KSU STUDENT PORTAL

MSE Project Presentation 2.0

Javier Ramos Rodríguez
OUTLINE

- Introduction
- Architecture Design
  - Dynamic view
    - Use Case Specification
    - Model Representation
    - Sequence Diagram
    - Collaboration Diagram
  - Static View
    - Class Diagram
- Data Model
  - E-R Diagram
  - UML Data Model
  - Tables
OUTLINE

- Formal Specification
  - Alloy Model
  - USE Model
- Inspection Checklist
- Project Plan
- Test Plan
- Conclusion
- References
Introduction

- Goals of this phase:
  - High Level description of the system.
  - Precise.
  - No Implementation details.
  - Portable.

- The Idea is to use an iterative process throughout the development to create a less expensive product. And at the same time increasing the quality, reusability and modularity.
Introduction

Phase 1
REQUIREMENTS

Phase 2
PLATFORM INDEPENDENT MODEL

Phase 3
IMPLEMENTATION INDEP. MODEL
IMPL. MODEL
IMPL. MODEL
IMPL. MODEL
## Architecture Design

### Dynamic View

<table>
<thead>
<tr>
<th>Name</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective</td>
<td>Register a new user in the application, so he/she can access to the main features.</td>
</tr>
<tr>
<td>Priority</td>
<td>High</td>
</tr>
<tr>
<td>Extends</td>
<td></td>
</tr>
<tr>
<td>Includes</td>
<td></td>
</tr>
<tr>
<td>Actors</td>
<td>User</td>
</tr>
<tr>
<td>Pre-conditions</td>
<td></td>
</tr>
<tr>
<td>Post-conditions</td>
<td>User will get registered in the application.</td>
</tr>
</tbody>
</table>
| Main Scenario | 1. The user selects “Register” from the user interface.  
2. The system receives the request.  
3. The system shows the user the register form.  
4. User fills the form.  
5. System validates the form.  
6. The system receives the information and saves it. |
| Secondary Scenario |                                                            |
| Exceptional Scenario | If there's an error in the register form the system will tell the user to correct it. |
| Function      | Register User                                                             |
Model Representation

- **Entity Classes**: These types of classes are used to represent entities that contain data. Usually they will match to tables in the relational model or entity beans when using the J2EE platform.

- **Control Classes**: These types of classes are used to represent classes that perform some kind of logic: business logic or controllers. In the view layer these classes will represent the controllers in the MVC pattern; in the model these classes will represent the business logic. For example, in the J2EE platform these classes probably will be session beans.

- **Boundary Classes**: These classes in the view are used to represent the classes that the user interacts with. These classes will contain all the forms and all the other UI components. So, these classes are the boundary between users and the application. For example, in a J2EE web application these classes will be JSP pages. In the model, this stereotype is used to represent the boundary between the view and the model, so it will be used in the class that implements the facade design pattern.
Sequence Diagram
Register

register
register() showRegForm()
fill form submit()
validateFields()
[NO fields properly filled]
showRegForm(String error)
[fields properly filled]
correct form
showRegForm(String error)
[fields properly filled]
registerUser()
registerUser() <<create>>
persist(user)
showResults(String results)
showResults(String results)
showResults(String results)
Sequence Diagram

Browse Events

- Registered User
- WelcomeUI
- WelcomeUIController
- ModelFacade
- UserController
- EventControl
- EventListUI
- events : List
- event : Event
- profile : UserProfile

browse events

browseEvents()  getEvents(login)

getUserProfile(login)  returnResults(profile)

getEvents(profile)  query the DB only getting the event according to the profile

<<create>>  showResults(events)

[repeat]

add(event)

[while ResultSet.isEmpty == true]

showEventList(events)
Collaboration Diagram

1: register
2: register()
3: showRegForm()
4: fill form
5: submit()
6: validateFields()
7: showRegForm(String error)
8: correct form
9: registerUser()
10: registerUser()
11: <<create>>
12: persist(user)
13: showResults(String results)
14: showResults(String results)
15: showResults(String results)

User
: User

MainUI
: MainUI

RegisterUI
: RegisterUI

ModelFacade
: ModelFacade

UserController
: UserControl

MainUIController
: MainUIController

RegisterUIController
: RegisterUIController

AppUser
: user

<<create>>

Persist
: <<create>>
STATIC VIEW
Class Diagram
DATA MODEL
E-R Diagram
UML Data Model
Data Tables

- **USER**: login, password, f_name, l_name, email, street, city, state, visibility, about, p_id
- **PROFILE**: p_id, h_univ, mayor, co_code
- **BLOG**: b_id, visibility, p_id, u_id
- **BLOG_ENTRY**: be_id, picture, text, be_date, b_id
- **MESSAGE**: m_id, subject, m_text, m_date, m_author
- **MESSAGE_DEST**: m_id, login
- **COURSE**: c_num, c_name
- **ARTICLE**: a_id, a_header, a_text, a_url, a_date, a_author, f_id
- **EVENT**: e_id, e_header, e_text, e_date, e_author, f_id
- **LINK**: l_id, link, l_date, l_author
- **FILTER**: f_id, h_univ, mayor, co_code, lan_code, c_num
- **PROFILE_COURSE**: p_id, c_num
- **PROFILE_LANGUAGE**: p_id, lan_code
- **COUNTRY**: co_code, country
- **LANGUAGE**: lan_code, language
Formal Specification

- **Alloy**: Non-Deterministic modeling language based on Sets and their relations.

- **Items:**
  - **Domains**: Here we define the possible values that the sets can take.
  - **Sets**: Unordered collections of objects that take values from the domains.
  - **Relations**: Relationships between sets
  - **Multiplicities**: Constrains on the number of objects that participate in a given relation.
  - **Invariants**: Constrains that the model should hold.
  - **Assertions**: Are useful to check if a specific invariant holds. If the Alloy Constraint Analyzer finds a counterexample this means that the invariant is incorrect.
  - **Operations**: Functions that the system can execute. We define pre, post and frame conditions.
Alloy Model

- We will model the User, Blog, Blog Entries, Messages, Events, Articles, Links, Profile and other attributes.

- **Associations:**

  1. Users can have only one Blog and a Blog corresponds to exactly one user
     
     ```
     user_blog(~blog_user): static User! -> static Blog!
     ```

  2. A blog can have several entries and one entry belongs to one blog
     
     ```
     blog_entries(~entry_blog): static Blog! -> BlogEntry
     ```

- **Operation:**

  ```
  op createEvent(e:Event!, author:User!, f:Filter!){
  e !in Event
  f !in Filter
  Filter'=Filter+f
  Event'=Event+e
  e.event_filter' = f
  e.event_published'= author
  Message'=Message
  User'=User
  Profile'=Profile
  }
  ```
USE Model

- Formally checks the UML diagrams.
- Specifically class diagrams.
- Deterministic approach.
- OO Environment (not only Sets).
  - We will use the OCL language to specify constraints in the UML classes and its relations.
USE Model

model MSEProject
class User
  attributes
    login:String
end
association UserBlog between
  User[1] role blogUser
  Blog[1] role userBlog
End
association MessageDest between
  Message[1] role message
  User[*] role dest
end
association PublishEvent between
  User[1] role eventPublished
  Event[*] role publishEvent
end
USE Model
Inspection Checklist

- UML Diagrams
  - Class Diagrams
  - Sequence Diagrams
  - Collaboration Diagrams
  - Class descriptions

- Formal Specification
  - Alloy Model
  - Use Model

- Data Model
  - E-R Diagram
  - Tables
Inspection Checklist

**Approach:**

- UML diagrams conform to the UML standards.
- UML diagrams correspond to the description in the Architecture Design document.
- The different diagrams are consistent with each other.
- The constraints in the Alloy model are well defined.
- There is a correspondence between the UML model and the Alloy model.
- The USE model describes the behavior described in the other documents.
Project Plan
Test Plan

- **Test items:** Critical Use Cases

- **Approach:**
  - Black Box Testing
  - Unit Tests
  - Integration Tests
  - Environmental Test
Conclusions

- Build a model that provides a lot of detail but at the same time is platform independent.
- This allows to reuse the model reducing cost and improving quality.
- We create:
  - Data Model
  - Business Logic Model
- Formal Specification check the correctness of both specifications.
References

- www.uml.org
- www.rational.com
- Applying UML and Patterns - An Introduction to Object-Oriented Analysis and Design and the Unified Process, Craig Larman