MontiWeb – Modular Development of Web Information Systems

Michael Dukaczewski¹, Dirk Reiss¹, Bernhard Rumpe², Mark Stein¹

¹ Inst. f. Wirtschaftsinformatik, Technische Universität Braunschweig
² Software Engineering, RWTH Aachen

9th OOPSLA Workshop on Domain-Specific Modeling
25-26 October 2009
Outline

- Introduction + Motivation
- Technical Infrastructure
- General Architecture
- Modeling Languages
- Conclusion + Future Work
Introduction

- Last 3 years working on a project initiated by TU Braunschweig
- Focus: Developing and customizing (web-based) applications for teachings and administration
- Developing with different languages and frameworks (depending on the existing infrastructure and requirements)

- Many different applications, still the same patterns and work …
Web Information Systems

- Our understanding of the domain:
  - Used to process data
  - HTML form based
  - Usually same layout and similar behavior

- Web information systems usually consist of
  - Data structure / Persistence mechanisms
  - Views on data structure
  - Navigation / workflow logic between these views

- Implementation often
  - Repetitive work
  - Repeating components
Traditional Approach

- Definition of the **same element at different parts** of a system
  - **Source code** (in e.g. classes)
  - **Database** (in tables and rows)
  - **GUI elements** in HTML / JSP form
  - Potentially **glue code** in XML files
  - All mostly **dependent** but still **not integrated**

- Changes need to be made on all parts
- Lots of **boilerplate** code
- **Consistency** checked often at runtime
MontiWeb Approach

- **Raising abstraction** from the implementation details
- **Models** to specify the elementary parts, actually
  - Data structure
  - Views
  - Control- and dataflow
- **Goal**: Keeping these aspects separate to allow reuse in different contexts

- Generators create **working prototypes**
  - Basic models already enough to generate CRUD application
  - Additional models to add more fine grained functionality
- Using **textual models** specified using MontiCore framework
MontiCore - Modeling Framework Infrastructure

- Framework for the efficient development of DSLs
- Developed at Software Systems Engineering Institute of TU Braunschweig and now RWTH Aachen
- Extended grammar format for language definition
- Generates components for the processing of models such as
  - Parsers
  - AST classes
  - Basic symbol tables
  - Pretty printers
  - Basic editor support
- Provides infrastructure to conveniently access and use the generated components
Architecture Overview

MontiCore Grammar

defines

Class-diagramm
Views

Activity-diagramms

input

generates

MontiWeb

Parser
Generator

Web-Application

input

generates

Web-Application
Modeling Data Structure

- **Requirements** for a data model in web information system (according to our experience)
  - Incorporates a *type system* (with domain-specific behavior)
  - Is *composable* (for reuse of elements)
  - Can have *associations* between model elements

- Textual representation of *class diagrams* as modeling language
  - Generally *well known* and understood
  - Expressive enough to fulfill the abovementioned requirements
Types of Classes

- **Base classes** (e.g. Email, Date, String, Number)
  - Do not contain further attributes
  - Usually domain-specific (or at least often used in that domain)
  - **Standard behavior** in the target domain (e.g. consistency checks, special input methods)

- **Enumerations**
  - Can hold static values and be used as attributes

- **Complex classes**
  - Consist of base classes, enumerations or other complex classes
Associations between Classes

- **Normal associations**
  - Represent **links** between two objects A and B
  - A and B need to **exist** (or one is just created)
  - Implemented by (multi-)selection mechanisms

- **Compositions**
  - Represents **part-whole association** between A and B
  - If A is composed of B, B exists only in combination with A
  - Implemented by simultaneous creation
    - B is created when A is created
    - B is deleted when A is deleted
Data Model

- Example: Very basic carsharing application

```java
classdiagram Carsharing {
  class Person {
    String name;
    Email email;
    Number age;
  }

  enum Brand {AUDI, BMW, VW;}

  class Car {
    Brand brand;
    Number numSeats;
    Date constYear;
  }

  composition Person (keeper) -> (cars) Car [*];
}
```
Modeling View Structure

- Requirements for a view language
  - Different views on the same data structure (e.g. edit, display)
  - Views can be composed and included in each other
  - Static parts (e.g. images, text) are possible
  - Convenience functionality (e.g. filtering, sorting) can specified

- Own language that fulfills these requirements
- Optional; if omitted, default views are generated

- Focus of the view language:
  - Generation of usable and consistent layout
  - Skinable through later inclusion of different CSS and a basic template mechanism
view structure

Person {
  attributes {
    @Required
    @Length(min=3, max=30)
    name;
    @Required
    age;
  }
  @Captcha
  editor registration {
    name;
    email;
    age;
    cars;
  }
  // ...
}
View Structure

Person {

  // ...

  display protectedMail {
    name;
    @AsImage
    email;
  }

  display welcome {
    text {Welcome to Carsharing Service}
    include protectedMail;
    age;
  }
}

Welcome
Welcome to Carsharing Service

<table>
<thead>
<tr>
<th>Name</th>
<th>Reiss</th>
</tr>
</thead>
<tbody>
<tr>
<td>Email</td>
<td><a href="mailto:d.reiss@tu-bs.de">d.reiss@tu-bs.de</a></td>
</tr>
<tr>
<td>Age</td>
<td>32</td>
</tr>
</tbody>
</table>

includes previously defined view
Modeling Control- and Dataflow

- Basic control can be generated from view or even classes alone
- Standard way: Class diagram to CRUD application with named standard views
- For more complex web information systems, we need means to specify
  - Order of pages
  - Flow of data between pages
  - Complex workflow logic

- Textual notation of activity diagrams
- Actually inclusion of views and Java code supported
- Hierarchical actions and most common control structures (decisionnodes, forks etc) supported
Control- and Dataflow

activity UserRegistration {

    action Registration {
        out: Person p;
        view : p = Person.registration();
    }

    action Welcome {
        in: Person p;
        view : Person.welcome(p);
    }

    action Error {
        in: Person p;
        view : Person.registrationError(p);
    }

    initial -> Registration;
    Registration.p -> [p.age >= 18] Welcome.p \\
                     | [p.age < 18] Error.p;
    Welcome | Error -> final;
}

holds the entered object

reference to a view

reference to a view
Interaction of Components

- Models are specified independently but partially rely on each other.
- Classviews reference class diagram attributes by name.
- Activity diagram references:
  - Classviews (to display them)
  - Classes (as data type)
- Therefore: Reuse of different parts of the system in different contexts possible.
- Intra- and intermodel correctness is checked on model level during generation.
Conclusion

- MontiWeb allows **modeling of data-intensive web information systems**
- Working web application even with **minimal model** through **default behavior**
- **Advanced behavior** specifiable through **additional models**
- DSL designed by reusing known concepts and languages (UML, Java)
- **Language concepts** so far **suitable** for the web information systems domain
Future Work

- Incorporation of means to model rights and roles system and access control
- Modeling global features and roles with use case diagrams
- More complete use of language features
  - Inheritance in class diagrams
  - Inclusion of method stubs in classes
- Extend base classes to include more predefined datatypes
- Generation of interfaces to use the generated code from handwritten classes (or other generated code)
- Means to pack models and source code to component libraries
Thanks for your attention!

Questions?