Building Domain-Specific Languages with Stratego/XT

Eelco Visser, Lennart Kats

Delft University of Technology

http://www.strategoxt.org

Code Generation 2009
Cambridge
June 16, 2009

Domain-Specific Languages

What?

- Language targeted at technical or application domain

Why?

- Scrap the boilerplate
- Increase expressivity
- Use domain-specific abstractions
- Find errors earlier
- Avoid common errors
- Encapsulate implementation knowledge

How?

- Design language
- Syntax: what are well-formed models
- Transformations: transform to other models (code)
- This tutorial: SDF + Stratego/XT

WebDSL

A domain-specific language for web applications

- Data models
- Page definition
- Forms with automatic data binding
- Actions
- Declarative access control rules
- Workflow procedures and process descriptions

NanoWebDSL

Very small subset of WebDSL

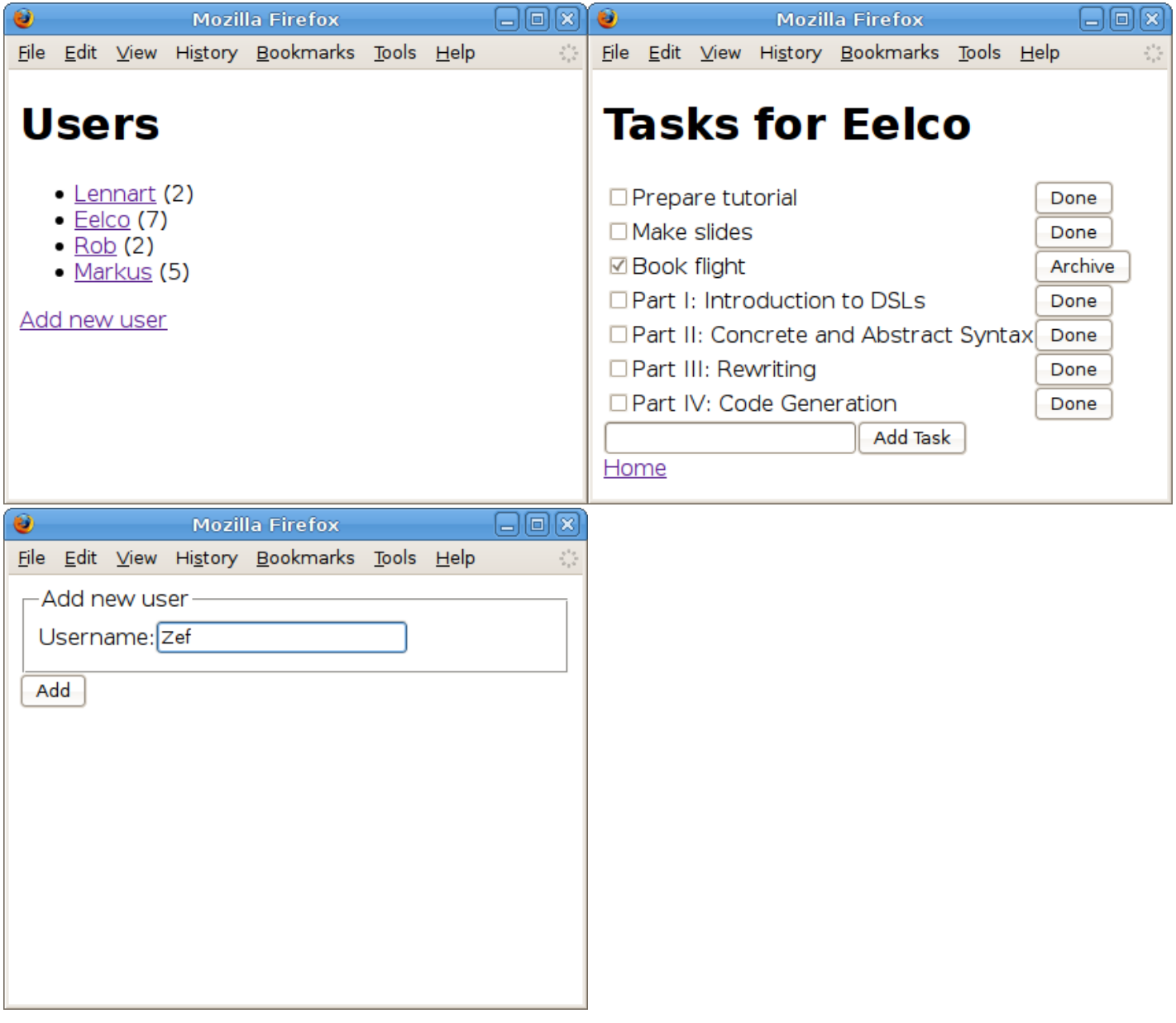
- Data models
- Page definition
- Forms with automatic data binding
- Actions
- Declarative access control rules
- Workflow procedures and process descriptions

NanoWebDSL: Tasks Application

Tasks

- Tiny web app for issue management
- To show basic ideas of NanoWebDSL

NanoWebDSL: Tasks Application



Exercise 1.1

Build the tasks application using the nanoweb script

```
$ cd exercises/1/tasks
$ webnano build deploy
```

Start tomcat to run the application

```
$ catalina.sh run
```

NanoWebDSL: Data Models

```
entity User {
  username :: String (id,name)
  tasks    -> List<Task>
  log      -> List<Task>
}

entity Task {
  description :: String
  done        :: Bool
}
```

NanoWebDSL: Page Definitions

```
define page home() {
  section{
    header{"Users"}
    list{
      for(user : User) {
        listitem{
          navigate(tasks(user)){output(user.username)}
          " (" output(user.tasks.length) ")"
        }
      }
    }
    navigate(newuser()){"Add new user"}
  }
}
```

NanoWebDSL: Forms

```
define page newuser() {
  var user : User := User {}
  form{
    group("Add new user") {
      derive editRows from user for ( username )
    }
    action("Add", add())
    action add() {
      user.save();
      return tasks(user);
    }
  }
}
```

NanoWebDSL: Actions

```
define page tasks(user : User) {
  var newTask : Task := Task{ done := false }
  action addtask() {
    user.tasks.add(newTask);
    newTask.save();
  }
  action done(task : Task) {
    task.done := true;
    task.save();
  }
  action archive(task : Task) {
    user.tasks.remove(task);
    user.log.add(task);
    user.save();
  }
  ...
```

NanoWebDSL: Actions

```
define page tasks(user : User) { ...
  section{
    header{"Tasks for " output(user.username) }
    table{
      for(task : Task in user.tasks) {
        row{
          output(task.done)
          output(task.description)
          form{
            if(!task.done) {
              action("Done", done(task))
            } else {
              action("Archive", archive(task))
            } } } } }
    form{
      input(newTask.description)
      action("Add Task", addtask())
```

Outline

- Part I: Introduction
- Part II: Concrete and abstract syntax
- Part III: Rewriting
- Part IV: Code generation