grained language: templates are fairly big (lots of reuse), but also inflexible

Techniques

- declarative syntax definition
- rewrite rules with concrete syntax
- strategies (a bit)

What’s next?

- Scrap your boilertemplate: develop a core language
- Not all abstractions can be generative: user templates
- More sugar, please! unlimited expressive power with model-to-model transformations
- Unfinished business: outlook on more stuff to do