Domain-Specific Language Engineering

A Case Study in Agile DSL Development

Eelco Visser

Software Engineering Research Group
Delft University of Technology
Netherlands

July 2, 2007
GTTSE'07 Summerschool

Domain-Specific Language Engineering

Specifically

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

Generally

- A systematic approach to designing domain-specific languages?

Outline

- 0: The domain-specific language engineering experiment
- 1: Capturing programming patterns
- 2: Scrap your boilertemplate
- 3: More Sugar, please!

Part I

Domain-Specific Language Engineering (Introduction)

Domain-Specific Languages: The Momentum

Many approaches

- Domain-specific languages
- Model-driven architecture
- Software factories
- Language workbenches
- Intentional programming

One goal

- Programming at higher-level of abstraction
- by capturing domain knowledge in language + generator
- Reduce effort of software development *and* maintenance
- by an order of magnitude

Terminology

Domain

- specialized area of software development
- technical domain (e.g. database management)
- application / business domain (e.g. insurance)

Domain-specific language (DSL)

- a language: a set of well-formed sentences
- concrete syntax may be textual or visual
- has an abstract syntax
- domain-specific: special features / assumptions for domain

Model

- DSL 'program'

Generator

- translates models to implementations in a general-purpose language (GPL)

Domain-Specific Languages: The Challenge

Design and implementation

- Designing domain-specific languages *systematically*
- How do you come up with a new DSL?
- Is there a systematic approach?
- How to keep the implementation small and maintainable?

Evolution

- Keep DSL in synch with technology, domain, requirements
- How to make generators portable?
- How to migrate models when DSL is adapted?

Project: Model-Driven Software Evolution

- Domain: enterprise software

My DSL Design Experience

SDF2: syntax definition

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

Stratego: program transformation

- 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: software deployment

- 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
- Developed by Eelco Dolstra

Some Observations

Domain should be well understood (by someone)

- designing a DSL not a good method for exploring new domain
- purpose of DSL is to make development more productive

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 essence of domain
- 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 webapplications: extensive
- Implementation
  - HTML, CSS
  - Maintenance of several wiki-based sites (since 2000?)
  - Tweaking TWiki (Perl)
  - Few experiments with servlets
- In summary: minimal experience

Fixing Some Variables

Contributions of this tutorial

- Experience report
- Introduction to Stratego/XT from an application perspective
- Ideas for systematic DSL development

You should not expect comparisons of

- Techniques and tools for DSL definition
- Visual vs textual languages
- Web programming languages and technologies

Discussions about these topics welcome (off-line)

Part II

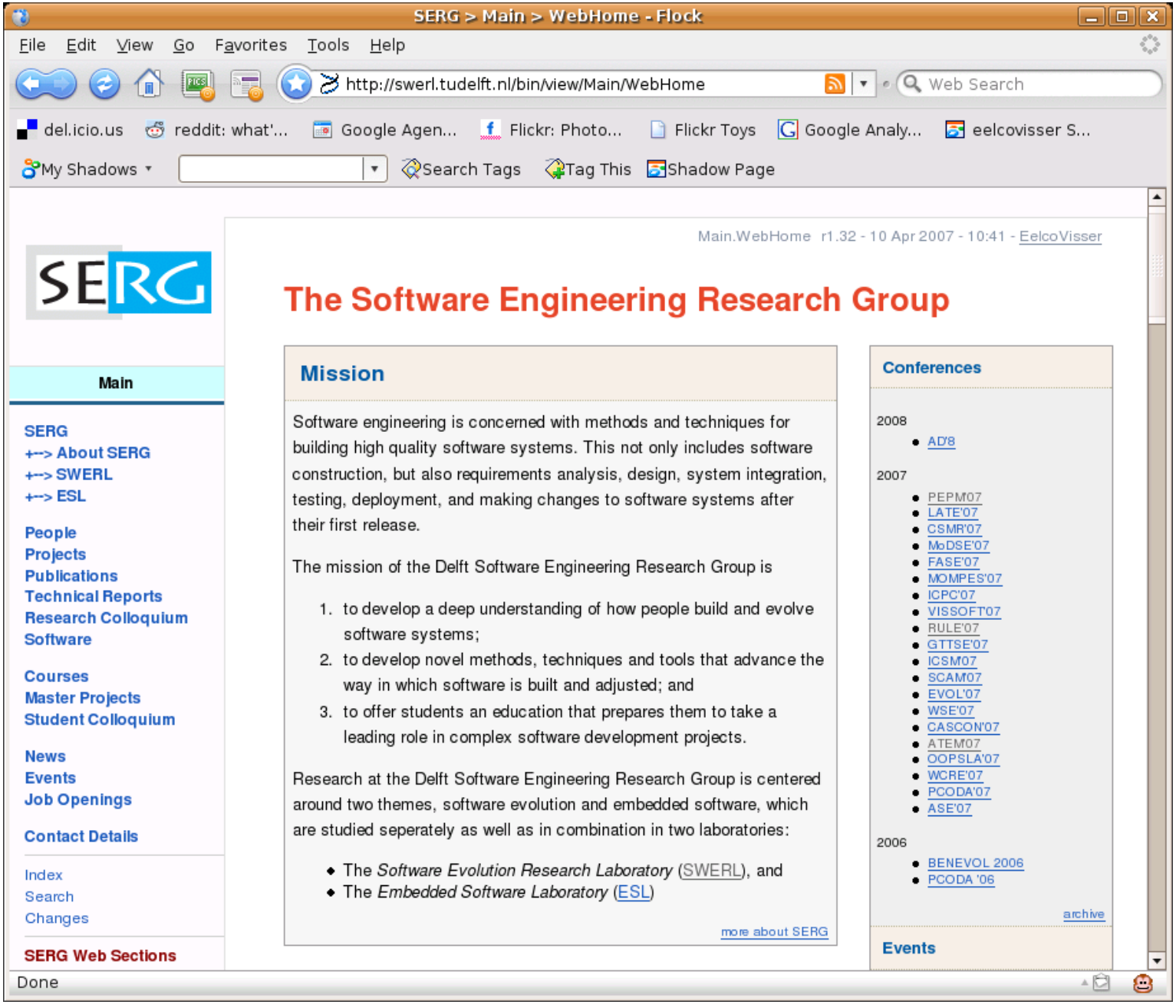
Domain Analysis

Web Applications

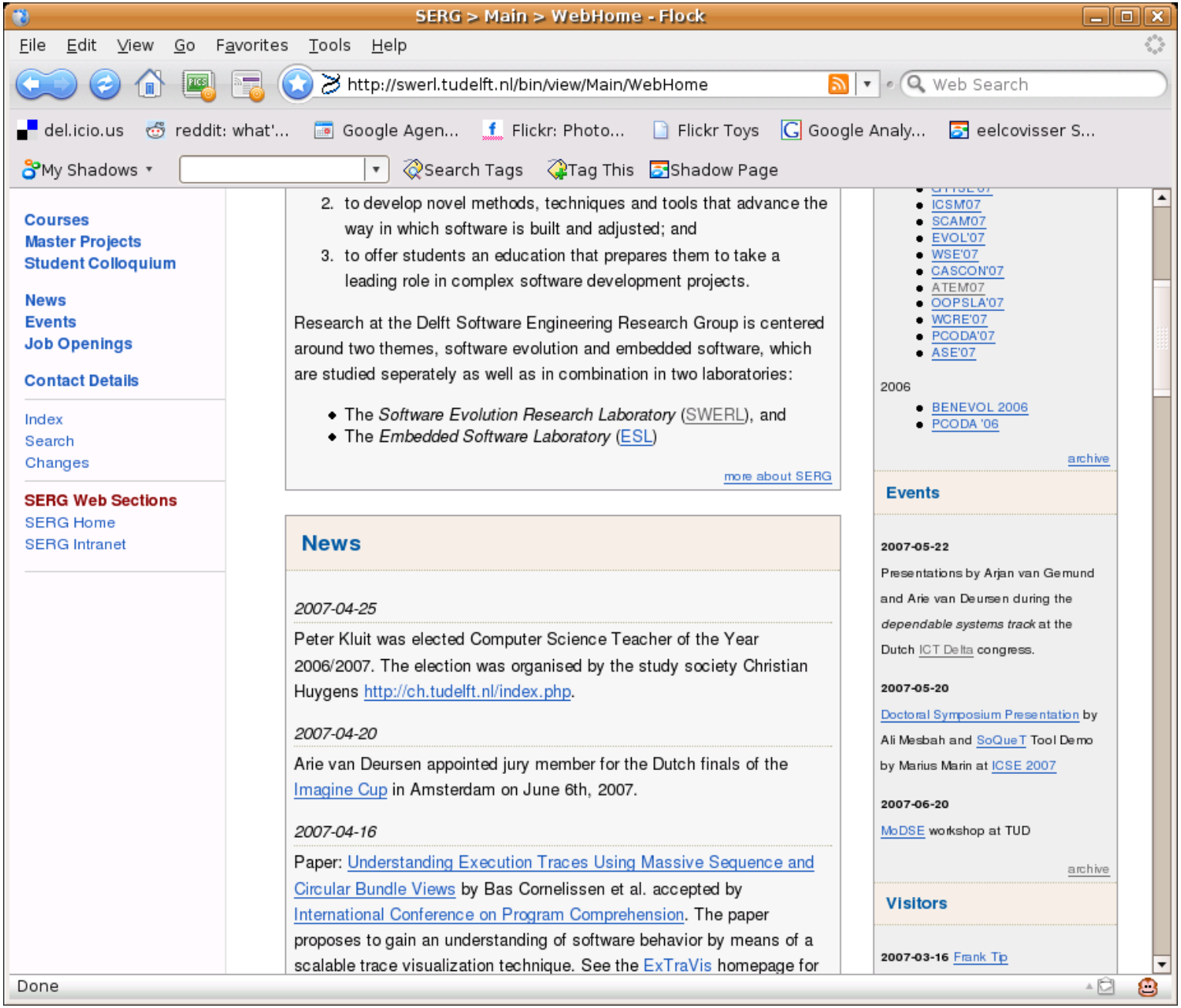
Scope: what types of web applications?

- 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

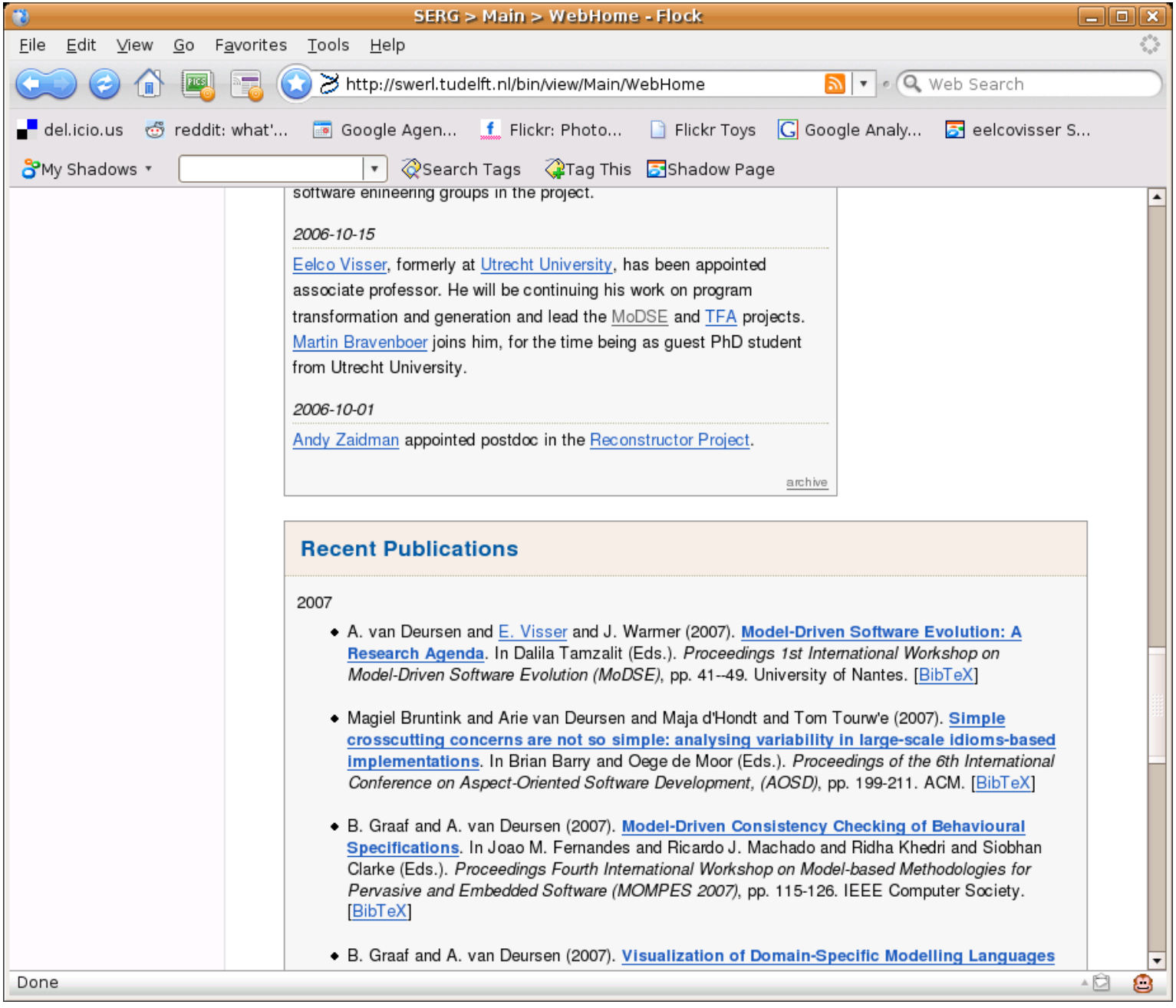
SERG: Text & Links



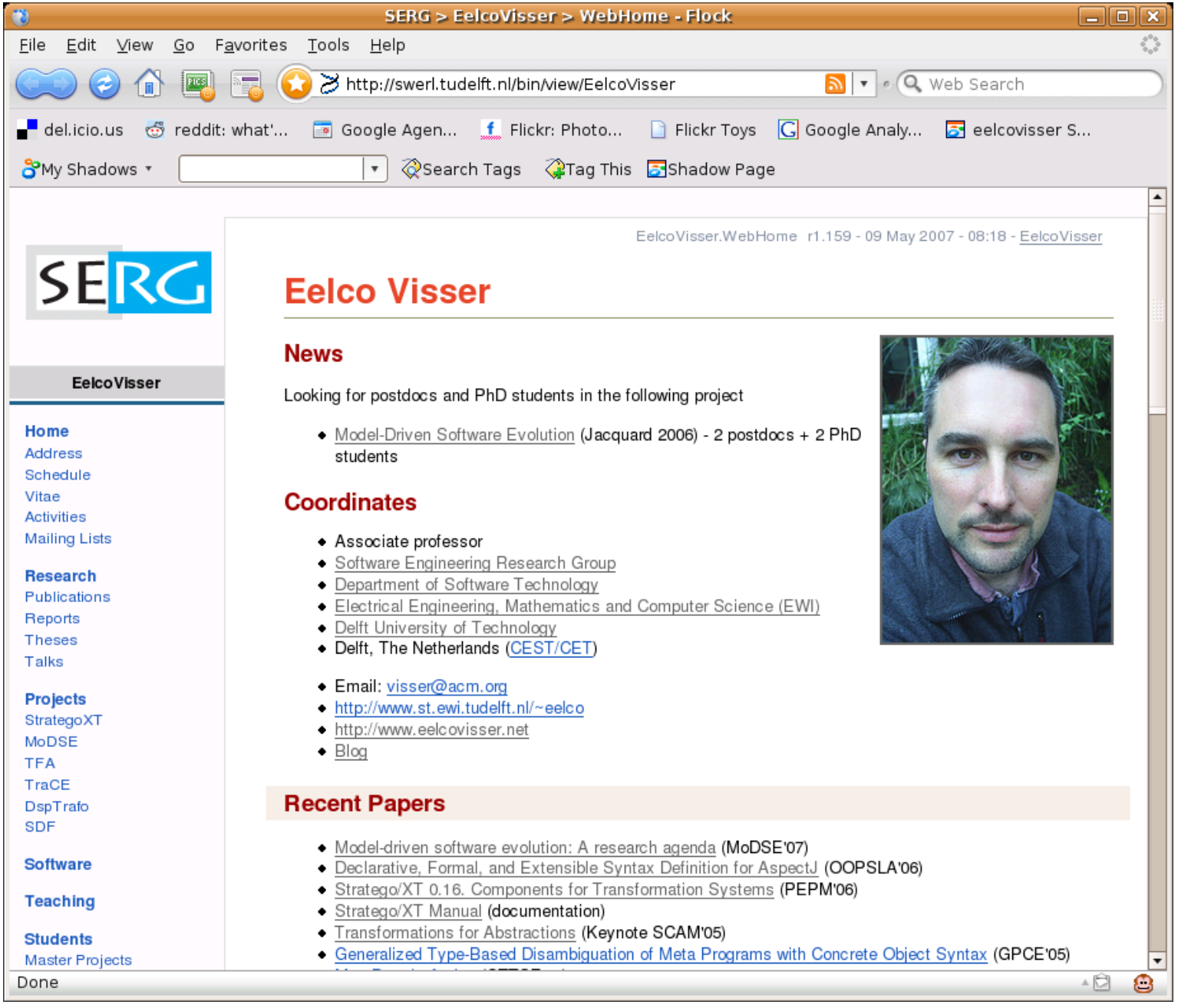
SERG: News Items



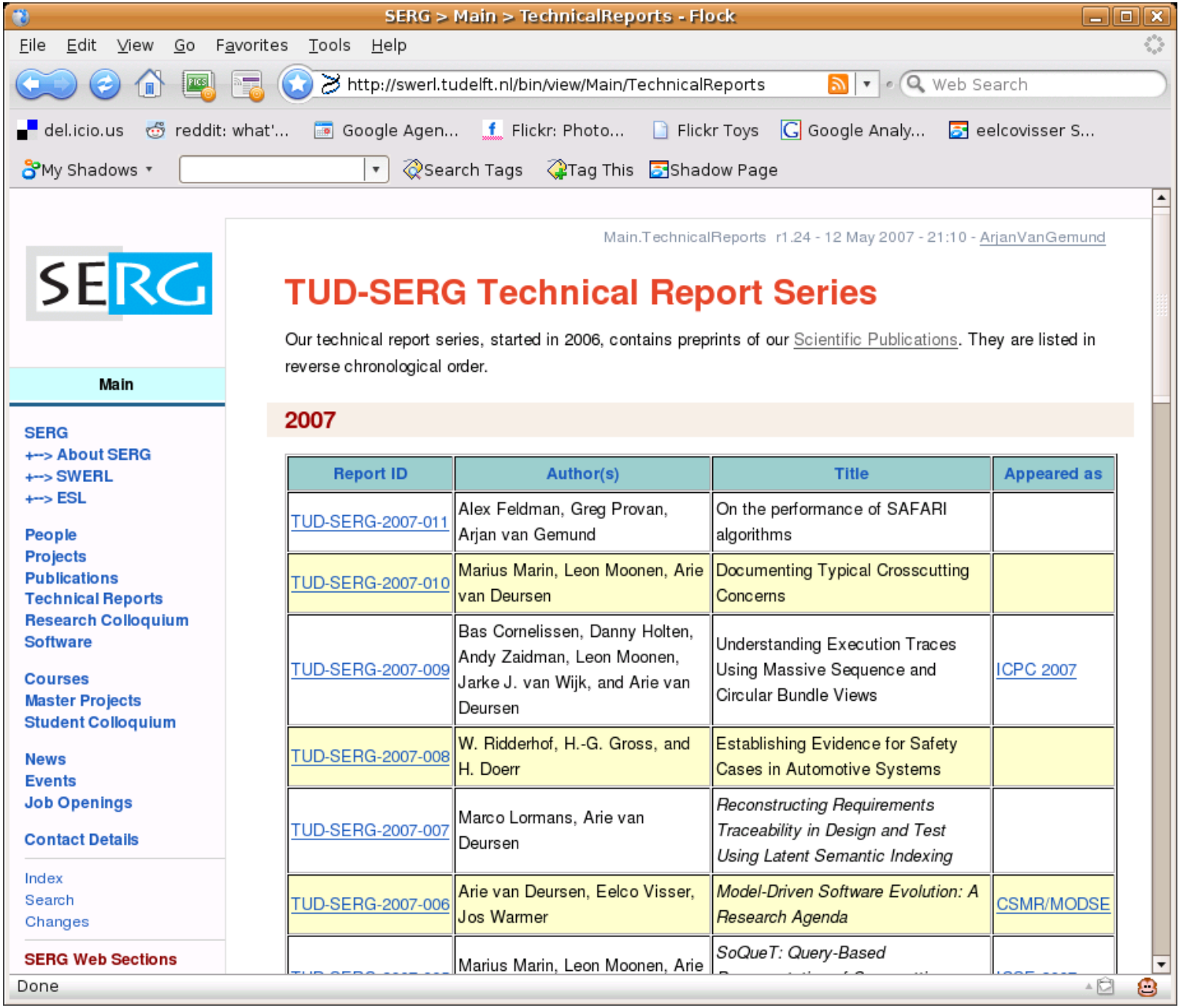
SERG: Publications



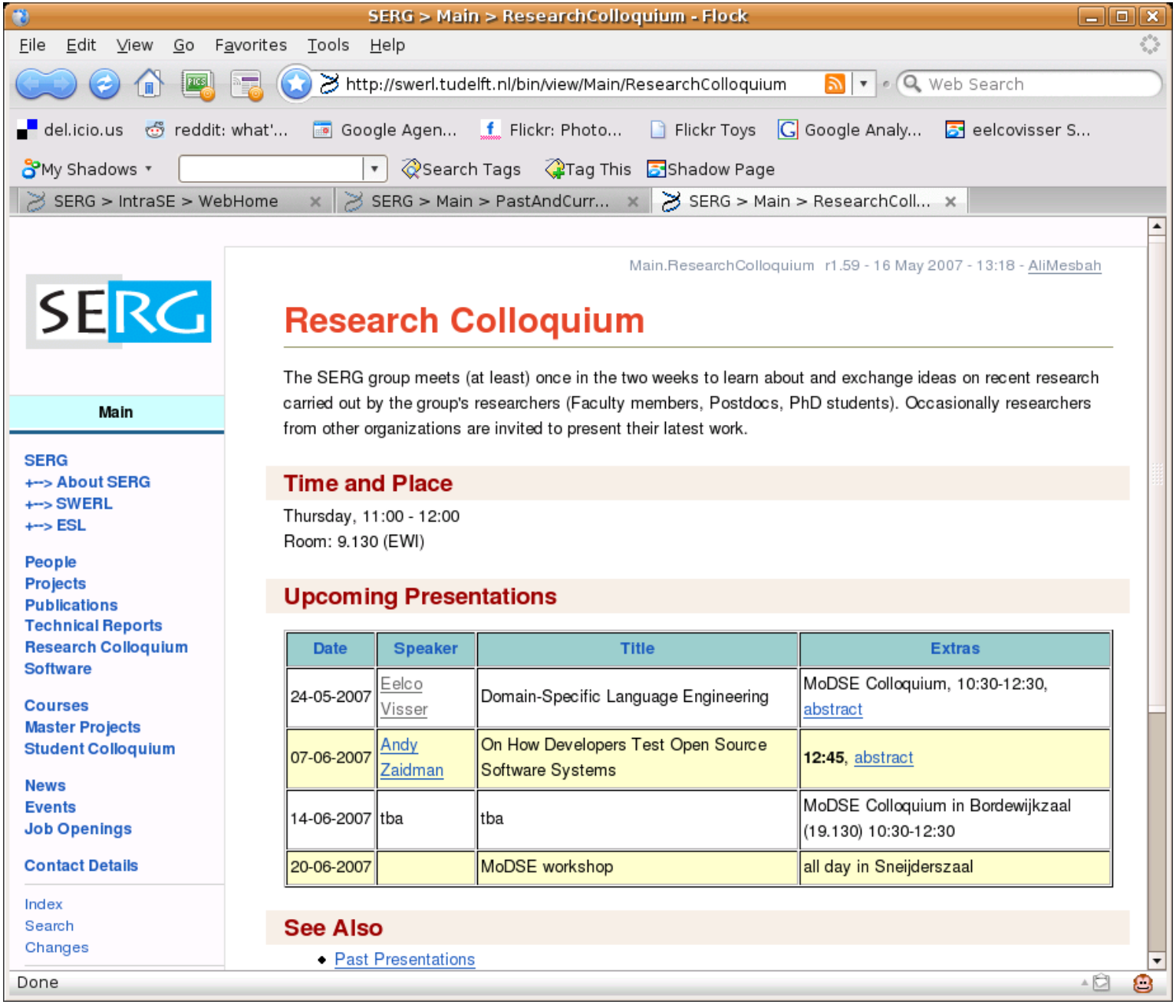
SERG: Homepages



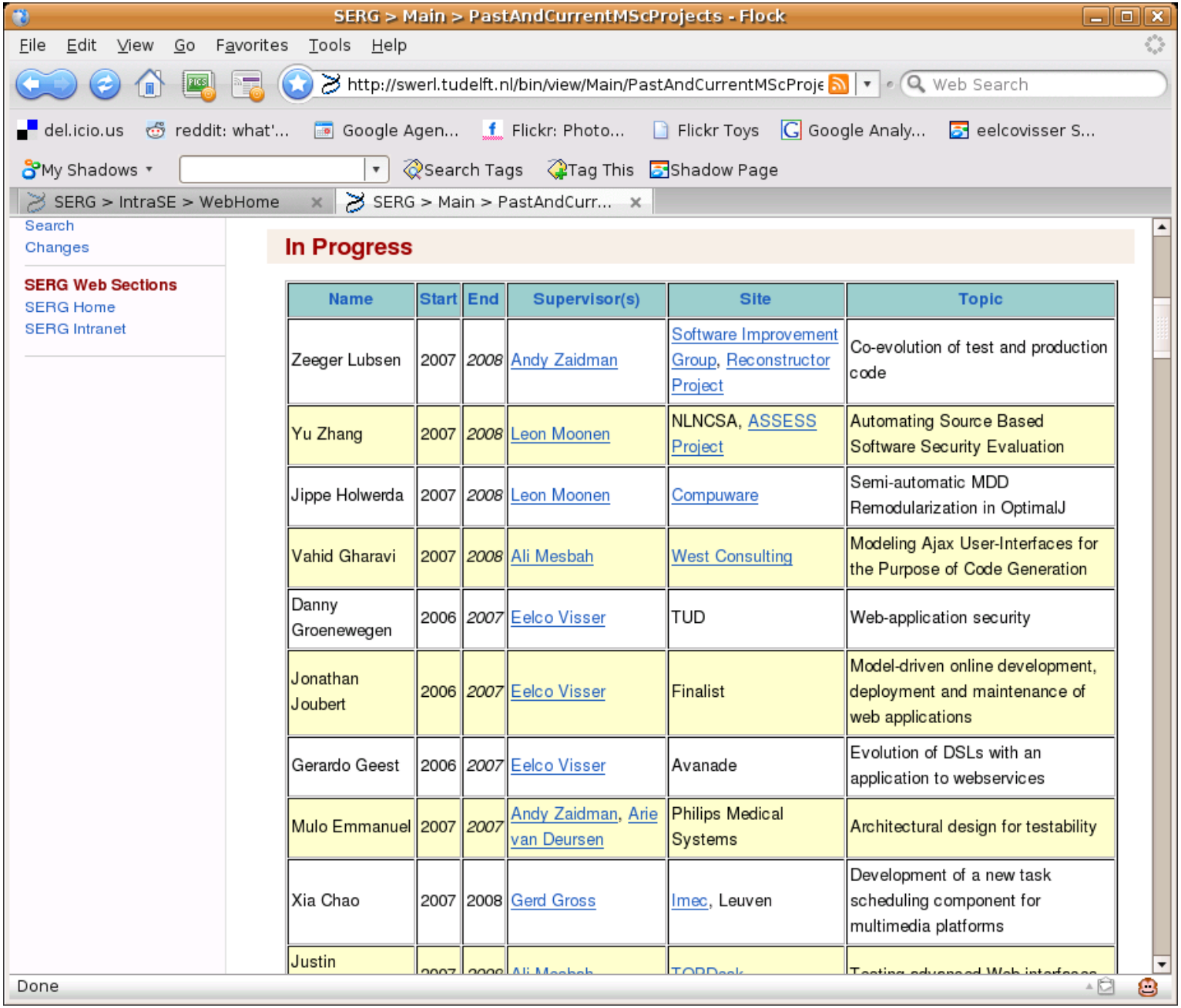
SERG: Technical Reports



SERG: Colloquium



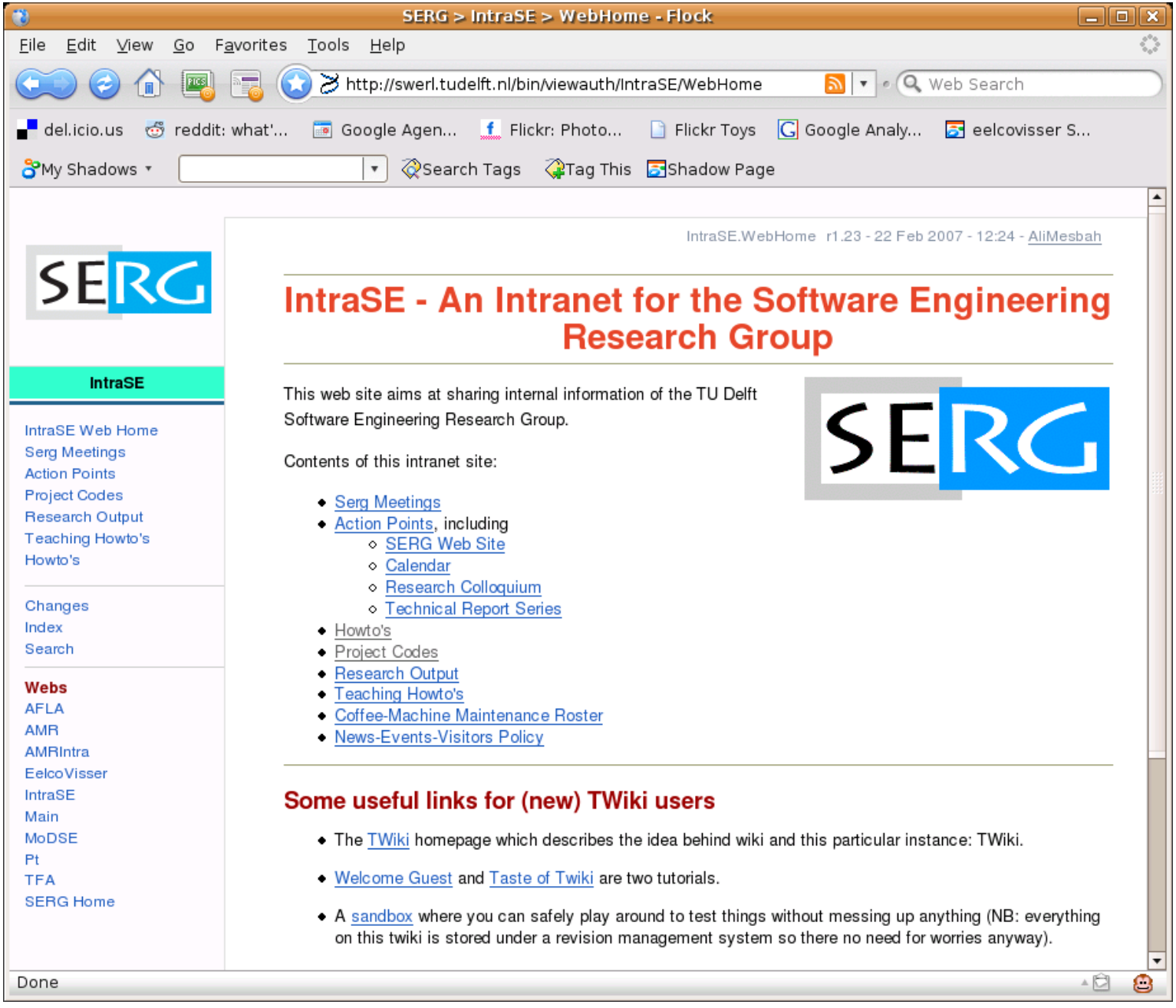
SERG: Students



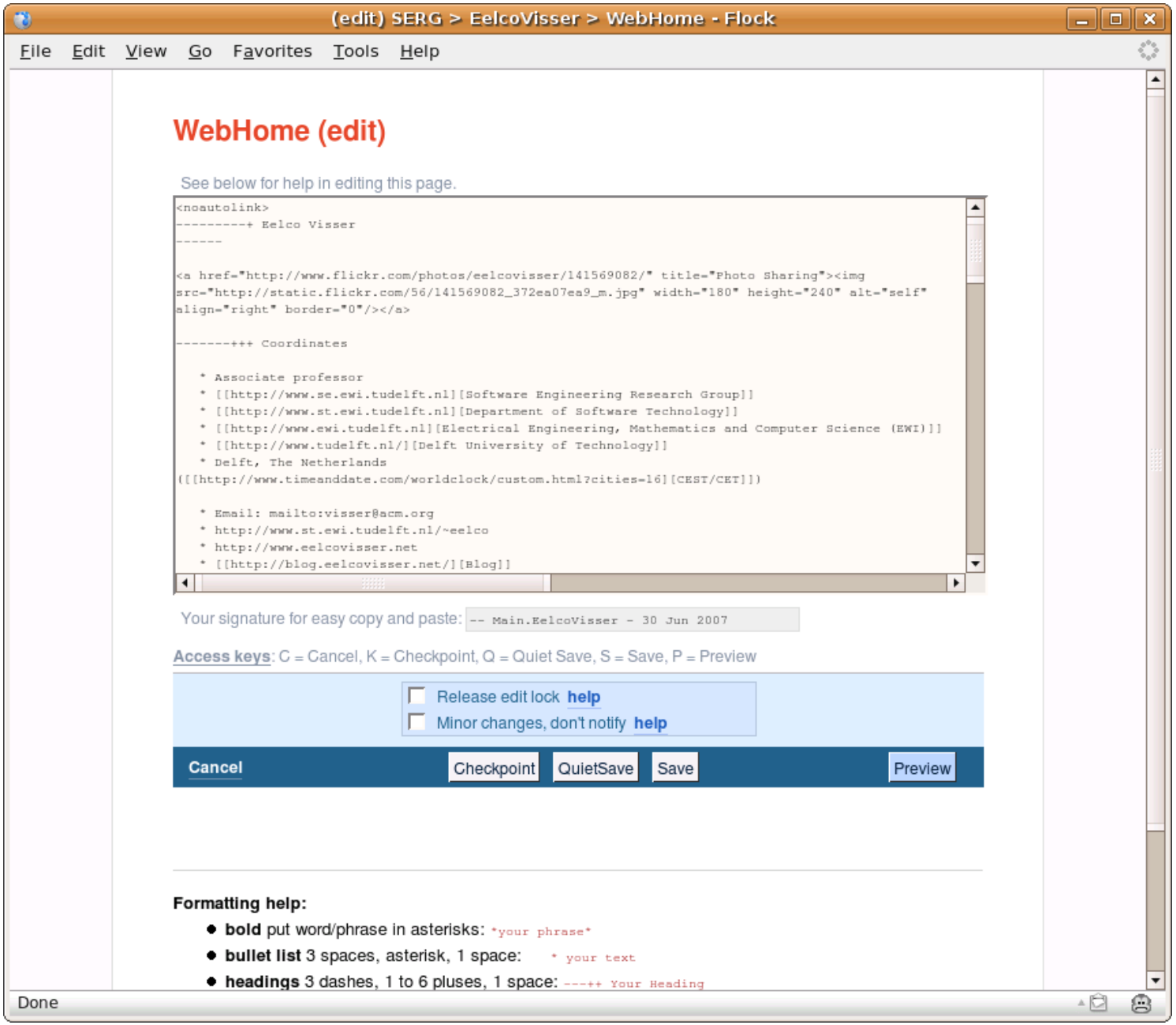
In Progress

Name	Start	End	Supervisor(s)	Site	Topic
Zeeger Lubsen	2007	*2008*	Andy Zaidman	Software Improvement Group, Reconstructor Project	Co-evolution of test and production code
Yu Zhang	2007	*2008*	Leon Moonen	NLNCSA, ASSESS Project	Automating Source Based Software Security Evaluation
Jippe Holwerda	2007	*2008*	Leon Moonen	Compuware	Semi-automatic MDD Remodularization in OptimalJ
Vahid Gharavi	2007	*2008*	Ali Mesbah	West Consulting	Modeling Ajax User-Interfaces for the Purpose of Code Generation
Danny Groenewegen	2006	*2007*	Eelco Visser	TUD	Web-application security
Jonathan Joubert	2006	*2007*	Eelco Visser	Finalist	Model-driven online development, deployment and maintenance of web applications
Gerardo Geest	2006	*2007*	Eelco Visser	Avanade	Evolution of DSLs with an application to webservices
Mulo Emmanuel	2007	*2007*	Andy Zaidman, Arie van Deursen	Philips Medical Systems	Architectural design for testability
Xia Chao	2007	2008	Gerd Gross	Imec, Leuven	Development of a new task scheduling component for multimedia platforms
Justin					

SERG: Internal Admin



SERG: Edit


(edit) SERG > EelcoVisser > WebHome - Flock
File Edit View Go Favorites Tools Help
WebHome (edit)
See below for help in editing this page.
<noautolink>
---------+ Eelco Visser

<img
src="http://static.flickr.com/56/141569082_372ea07ea9_m.jpg" width="180" height="240" alt="self"
align="right" border="0"/>
-------+++ Coordinates
* Associate professor
* [[http://www.se.ewi.tudelft.nl][Software Engineering Research Group]]
* [[http://www.st.ewi.tudelft.nl][Department of Software Technology]]
* [[http://www.ewi.tudelft.nl][Electrical Engineering, Mathematics and Computer Science (EWI)]]
* [[http://www.tudelft.nl/][Delft University of Technology]]
* Delft, The Netherlands
([[http://www.timeanddate.com/worldclock/custom.html?cities=16][CEST/CET]])
* Email: mailto:visser@acm.org
* http://www.st.ewi.tudelft.nl/~eelco
* http://www.eelcovisser.net
* [[http://blog.eelcovisser.net/][Blog]]
Your signature for easy copy and paste: -- Main.EelcoVisser - 30 Jun 2007
Access keys: C = Cancel, K = Checkpoint, Q = Quiet Save, S = Save, P = Preview
Release edit lock help
Minor changes, don't notify help
Cancel
Checkpoint
QuietSave
Save
Preview
Formatting help:
bold put word/phrase in asterisks: *your phrase*
bullet list 3 spaces, asterisk, 1 space: * your text
headings 3 dashes, 1 to 6 pluses, 1 space: ---++ Your Heading
Done

SERG: Domain Model

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

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

Inductive (bottom-up)

- Look at existing applications / frameworks in the domain
- Find common programming patterns
- Define abstractions that capture these patterns
- Risk: abstractions too close to existing practice
- Solution: iterative abstraction

Technology: Deductive vs Inductive

Deductive (top-down)

- Start with requirements obtained from domain analysis
- Match to existing technology and/or build your own components
- Advantage: perfect fit for requirements
- Risk: poor reuse, lot of effort, solution incompatible with mainstream technology

Inductive (bottom-up)

- What technology (libraries, frameworks) is available?
- How is this technology typically used?
    - e.g. ORM frameworks abstract from DBMS
- Incremental introduction of abstractions
    - Abstract from boilerplate in use of frameworks
    - Quick turn-around time for abstractions:
    - Implementation technique is clear

My Technology Stack

I chose the Java route (or it chose me)

- 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

Universe of Virtual machines

Many alternatives available

- PHP, Ruby/Rails, .Net, ...

Thousands of virtual machines

- each combination of languages, libraries, frameworks constitutes 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?

Part III

Capturing Common Programming Patterns

Architecture of Seam/JSF Web Application

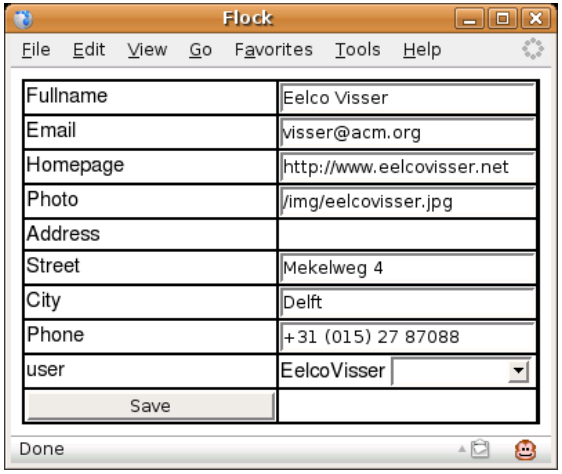
Tiers

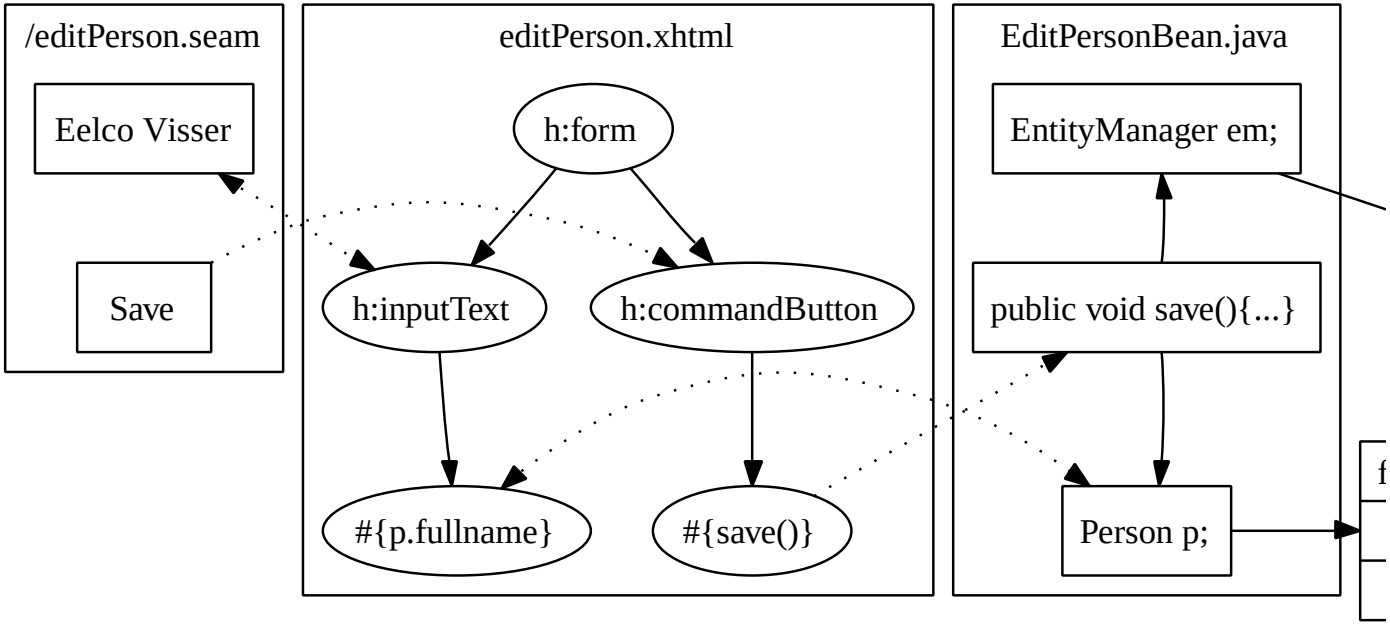
- Presentation layer
  - Java Server Faces (JSF)
- Session beans
  - Java objects that connect presentation and domain objects
- Domain objects
  - store persistent data
  - correspond to 'real-world' concepts

And

- A bit of configuration (XML)

Architecture of Seam/JSF Web Application





Programming Pattern: Entity Class

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

```
@Entity
public class Publication {

  public Publication () { }

  // properties

}
```

Programming Pattern: Entity Class / Identifier

Entities have an identifier as primary key

```
@Id @GeneratedValue
private Long id;

public Long getId() {
  return id;
}

private void setId(Long id) {
  this.id = id;
}
```

Programming Pattern: Entity Class / Value Type Property

Properties represent data (columns in database)

```
private String title;

public String getTitle() {
  return title;
}

public void setTitle(String title) {
  this.title = title;
}
```

Programming Pattern: Entity Class (continued)

Properties referring to other entities require annotations

```
@ManyToOne
@JoinColumn(name = "PublicationAuthor")
@Cascade({
  CascadeType.PERSIST,
  CascadeType.SAVE_UPDATE,
  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
}
```

Domain Model: Implementation

Implementing a DSL

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

Domain Model: Implementation

Implementing a DSL

- Definition of concrete syntax
  - using the syntax definition formalism SDF
- Parser
- Definition of abstract syntax
- Transformation of models to Java code

Domain Model: Implementation

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

Domain Model: Implementation

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

Domain Model: Implementation

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

Domain Model: Implementation

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

Domain Model: Syntax Definition in SDF

```
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")}
```

Domain Model: Abstract Syntax Definition

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
```

Domain Model: Parsing

Generate parser from syntax definition

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

Parsing gives abstract syntax *terms*

input: text

```
Person {
  fullname : String
  email    : String
  homepage : String
}
```

output: term

```
Entity("Person",
  [ Property("fullname", SimpleSort("String"))
  , Property("email",    SimpleSort("String"))
  , Property("homepage", SimpleSort("String"))
  ]
)
```

```
sglri -p DomainModel.tbl -i publication.dom | pp-aterm
```

Domain Model: Code Generation – Programs are models

Concrete syntax

```
@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(), []))
    ])
)
```

Domain Model: Code Generation by Term Rewriting

```
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(), []))
      ])
  )
```

Domain Model: Code Generation with Concrete Syntax

Use concrete syntax of Java in transformation rules

```
entity-to-class :
  |[ 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

```
entity-to-class :
  |[ x_Class { prop* } ]| ->
  |[
     @Entity 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*
```

Domain Model: Code Generation for Value Property

```
property-to-gettersetter :
  |[ x_prop : s ]| ->
  |[
     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 ]|
```

Domain Model: Code Generation for Entity Property

```
property-to-property-code(|x_Class) :
  |[ x_prop : s ]| ->
  |[
    @ManyToOne
    @Cascade({CascadeType.PERSIST,
              CascadeType.SAVE_UPDATE,
              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]
```

Domain Model: Code Generation / Declaring Entities

Propagate declared entities

```
declare-entity =
  ?|[ x_Class { prop* } ]|
  ; rules(
      defined-java-type :
        SimpleSort(x_Class) -> type|[ x_Class ]|
    )
```

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

Domain Model: Code Generator

Composing a code generator

```
webdsl-generator =
  xtc-io-wrap(webdsl-options,
    parse-webdsl
    ; alltd(declare-entity)
    ; collect(entity-to-class)
    ; output-generated-files
  )
```

What it does

- invoke parser to read input
- define dynamic rules for all declared entities
- generate java code for each entity declaration
- pretty-print each generated class to separate file

Capturing Programming Patterns: The Recipe

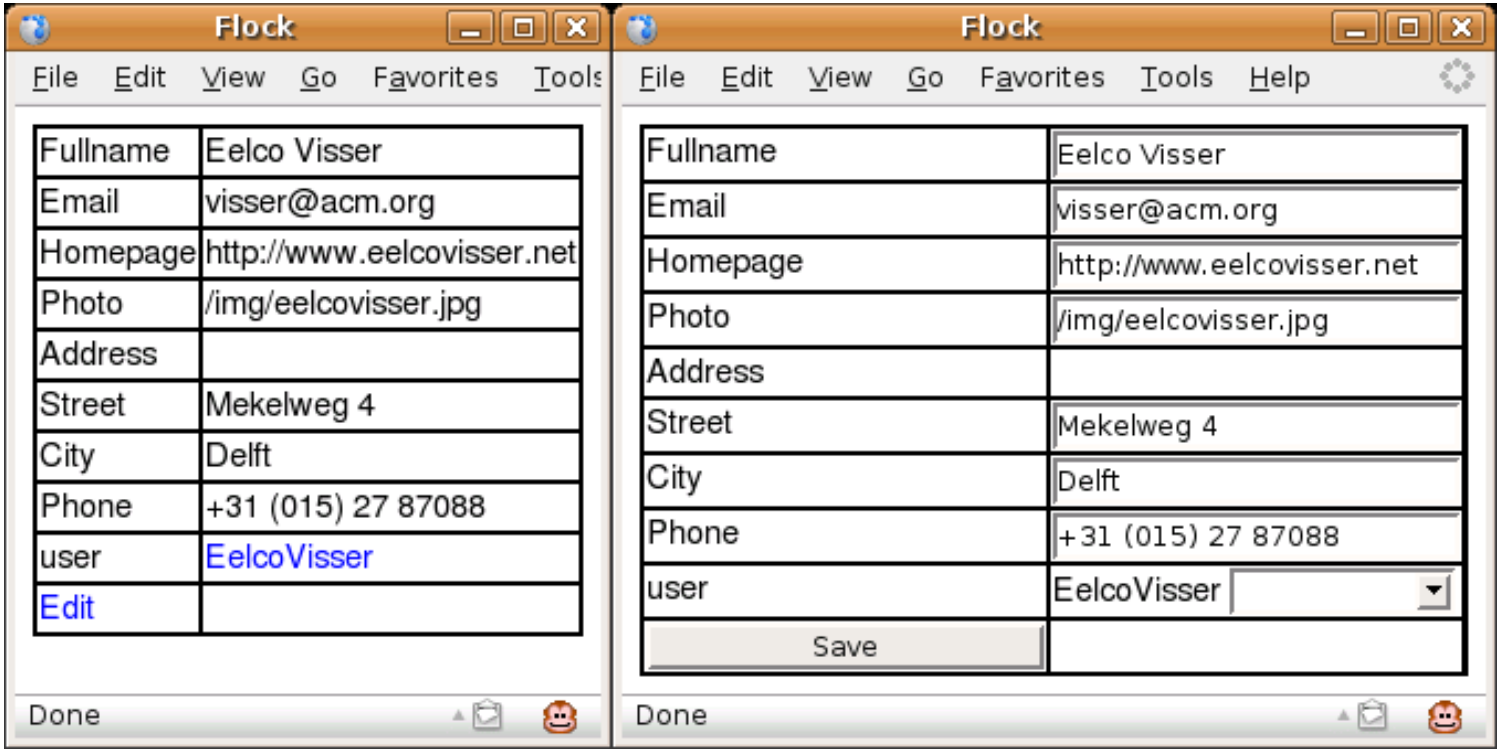
Recipe

- Find reoccurring programming patterns
- Factor out the repetitive code
- Turn parameters into DSL constructs
- Repetitive code fragments become rhs of rewrite rule

Part IV

Capturing More Programming Patterns

CRUD Userinterface

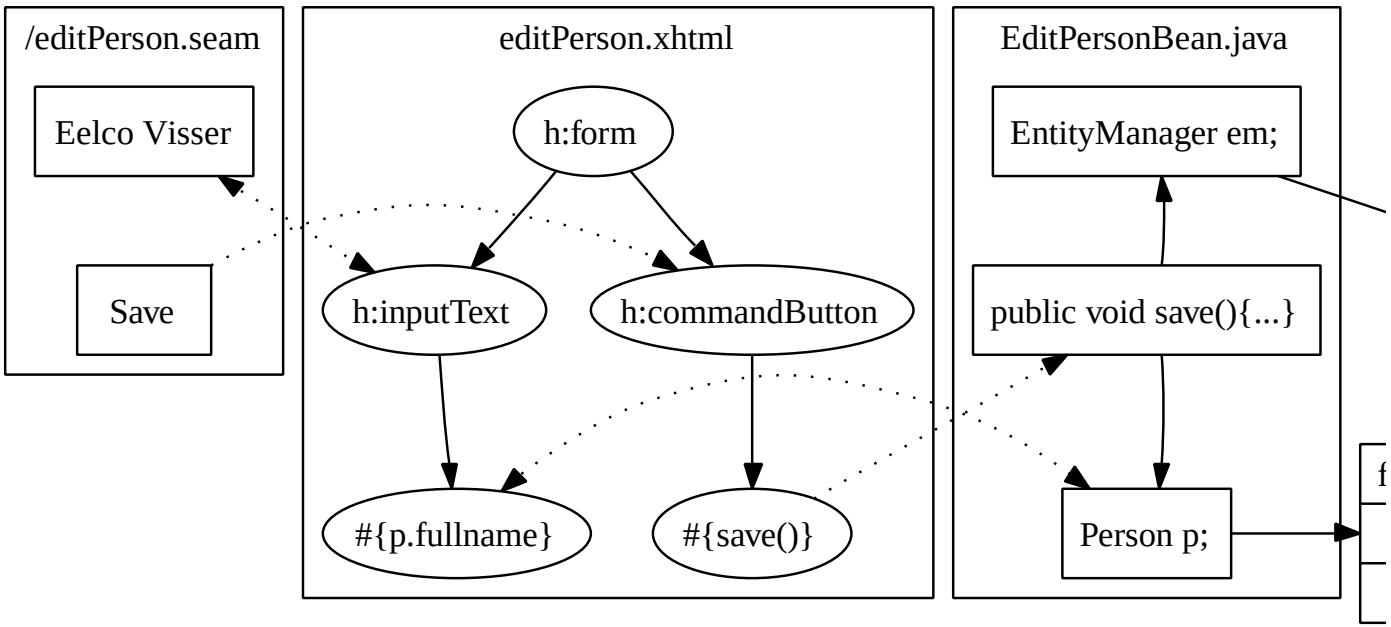



```
Person {
  fullname  : String
  email     : String
  homepage  : String
  photo     : String
  address   : Address
  user      : User
}
```

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

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

Architecture of Seam/JSF Web Application



Java Server Faces: viewPerson.xhtml

```
<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>
</table>
```

Generating JSF Pages: Entities

```
entity-to-xhtml-viewEntity :
  |[ x_Class { 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
```

Generating JSF Pages: Properties (1)

```
row-in-view-form(|x_class)  :
  prop@|[ x_prop : s ]| ->
  %>
    <tr>
      <td> <h:outputText value="<%=x_prop%>"/> </td>
      <td> <%= input %> </td>
    </tr>
  <%
  where input := <property-to-view-component(|x_class)> prop
```

Generating JSF Pages: Properties (2)

Output of property

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

Input of property

```
property-to-edit-component(|x_component) :
  |[ x_prop : String ]| ->
  %>
    <h:inputText value="#{<%=x_component%>.<%=x_prop%>}"/>
  <%
```

Seam Session Beans: ViewPersonBean.java

Seam component can be approached directly from JSF page

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

  @Destroy @Remove              // required for stateful beans
  public void destroy() { }
}
```

Seam Session Beans: ViewPersonBean.java

Necessary services are obtained by *injection*

```
@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

Seam Session Beans: ViewPersonBean.java

Domain object made available to JSF via property

```
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 paramater

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

Initialization of object based on parameter identifier

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

Generating Session Beans

Replacing names in boilerplate code

```
entity-to-session-bean :
  |[ 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 :
  |[ mod* t x(arg*) { bstm* } ]| -> |[ mod* t x(arg*); ]|
  where <fetch(?Public())> mod*
```

Summary

Domain modeling language

- generate entity classes

Generate userinterface

- very basic UI for viewing and editing objects

What have we learned?

- understanding of the basics of the technology
- setup of a complete generator

Next

- refining domain modeling language
- consider proper UI

Part V

Refining Programming Patterns

Special Types

Special types allow to generate refined behaviour

```
Person {                    User {
  fullname  : String          username : String
  email     : Email           password : Secret
  homepage  : URL             person   : Person
  photo     : Image         }
  address   : Address
  user      : User
}
```

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

property-to-edit-component(|x_component) :
  |[ x_prop : 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

```
property-to-property-code(|x_Class) :
  |[ x_prop : List<t> ]| -> |[
     @ManyToMany
     @Cascade({
        CascadeType.PERSIST,
        CascadeType.SAVE_UPDATE,
        CascadeType.MERGE
     })
     private List<t> x_prop = new LinkedList<t>();
  ]|
```

Refining Associations

Value Types

- `title :: String`

Composite Associations

- `address <> Address`

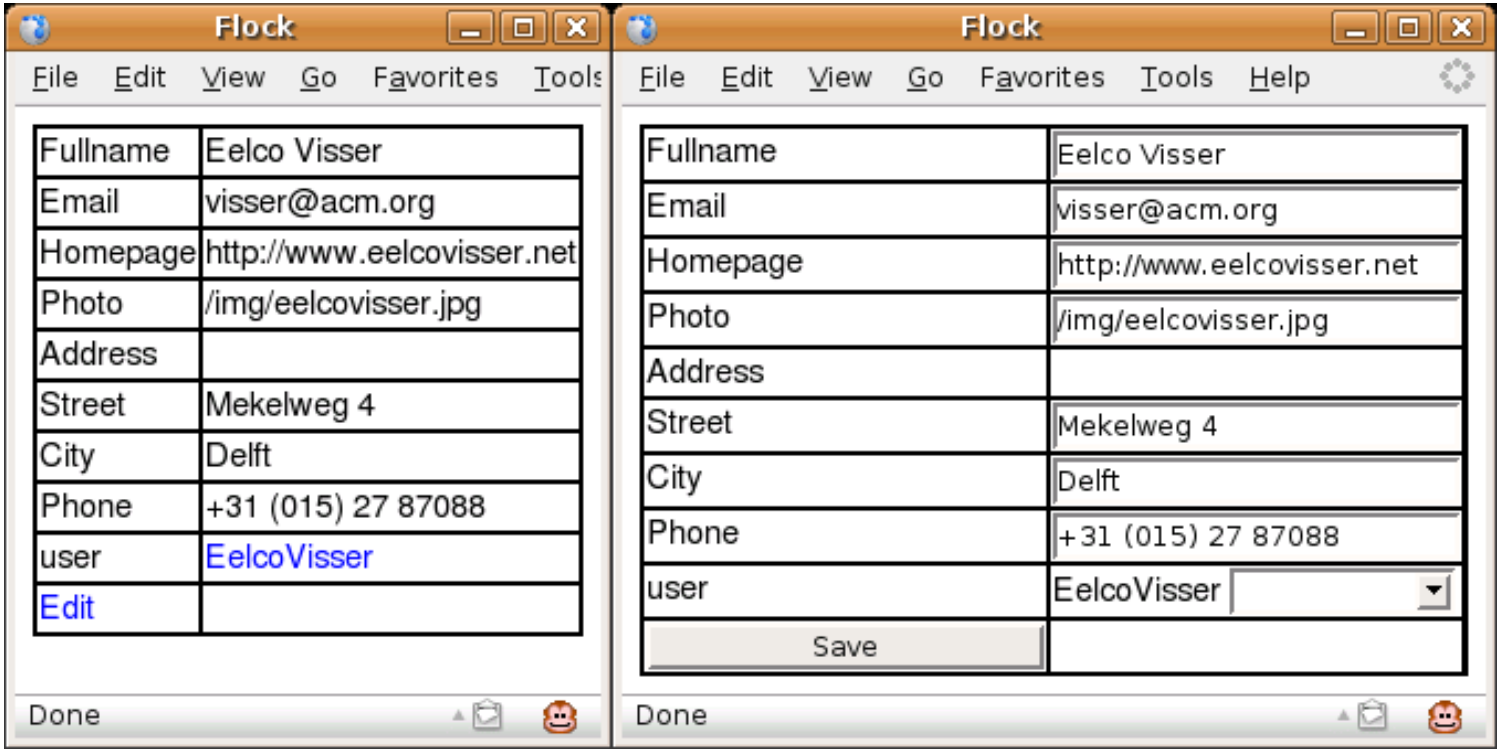
Reference Associations

- `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




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

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

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

Generating JSF Pages: Unfolding Entities

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

Summary

Recipe

- Turn programming patterns into generator rules

DSL

- Language for domain models
- With refinements to support sophisticated crud operations
- Generate entity classes, session beans, and JSF pages

Techniques

- Declarative syntax definition
- Term rewriting with concrete syntax

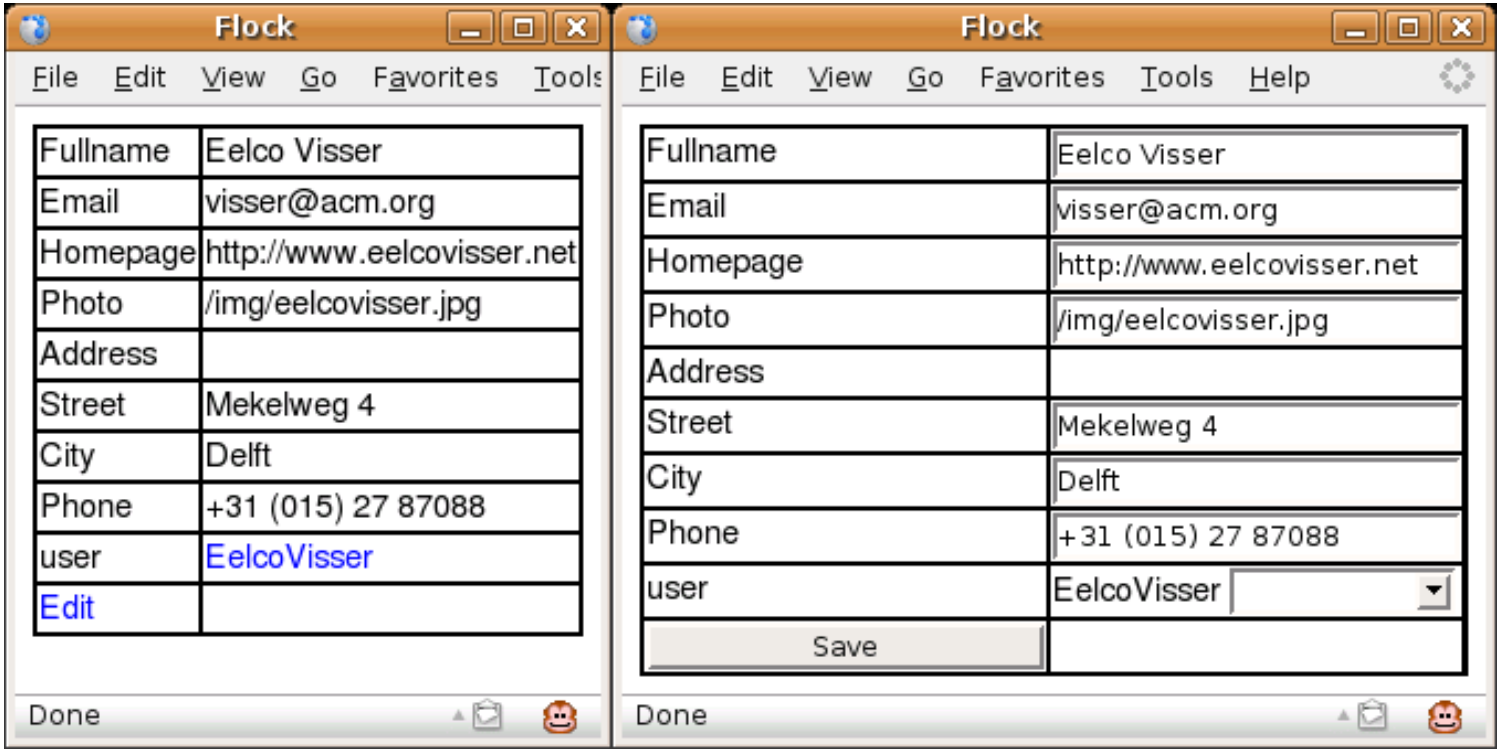
Next

- Scrap your boilertemplate: refactoring the generator

Part VI

Scrap your Boilertemplate™

Generating CRUD pages from domain models




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

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

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

Domain-driven Application Generation

For each entity generate several types of artifacts

- Entity class
- View page
- Edit page
- Page with all objects
- Search page
- ...

For each page generate

- JSF file
- Java session bean
- Local interface of session bean

But you don't want that!

Limited expressivity

- Adding new type of page requires extending the generator

Code duplication in templates

- Templates are large
- Similar coding patterns are used in different templates
- Only a complete page type is considered as a reusable pattern

Time for template refactoring

- Intermediate language for defining presentations

'This does not look like a compiler'

Eelco Visser to Martin Bravenboer

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

Presentation and Page Flow

Language for defining page presentation and flow

- Derive from one definition
  - the JSF presentation
  - the Java implementation of the session beans
- Inspiration: LaTeX
  - TeX provides basic machinery for typesetting
  - LaTeX provides abstractions for structuring documents
  - LaTeX philosophy: separate layout from content
  - Advantage over XML/HTML: user-definable abstraction mechanism (macros)

Composing Presentations: Running Example

```
ResearchGroup {
  acronym   :: String (name)
  fullname  :: String
  mission   :: Text
  logo      :: Image
  members   -> Set<Person>
  projects  -> Set<ResearchProject>
  colloquia -> Set<Colloquium>
  news      -> List<News>
}
```

Page Flow

Page definition

```
define page viewResearchGroup(group : ResearchGroup) {
  <presentation>
}
```

→ URL

```
/viewResearchGroup.seam?group=1
```

Page navigation

```
navigate(pers.group.acronym,viewResearchGroup(pers.group))
```

→ Link

```
<a href="/viewResearchGroup.seam?group=1">SERG</a>
```

Composing Presentations: Content Markup

Composing Presentations: Content Markup

```
define page viewResearchGroup(group : ResearchGroup) {
  section {
    header{text(group.fullname)}
    section {
      header{"Mission"}
      outputText(group.mission)
    }
    section {
      header{"Recent Publications"}
      list { ... }
    }
    section {
      header{"People"}
      list { for(p : Person in group.members) {
        listitem { navigate(p.name, viewPerson(p)) }
      } }
    }
  }
}
```

Composing Presentations: Page Layout

Composing Presentations: Page Layout

```
define page viewResearchGroup(group : ResearchGroup) {
  block("outersidebar"){
    block("logo"){ ... }
    block("sidebar"){ ... }
  }
  block("outerbody"){
    block("menubar"){
      block("menu"){ ... }
    }
    block("body"){
      section {
        header{text(group.fullname)}
          ...
        }
    }
  }
}
```

Composing Presentations: Page Layout with CSS

```
.outersidebar {
  position   : absolute;
  overflow   : hidden;
  top        : 0px;
  left       : 10px;
  margin-top : 10px;
  width      : 10em;
}

.logo {
  text-align : left;
}

.sidebar {
  top          : 0px;
  margin-top   : 20px;
  color        : darkblue;
  border-right : 1px dotted;
}
```

```
.outerbody {
  position : absolute;
  top      : 10px;
  left     : 12.5em;
  right    : 40px;
}

.menubar {
  height        : 62px;
  border-bottom : 1px dotted;
  color         : darkblue;
}

.body {
  position      : relative;
  top           : 20px;
  margin-bottom : 2.5em;
}
```

Composing Presentations: Sidebar

Composing Presentations: Sidebar is just a list

```
block("sidebar"){
  list {
    listitem {
      navigate(group.acronym, viewResearchGroup(group))
    }
    listitem{
      navigate("People", groupMembers(group))
    }
    listitem{
      navigate("Publications", groupPublications(group))
    }
    listitem{
      navigate("Projects", groupProjects(group))
      list{ for( p : ResearchProject in group.projectsList ) {
        listitem{ navigate(p.name, viewResearchProject(p)) }
      } }
    }
    ...
 }
}
```

Composing Presentations: Styling Sidebar

```
.sidebar ul {
  list-style : none;
  margin     : 0em;
  padding    : 0px;
}

.sidebar ul li {
  margin     : 0em;
  padding    : 0px;
}

.sidebar ul ul {
  list-style-type : square;
  font-size       : .8em;
  padding         : .2em;
  margin-left     : 1em;
}
```

Composing Presentations: Drop Down Menus

Composing Presentations: Menu is just a list

```
block("menu") {
  list{
    listitem{
      "People"
      list{ for(person : Person) {
        listitem{ navigate(person.name, viewPerson(person)) }
      } }
    }
  }
  list {
    listitem {
      "Projects"
      list { for(p : ResearchProject) {
        listitem { navigate(p.acronym, viewResearchProject(p)) }
      } }
    }
 }
 ...
}
```

Composing Presentations: Styling Menus with :hover

```
div.menu ul ul,
div.menu ul li:hover ul ul,
div.menu ul ul li:hover ul ul,
div.menu ul li table
{
  display: none;
}

div.menu ul li:hover ul,
div.menu ul ul li:hover ul,
div.menu ul ul ul li:hover ul,
div.menu ul li:hover table
{
  display          : block;
  width            : 9em;
  border           : ...;
  background-color : white;
}
```

Styling Presentations

Current elements provide basics

- CSS goes a long way
- AJAX/JavaScript is complementary
  - map to appropriate JSF tag library (e.g. richfaces)
  - keep abstractions general

Presentation Language Constructs: Template Call

Template Call

- concrete syntax

  `f(e1,...,em) {elem1 ... elemn}`
- abstract syntax

  `TemplateCall(f, [e1,...,em], [elem1, ..., elemn])`
- expression and element argument lists are optional

Examples

- `block("menu") { ... }`
- `section { header{ ... } ... }`
- `list { listitem { ... } ... }`
- `table { row{ ... } row{ ... } }`
- `text(group.name)`
- `navigate(pub.name, viewPublication(pub))`

Presentation Language Constructs: Iteration

Iteration

- concrete syntax

```
for( x : sort in e ) { elem* }
```

- abstract syntax

```
For(x, sort, e, elem*)
```

Example

-
```
list {
  for(p : ResearchProject in pers.groups) {
    listitem {
      navigate(p.acronym, viewResearchProject(p))
    }
  }
}
```

Part VII

Translating to JSF+Seam

Mapping Pages to JSF+Seam

Page to JSF

- presentation elements to JSF components
- object access expressions to JSF EL expressions

Page to Seam Session Bean

- connect JSF page to entity objects
- properties for page arguments
- datamodels for iteration

Mapping Pages to JSF+Seam

```
User { name :: String }
page viewUser(user : User) {
  text(user.name)
}
```

```
@Stateful @Name("viewUser")
class viewUserBean {
  @PersistenceContext
    EntityManager em;
  @RequestParameter("user")
    private Long userId;
  property User user;
  @Begin @Create
  public void initialize() {
    user =
     em.find(User.class,userId)
  }
}
```

```
<html ...> ...
<body>
  <h:outputText value="#{viewUser.user.name}"/>
</body>
</html>
```

Mapping Presentation Elements to JSF

Basic element

```
elem-to-xhtml :
  Text(x) -> %> <h:outputText value="<%=x%>"/> <%
```

Recursive call

```
elem-to-xhtml :
  |[ block(str){ elem* } ]| ->
  %>
    <div class="<%= str %>">
       <%= <elems-to-xhtml> elems* ::*%>
    </div>
  <%
```

Mapping Presentation Elements to JSF: Navigate

```
navigate(viewPerson(p)){text(p.name)}
```

```
<s:link view="/viewPerson.xhtml">
 <f:param> name="person" value="#{p.id}" />
 <h:outputText value="#{p.name}" />
</s:link>
```

```
elem-to-xhtml :
  |[ navigate(p(args)){elems1} ]| ->
  %> <s:link view = "/<%= p %>.xhtml"><%=
       <conc>(params,elems2) ::*
  %></s:link> <%
  where <IsPage> p
      ; fargs  := <TemplateArguments> p
      ; params := <zip(bind-param)> (fargs, args)
      ; elems2 := <elems-to-xhtml> elems1
bind-param :
  (|[ x : s ]|, e) ->
  %><f:param name="<%= x %>" value="<%= el %>" /><%
  where <defined-java-type> s
      ; el := <arg-to-value-string> |[ e.id ]|
```

Mapping Presentation Elements to JSF: Iteration

```
list{ for ( project : ResearchProject ) {
  listitem { navigate(project.acronym,viewResearchProject(project)) }
}}
```

```
<ul> <ui:repeat var="project"
                value="#{viewResearchGroup.group.projectsList}">
   <li> <s:link view="/viewResearchProject.xhtml">
          <f:param name="researchProject" value="#{project.id}"/>
          <h:outputText value="#{project.name}"/>
        </s:link> </li>
</ui:repeat> </ul>
```

```
elem-to-xhtml :
  |[ for(x : s in e){elem*} ]| ->
  %>
    <ui:repeat var="<%= x %>" value="<%= el %>">
      <%= <elems-to-xhtml> elem* ::*%>
    </ui:repeat>
  <%
  where el := <arg-to-value-string> e
```

Mapping Presentation Elements to JSF: Nested Sections

```
section{
  header{"Foo"} ...
  section{ header{"Bar"} ... }
}

<h1>Foo</h1> ...
<h2>Bar</h2> ...
```

```
elem-to-xhtml :
  |[ section{elems1} ]| -> elems2
  where {| SectionDepth
         : rules( SectionDepth := <SectionDepth; inc> )
         ; elems2 := <elems-to-xhtml> elems1
         |}
elem-to-xhtml :
  |[ header{elems} ]| ->
  %>
    <~n:tag><%= <elems-to-xhtml> elems ::*%></~n:tag>
  <%
  where n := <SectionDepth>
      ; tag := <concat-strings>["h", <int-to-string> n]
```

Mapping Pages to JSF+Seam

```
User { name :: String }
page viewUser(user : User) {
  text(user.name)
}
```

```
@Stateful @Name("viewUser")
class viewUserBean {
  @PersistenceContext
    EntityManager em;
  @RequestParameter("user")
    private Long userId;
  property User user;
  @Create @Begin
  public void initialize() {
    user =
     em.find(User.class,userId)
  }
}
```

```
<html ...> ...
<body>
  <h:outputText value="#{viewUser.user.name}"/>
</body>
</html>
```

Mapping Pages to Seam: Page to Compilation Unit

```
page-to-java :
  def@|[ define page x_page(args){elems1} ]| ->
  compilation-unit|[
    @Stateful @Name("~x_page")
    public class x_PageBean implements x_PageBeanInterface {

      @PersistenceContext private EntityManager em;

      @Create @Begin public void initialize() { bstm* }

      @Destroy @Remove public void destroy() {}

      ~*cbd*
    }
  ]|)
  where x_Page     := <capitalize-string> x_page
      ; x_PageBean := <concat-strings> [x_Page, "Bean"]
      ; cbd*       := <collect(page-elem-to-method)> def
      ; bstm*      := <collect(page-elem-to-init)> def
```

Mapping Pages to Seam: Page Arguments

```
argument-to-bean-property :
  |[ x : x_Class ]| ->
  |[
     @RequestParameter("~x") private Long x_Id;
     private x_Class x;
     public void x_set(x_Class x) { this.x = x; }
     public x_Class x_get() { return x; }
  ]|
  where x_Id  := <concat-strings>[x, "Id"]
      ; x_get := <property-getter> x
      ; x_set := <property-setter> x

argument-to-initialization :
  |[ x : x_Class ]| ->
  |[
     if (x_Id == null) { x = new x_Class(); }
     else { x = em.find(x_Class.class, x_Id); }
  ]|
  where x_Id := <concat-strings>[x, "Id"]
```

Mapping Pages to JSF+Seam

Now that looks more like a compiler!

- language constructs that do one thing
- translation rules with (mostly) small right-hand sides

Next

- Extensions
  - completing the core language with
  - typechecking, actions, queries
- Not all abstraction can be generative
  - abstraction mechanisms for the application developer
  - templates, modules
- More sugar, please
  - enriching the DSL with higher level abstractions
  - implemented using desugarings (model-to-model transformations)

Part VIII

Demonstration

Part IX

Extensions

Typechecking

JSF

- JSF pages 'compiled' at run-time
- Many causes of errors unchecked
  - Missing or non-supported tags
  - References to non-existing properties
  - References to non-existing components
- Cause run-time exceptions

Seam

- Seam component annotations scanned at deployment-time
- Method not declared in `@Local` interface not found (silent)

WebDSL

- WebDSL programs are statically typechecked
- Typechecker annotates expressions with their type, which is key to type-based desugarings

Typechecking: Example

```
User {
  name :: String
}
define page viewUser(user : User) {
  text(user.fullname)
  text(us.name)
}
```

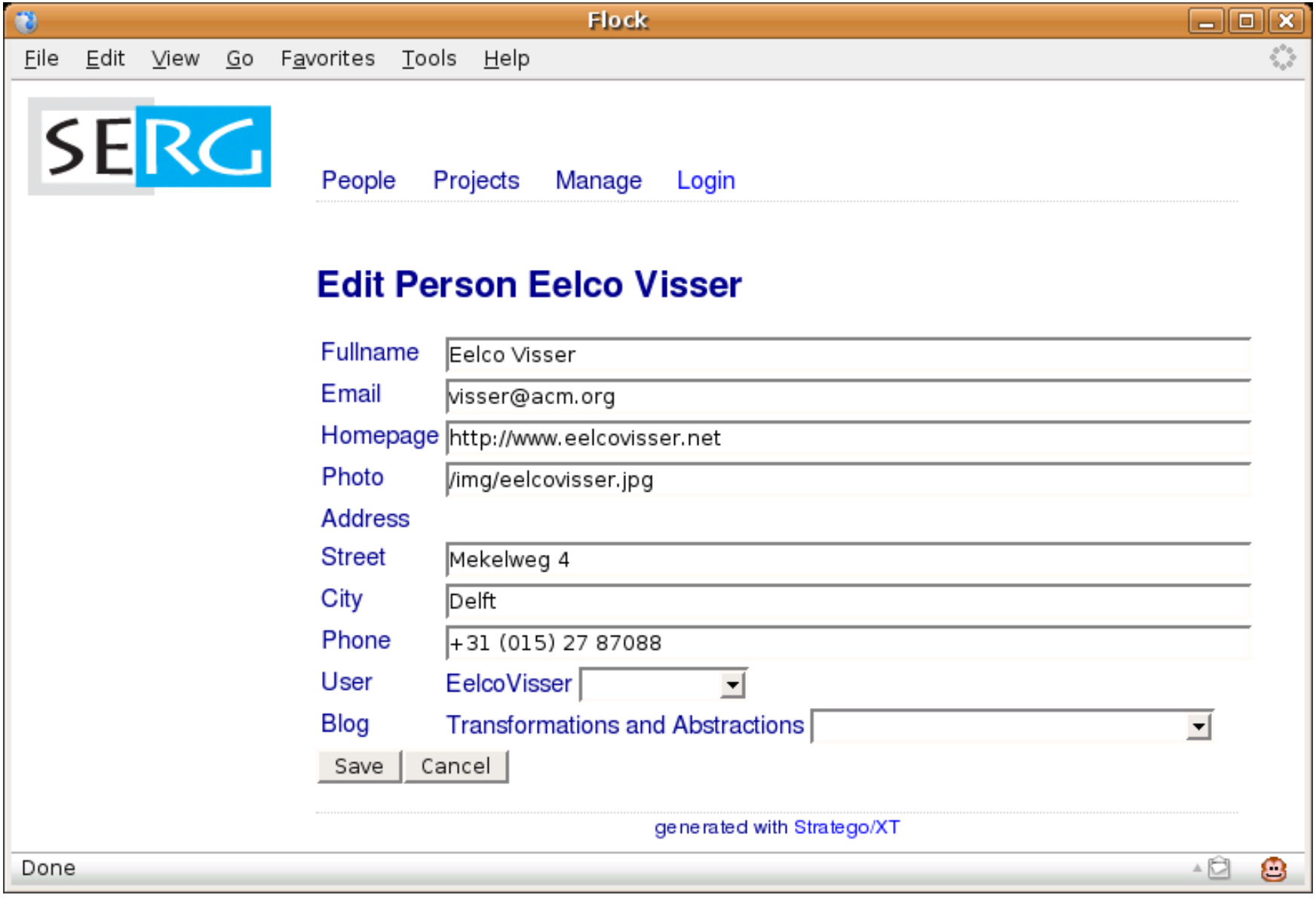
```
$ dsl-to-seam -i test.app
[error] definition viewUser/text/:
        expression 'user.fullname' has type error
[error] definition viewUser/text/:
        variable 'us' has no declared type
```

(error messages are not quite as pretty yet)

Typechecking: Rules

```
typecheck-iterator :
  For(x, s, e1, elems1) -> For(x, s, e2, elems2)
  where in-tc-context(id
        ; e2 := <typecheck-expression> e1
        ; <should-have-list-type> e2
        ; {| TypeOf
           : if not(<java-type> s) then
               typecheck-error(|[
                 "index ", x, " has invalid type ", s
               ])
             else
               rules( TypeOf : x -> s )
             end
           ; elems2 := <typecheck-page-elements> elems1
           |}
        | ["iterator ", x, "/"] )
```

Data Input and Actions



Data Input and Actions: Translation

```
User { name :: String }
page editUser(user : User) {
  form{
    inputString(user.name)
    action("Save", save())
    action save() {
      user.save();
      return viewUser(user);
    }
  }
}
```

```
@Stateful @Name("editUser")
class viewUserBean {
  property User user;
  @End public String save()
  {
    em.persist(this.getUser());
    return "/viewUser.seam"
         + "?user=" + user.getId();
  }
}
```

```
<h:form>
  <h:inputText value="#{editUser.user.username}"/>
  <h:commandButton type="submit" value="Save"
                   action="#{editUser.save()}"/>
</h:form>
```

Action Language Constructs

Expressions

- Object creation: `Person{ name := e ... }`
- Set creation: `{ e1, e2, ... }`
- List creation: `[ e1, e2, ... ]`
- Variables, constants, field access

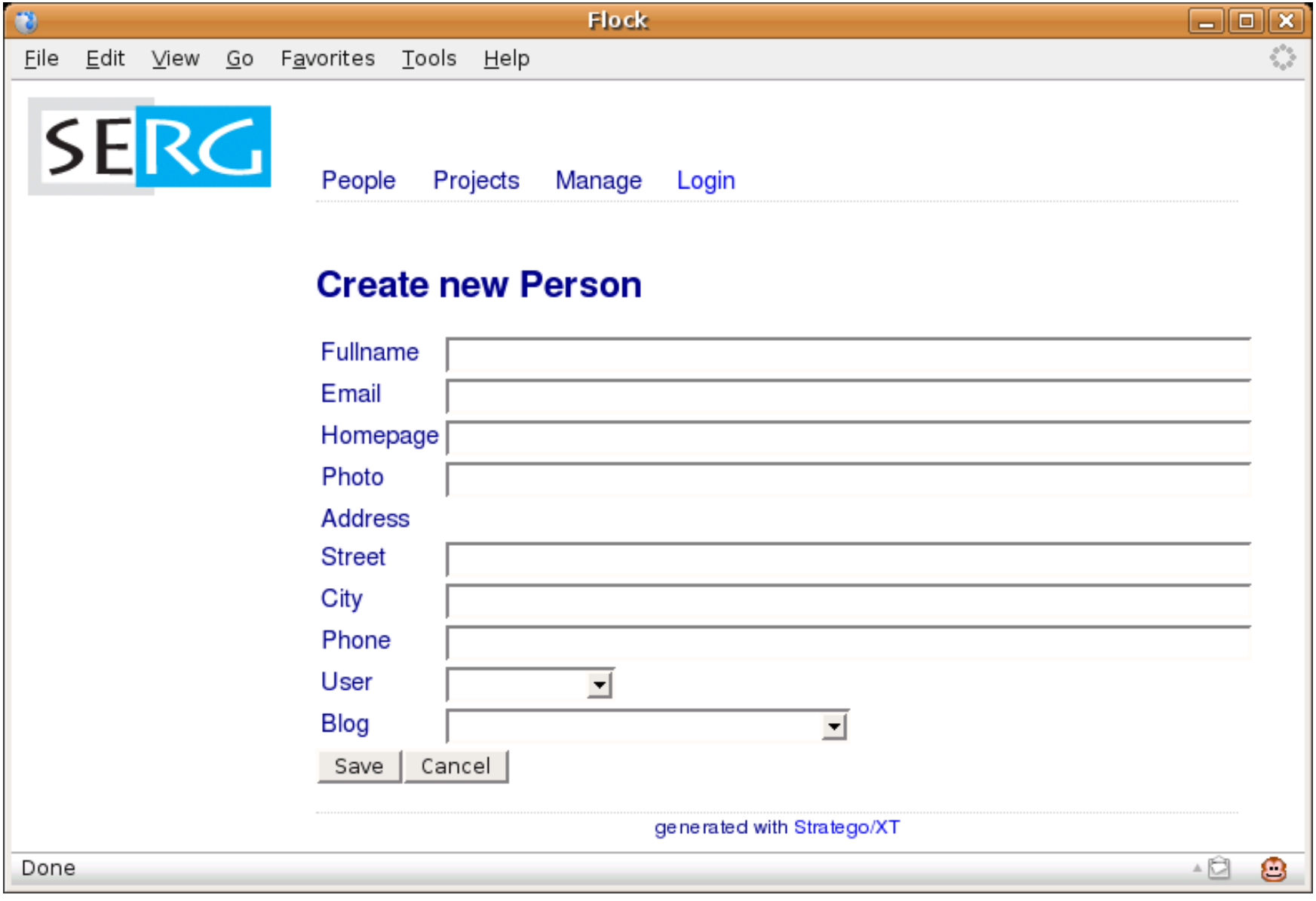
Statements

- Assignment: `person.blog := Blog{ title := name };`
- Method call: `publication.authors.remove(author);`
- Return: `return viewUser(u);` (page-flow)

Embed Java (subset)?

+ solid syntax and semantics
- no control over what is used
- no translation to other platforms
- typechecking and other analyses much harder (reuse dryad?)

Page Local Variables



Page Local Variables

```
User { name :: String }
page createUser() {
  var user : User := User{};
  form{
   inputString(user.name)
   action("Save", save())
   action save() {
     user.save();
     return viewUser(user);
   }
}
```

```
@Stateful @Name("editUser")
class viewUserBean {
  property User user;
  @Create @Begin
  public void initialize() {
    user = new User();
  }
  @End public String save() {
    em.persist(this.getUser());
    return "/viewUser.seam"
         + "?user=" + user.getId();
  }
}
```

```
<h:form>
  <h:inputText value="#{createUser.user.username}"/>
  <h:commandButton type="submit" value="Save"
                   action="#{createUser.save()}"/>
</h:form>
```

Queries


Martin Bravenboer - Flock
File Edit View Go Favorites Tools Help
SERG
People Projects Manage Login
Martin Bravenboer
Publications
Blog
Projects
TFA
TraCE
Martin Bravenboer
Coordinates
homepage http://martin.bravenboer.name
email martin.bravenboer@gmail.com
address Mekelweg 4
Delft
phone 015
Publications
Grammar Engineering Support for Precedence Rule Recovery and Compatibility Checking (2007)
Preventing Injection Attacks with Syntax Embeddings (2007)
Done

Queries: Embedding HQL

```
User{ name :: String } Publication{ authors -> List<User> }

page viewUser(user : User) {
  var pubs : List<Publication> :=
      select pub from Publication as pub, User as u
       where (u.id = ~user.id) and (u member of pub.authors)
       order by pub.year descending;
  for(p : Publication in pubs) { ... }
}
```

```
class viewUserBean {
  property List<Publication> pubs;
  @Factory("pubs") public void initPubs() {
    pubs = em.createQuery(
      "select pub from Publication as pub, User as u" +
      " where (u.id = :param1) and (u member of pub.authors)" +
      " order by pub.year descending"
    ).setParameter("param1", this.getUser().getId())
     .getResultList();
  }
}
```

Queries: Syntax and Type Checking

Syntax

- Hibernate queries are composed as strings and parsed at run-time
- In WebDSL query is parsed by the generator
  - Syntax of HQL is embedded in syntax of WebDSL
  - Generated HQL pretty-printer is used to 'generate' queries in Java code

Typechecking

- Hibernate queries are typechecked at run-time
- In WebDSL query is checked against entity declarations and local variables used as parameters (under construction)

Part X

Not all abstraction can be generative

Demo Preparation

Prototyping experiment

- Expanding the SERG webapplication on demand
- Think of a domain model to add to the website (what properties?)
- Think of the presentation and editing interface

Templates and Modules

Application programmer needs abstraction mechanisms

- Naming reusable fragments
- Avoiding code duplication
- Building a library

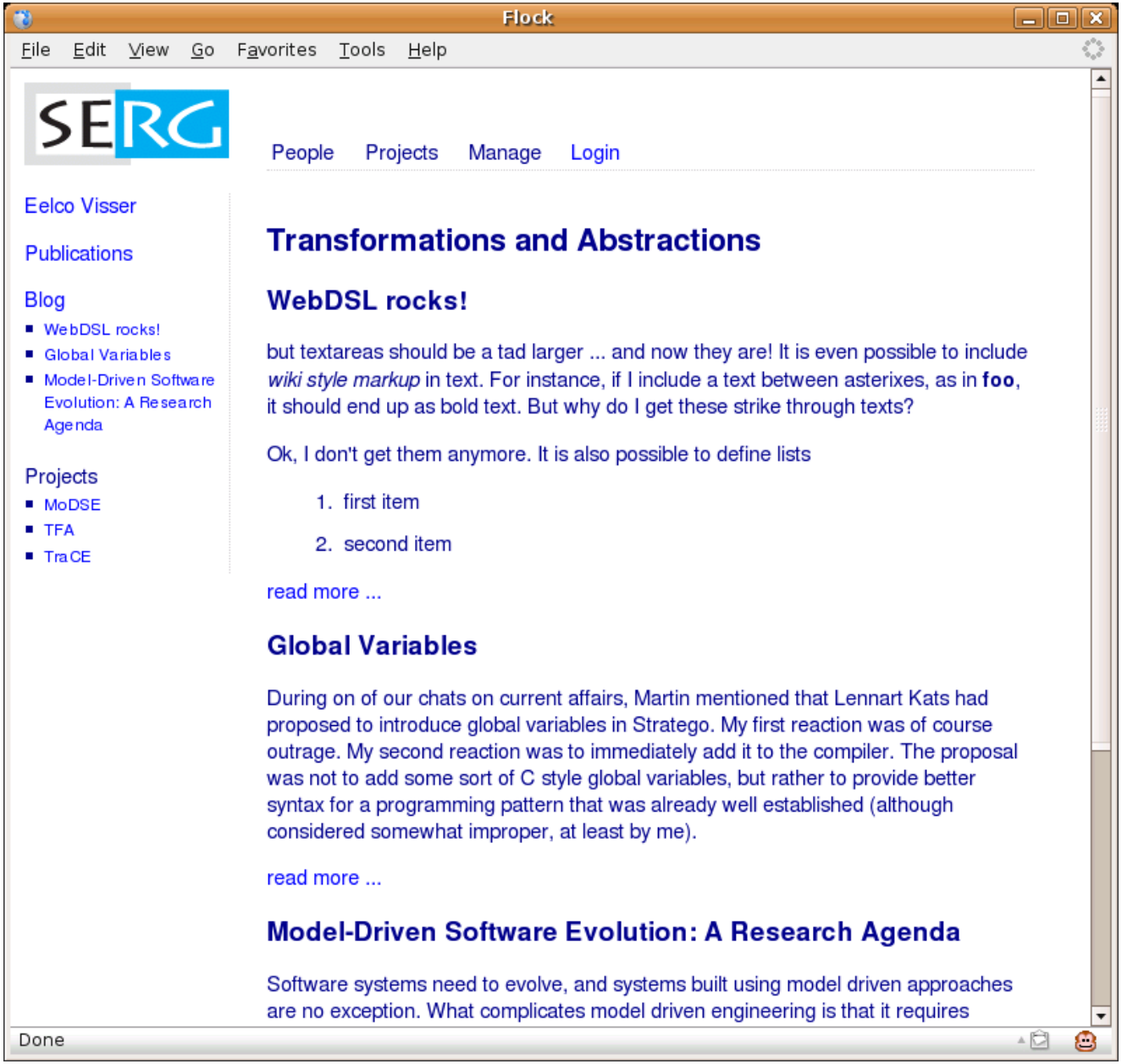
Templates

- Named pieces of code with parameters and hooks

Modules

- Organization of code base
- Library of reusable code

Consider viewBlog



Blog Domain Model

```
Blog {
  title      :: String (name)
  author     -> Person
  entries    <> List<BlogEntry>
  categories -> List<Category>
}

BlogEntry {
  blog     -> Blog
  title    :: String (name)
  created  :: Date
  category -> Category
  intro    :: Text
  body     :: Text
  comments <> List<BlogComment>
}
```

Some numbers about viewBlog

file name	LOC
Blog + BlogEntry	16
BlogEntry.java Blog.java	116 85
generated : source	201 : 16 = 12.5
viewBlog.app	91
viewBlog.xhtml	164
ViewBlogBeanInterface.java ViewBlogBean.java	32 131
generated : source	327 : 91 = 3.6

Some numbers about SERG application

file name	LOC
serg.app	983
*.xhtml	17329
*.java (ent.) *BeanInterface.java *Bean.java	1848 4069 15588
generated java	21505
generated total	38834
generated : source	39.5

Some numbers about SERG application

file name	LOC
serg.app	983
*.xhtml	17329
*.java (ent.) *BeanInterface.java *Bean.java	1848 4069 15588
generated java	21505
generated total	38834
generated : source	39.5
serg-full.app generated : source	9165 4.2

Some numbers about SERG application (revisited)

file name	LOC
serg.app	983
serg-full.app	9165
generated : source	9.3
generated total	38834
generated : source	4.2

Some numbers about SERG application (revisited)

file name	LOC
serg.app	983
serg-full.app generated : source	9165 9.3
generated total	38834
generated : source	4.2

Basic WebDSL

- Reduce code size to 25%

WebDSL with model-to-model transformations

- Reduce code size to 2.5%
- By means of template expansion and desugaring

Note

- These numbers are not definitive; full blown application will require more DSL code

Templates: Reusing Page Fragments

Define a fragment once

```
define logo() {
  navigate(home()){image("/img/serg-logo-color-smaller.png")}
}
define footer() {
  "generated with "
  navigate("Stratego/XT", url("http://www.strategoxt.org"))
}
define menu() {
  list{ listitem { "People" ... } } ...
}
```

Reuse fragment in many pages

```
define page home() {
  block("menubar"){ logo() menu() }
  section{ ... }
  footer()
}
```

Templates with Hooks

Template definition calls other templates

```
define main() {
  block("outersidebar") { logo() sidebar() }
  block("outerbody") {
    block("menubar") { menu() }
    body()
    footer()
  }
}
```

(Re)define hook templates locally

```
define page viewBlog(blog : Blog) {
  main()
  define sidebar(){ blogSidebar(blog) }
  define body() {
    section{ header{ text(blog.title) }
      for(entry : BlogEntry in blog.entries) { ... }
    }
  }
}
```

Templates with Entity Parameters

Pass objects to template definitions

```
define personSidebar(p : Person) {
  list {
    listitem { navigate(p.name, viewPerson(p)) }
    listitem { navigate("Publications", personPublications(p)) }
    listitem { navigate("Blog", viewBlog(p.blog)) blogEntries() }
    listitem { "Projects" listProjectAcronyms(p) }
  }
}
```

Reuse same template in different contexts

```
define page viewPerson(person : Person) {
  main()
  define sidebar() { personSidebar(person) } ...
}
define page personPublications(person : Person) {
  main()
  define sidebar() { personSidebar(person) } ...
}
```

Template Expansion

```
declare-template-definition =
  ?def@|[ define mod* x(farg*){elem*} ]|
  ; rules( TemplateDef : x -> def )

expand-template-call :
  |[ x(e*){elem1*} ]| -> |[ block(str){elem2*} ]|
  where <TemplateDef;rename>x => |[define mod* x(farg*){elem3*}]|
      ; {| Subst
         : <zip(bind-variable)> (farg*, <alltd(Subst)> e*)
         ; elem2* := <map(expand-element)> elem3*
         ; str := x
         |}

bind-variable =
  ?(Arg(x, s), e); rules( Subst  : Var(x) -> e )
```

Template Expansion: A Trail of Blocks

```
define page viewBlog(blog : Blog) {
  main()
  define sidebar(){ ... }
  define body() { ... }
}
```

Expands to

```
define page viewBlog(blog : Blog) {
  block("main"){
    block("outersidebar") {
      block("logo"){ ... } block("sidebar"){ ... }
    }
    block("outerbody") {
      block("menubar") { block("menu") { ... } }
      block("body") { ... } block("footer") {  }
    }
  }
}
```

Trail of template expansion can be used in stylesheets

Module Definitions

```
module publications
section domain definition.

  Publication {
    title    :: String (name)
    subtitle :: String
    year     :: Int
    pdf      :: URL
    authors  -> List<Person>
    abstract :: Text
    projects -> Set<ResearchProject>
  }
section presenting publications.

 define showPublication(pub : Publication) {
    for(author : Person in pub.authors){
      navigate(author.name, viewPerson(author)) ", " }
    navigate(pub.name, viewPublication(pub)) ", "
    text(pub.year) "."
  }
```

Module Imports

```
application org.webdsl.serg

description
  This application organizes information relevant for a
  research group, including people, publications, students,
  projects, colloquia, etc.
end

imports app/templates
imports app/people
imports app/access
imports app/blog
imports app/colloquium
imports app/publications
imports app/projects
imports app/groups
imports app/news
imports app/issues
```

Module System Implementation

A simple module systems costs as little as 11 LOC

```
import-modules =
  topdown(try(already-imported <+ import-module))

already-imported :
  Imports(name) -> Section(name, [])
  where <Imported> name

import-module :
  Imports(name) -> mod
  where mod := <xtc-parse-webdsl-module>FILE(<concat-strings>[name, "
      ; rules( Imported : name )
```

But then you don't get separate compilation

Part XI

More Sugar, Please!

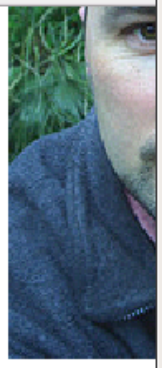
Higher-Level Language Constructs aka Syntactic Sugar

- An assesment of WebDSL
  - \+ flexibility
  - \- some patterns tedious to encode
- Solution
  - identify common patterns
  - define higher-level constructs (syntactic sugar)
  - implement using desugaring transformation rules
  - aka model-to-model transformations
- Examples
  - links to entities
  - editing associations
  - edit pages

Output: Entity Links

Eelco Visser - Flock

View Go Favorites Tools Help



email visser@acm.org

address Mekelweg 4
Delft

phone +31 (015) 27 87088

Publications

- Model-Driven Software Evolution: A Research Agenda (2007)
- Domain-Specific Language Engineering (2007)
- Grammar Engineering Support for Precedence Rule Reco Compatibility Checking (2007)
- Preventing Injection Attacks with Syntax Embeddings (200
- Transformations for Abstractions (2005)

Projects

- Model-Driven Software Evolution (MoDSE)
- Transformations for Abstractions (TFA)
- Capturing Timeline Variability with Transparent Configurat (TraCE)

generated with Stratego/XT

0.1:8080/serg/viewPublication.seam?publication=4

Flock

File Edit View Go Favorites Tools Help

SERG

People Projects Manage Login

Transformations for Abstractions

Title : Transformations for Abstractions

Subtitle :

Year : 2005

Pdf : http://www.cs.uu.nl/research/techreps/repo/CS-2005/2005-034.pdf

Authors

- Eelco Visser

Abstract

The transformation language Stratego provides highlevel abstractions for implementation of a wide range of transformations. Our aim is to integrate transformation in the software development process and make it available to programmers. This requires the transformations provided by the programming environment to be extensible. This paper presents a case study in the implementation of extensible programming environments using Stratego, by developing a small collection of language extensions and several typical transformations for these languages

Done

Output: Entity Links

Pattern

```
navigate(viewPublication(pub)){text(pub.name)}
```

Abstraction

```
output(pub)
```

Desugaring rule

```
DeriveOutputSimpleRefAssociation :
  |[ output(e){} ]| -> |[ navigate($viewY(e)){text(e.name)} ]|
  where SimpleSort($Y) := <type-of> e
      ; <defined-java-type> SimpleSort($Y)
      ; $viewY := <concat-strings>["view", $Y]
```

Enabled by type annotations on expressions

Output: Other Type-Based Desugarings

Similar desugaring rules

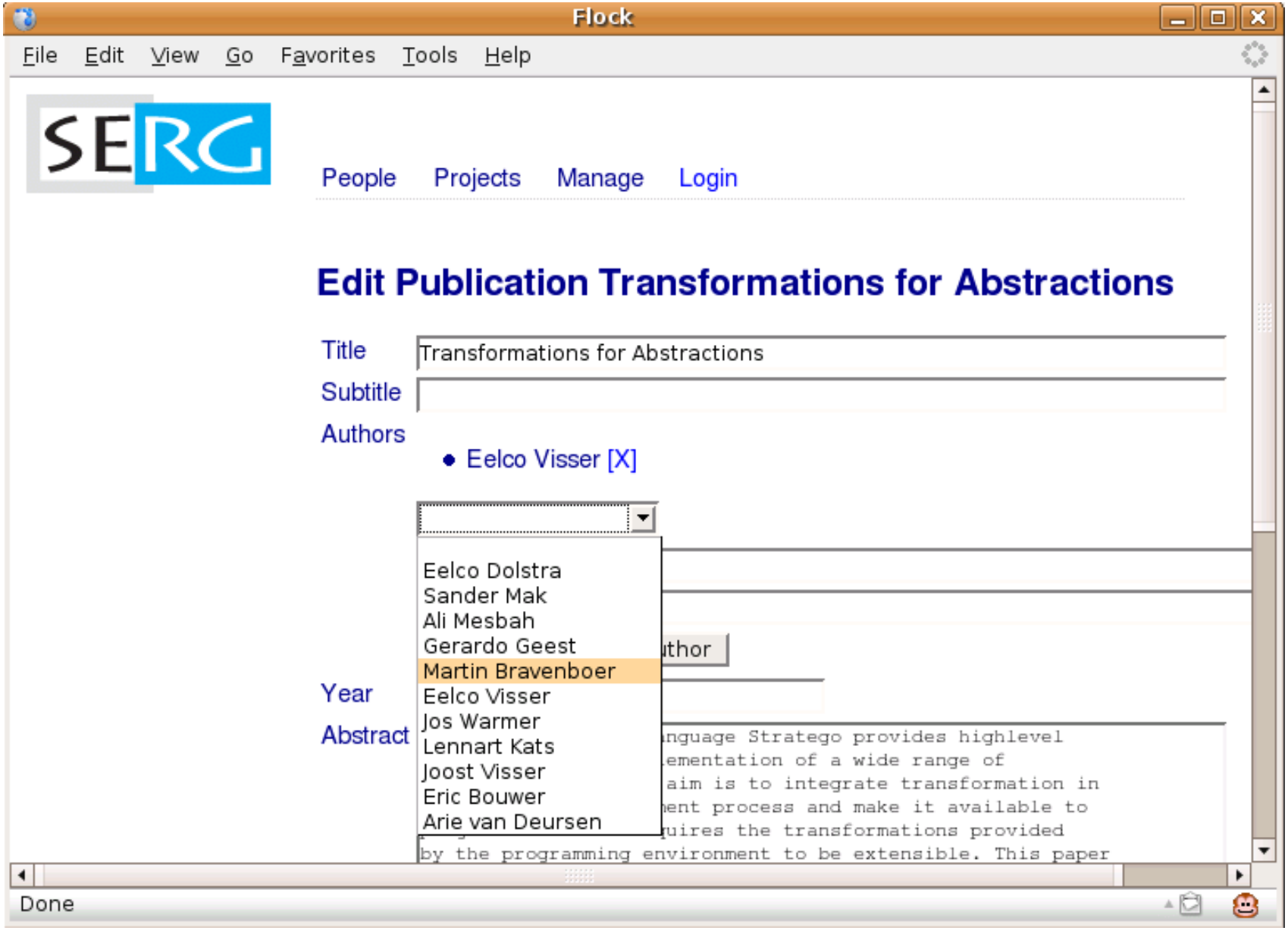
```
DeriveOutputText :
  |[ output(e){} ]| -> |[ navigate(url(e)){text(e)} ]|
  where SimpleSort("URL") := <type-of> e

DeriveOutputText :
  |[ output(e){} ]| -> |[ image(e){} ]|
  where SimpleSort("Image") := <type-of> e
```

Consequence

- `output(e)` sufficient for producing presentation

Input: Editing Entity Collection Associations



Input: Editing Entity Collection Associations

Ingredients

- List of names of entities already in collection
- Link to remove entity from collection [X]
- Select menu to add a new (existing) entity to collection

Pattern

```
list { for(person : Person in publication.authors) {
  listitem{ text(person.name) " "
            actionLink("[X]", removePerson(person)) }
} }
select(person : Person, addPerson(person))

action removePerson(person : Person) {
  publication.authors.remove(person);
}
action addPerson(person : Person) {
  publication.authors.add(person);
}
```

Input: Editing Entity Collection Associations

Desugaring rule

```
DeriveInputAssociationList :
  elem|[ input(e){} ]| ->
  elem|[
    block("inputAssociationList"){
      list { for(x : $X in e){ listitem {
        text(x.name) " "
        actionLink("[X]", $removeX(x))
        action $removeX(x : $X) { e.remove(x); }
      }} }
      select(x1 : $X, $addX(x1))
      action $addX(x : $X) { e.add(x); }
    }
  ]|
  where |[ List<$X> ]| := <type-of> e
      ; x         := <decapitalize-string; newname> $X
      ; x1        := <decapitalize-string; newname> $X
      ; $viewX    := <concat-strings>["view", $X]
      ; $removeX  := <concat-strings; newname>["remove", $X]
      ; $addX     := <concat-strings; newname>["add", $X]
```

Input: Editing Entity Collection Associations

Similar desugaring rules

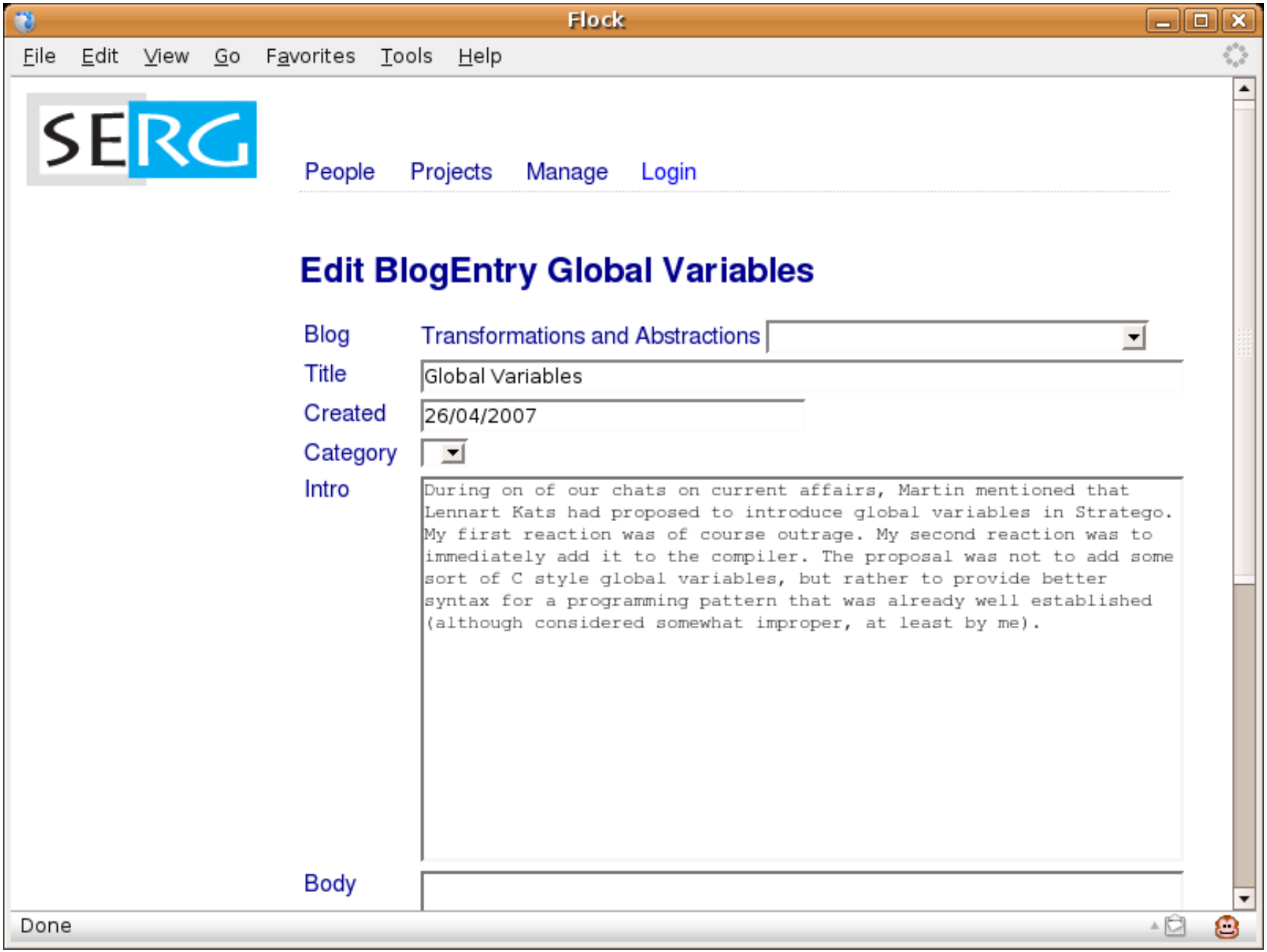
```
DeriveInputText :
  |[ input(e){} ]| -> |[ inputText(e){} ]|
  where SimpleSort("Text") := <type-of> e

DeriveInputSecret :
  |[ input(e){} ]| -> |[ inputSecret(e){} ]|
  where SimpleSort("Secret") := <type-of> e
```

Consequence

- `input(x.y.z)` suffices for producing input of property

Edit Page


Flock
File Edit View Go Favorites Tools Help
SERG
People Projects Manage Login
Edit BlogEntry Global Variables
Blog
Transformations and Abstractions
Title
Global Variables
Created
26/04/2007
Category
Intro
During on of our chats on current affairs, Martin mentioned that Lennart Kats had proposed to introduce global variables in Stratego. My first reaction was of course outrage. My second reaction was to immediately add it to the compiler. The proposal was not to add some sort of C style global variables, but rather to provide better syntax for a programming pattern that was already well established (although considered somewhat improper, at least by me).
Body
Done

Edit Page for Entity

Ingredients

- Input box for each property of an entity organized in a table
- Save and Cancel buttons

Pattern

```
form {
  table {
    row{ "Blog"     input(entry.blog) }
    row{ "Title"    input(entry.title) }
    row{ "Created"  input(entry.created) }
    row{ "Category" input(entry.category) }
    row{ "Intro"    input(entry.intro) }
    row{ "Body"     input(entry.body) }
  }
  action("Save", save()) action("Cancel", cancel())
  action cancel() { return viewBlogEntry(entry); }
  action save() { entry.save(); return viewBlogEntry(entry); }
}
```

Edit Page for Entity

Desugaring rules

```
entity-to-edit-form :
  |[ $X : $Y { prop* } ]| ->
  |[
     form {
       table { elem* }
       action("Save",   save())
       action("Cancel", cancel())
     }
     action cancel() { return $viewX(x); }
     action save() { x.save(); return $viewX(x); }
  ]|
  where $viewX := <concat-strings>["view", $X]
      ; x      := <decapitalize-string> $X
      ; str    := $X
      ; elem*  := <map(property-to-edit-row(|x))> prop*

property-to-edit-row(|x) :
  |[ y k s (anno*) ]| -> |[ row { str input(x.y) } ]|
  where str := <capitalize-string> y
```

DSL Design: Balance between Salt and Sugar

Salt (core language)

- low-level constructs guarantee sufficient expressivity
- completeness: can everything (in the domain) be expressed?

Sugar (syntactic abstractions)

- high-level constructs support high productivity
- completeness: conceptually easy things should be *easily* expressable

Part XII

Demonstration

Part XIII

Implementation

The WebDSL Generator

Transformation pipeline

- Parsing
- Importing modules
- Desugaring
- Declaring definitions
- Typechecking (also of embedded queries)
- Template expansion
- Derivation
- Code generation (JPA/Hibernate + Seam + JSF)
- Write code models to file

Implementation / metrics

- Implemented in Stratego/XT
- Rewrite rules with concrete syntax
- Time: first commit March 8, 2007 (3 months / 1 week ago)
- At most 50% spent on DSL

The WebDSL Generator

Syntax

```
1081 HQL.sdf                 // migrated from antlr grammar (included)
  44 MixHQL.sdf              // generated
   9 StrategoWebDSL.sdf
  86 WebDslMix.sdf           // generated
 215 WebDSL.sdf
1435 total
```

Generator (129 rules)

```
 271 dsl-to-seam.str
 109 generator.str
 280 java-code.str
 234 java-Entity.str
 432 java-page.str
  49 java-utils.str
 507 xhtml-page.str
1882 total
```

Transformations (166 rules)

```
 591 desugar.str
 194 expand-page.str
 116 java-types.str
 112 register-declarations.str
 524 types.str
1537 total
```

Utils 380 LOC - should be in library

Part XIV

Unfinished Business

Modeling Web Applications

Implementation is no longer an obstacle

- Easy to try alternative scenarios

Domain modeling

- Coupling
- Inverse associations or queries
- Roles
- Subtyping
- ...

Interaction modeling

- UI design
- Interaction patterns
- ...

Modeling Web Applications: DSL expressivity

Completeness of WebDSL

- Loose ends
  - Pagination of query results
  - Collections of value types
  - Punctuation in generated output (commas, delimiters, ...)
  - Better URLs
- More default interaction patterns
  - Identify styles of interaction and generate good defaults
  - In particular associations
- Rich(er) userinterface
  - Integration of iteration with UI components
  - Using AJAX JSF components
  - Single page user interface (e.g. using Echo2) (Jonathan Joubert)

Modeling Web Applications: DSL expressivity

Completeness of WebDSL

- Input validation and conversion
- Security
  - authentication and access control (Danny Groenewegen)
  - Preventing injection attacks (seems to be covered well by base frameworks?)
- Workflow: business process modeling
- and of course: business logic
  - what is needed? (what is business logic, by the way?)

Engineering

- Testing of WebDSL applications

DSL Design and Implementation

Implementation of WebDSL

- Pretty-printed error messages (instead of dumping terms)
- Templates that abstract over template element (not only via hooks)
- Fully typechecking HQL expressions
- Easier name mangling with guaranteed consistency (?)
- Optimization of database queries

General Concerns

- DSL interaction and separate compilation (Sander Mak)
  - modular typechecking, template expansion, ...
  - generate modular code (depends on target platform)
- Reusable framework for DSL implementation
  - parameterized with syntax definition
  - organizes main generator pipeline
  - generation of multiple files
  - import chasing

Programming Environment

IDEs for DSLs

- New DSL not supported by IDE (Eclipse)
- Generate Eclipse plugin from language definition
  - syntax highlighting
  - syntax checking
  - typechecking
  - refactoring
  - ...
- Integrate Stratego/XT with Safari (IBM)

Visualization

- Visual views
  - class diagrams
  - page flow diagrams
- Editing via visual views?

Deployment

Status

- Generation of JSF and Java source files
- Skeleton of application source tree generated by seam-gen
- Manual build steps
  - .app to code (make)
  - code to .war/.ear (ant)
  - activation of database & webserver

Future

- Generate complete source tree
- Integrate building of the source tree (build .war file)
- Automatic deployment and activation of the webserver
- WebDSL virtual machine
  - drop `foo.app` and activate
  - server takes care of code generation, deployment, activation
  - using Nix deployment system

Evolution

Data conversion

- Adapting entity declarations leads to new database scheme
- Convert data in old database to new one
- Define relation mapping old entities to new ones
- Generate scripts for existing tools?

Model migration

- Changing DSL definition requires adapting existing models

Abstraction evolution

- Model sweetening: apply new sugar to old models

Harvesting from legacy code

- Transform legacy EJB applications to WebDSL?
- JSF to page definitions
- Entity classes to entity declarations
- Session beans to actions

Summary: Properties of a good DSL

- Core language that covers needed domain expressivity
- Syntactic extensions that allow concise expression
- Facilities to build a library
  - Modules for organization of code base
  - Parametric abstraction over DSL fragments

Summary: How to develop a DSL?

- Choose high-level technology
  - DSL should not readdress problems already solved by technology
- Start with large chunks of programs
  - Understand the technology
  - Recognize common patterns
- Setup a basic generator early on
  - makes it easy to experiment with alternative implementation strategies
- Don't try to find core language from the start
  - result may be too close to target
  - e.g., modeling language that covers all EJB concepts
- Don't over specialize syntax
  - template call vs header, section, ... as constructs
- Don't over generalize syntax (XML)

Future

- Extend WebDSL (see ideas before)
- Apply to industrial case studies
- Abstractions for application (business) domains?
  - finance, insurance, ...
- Repeat exercise for other domains
- Develop systematic method for building new modeling languages