Specification-Oriented Construction
of Web Information Systems

Project Thesis

Submitted in Partial Fulfillment of The Requirements
for the Degree of Master of Science in
Computer Science

Alexander Lyakas

Submitted to the Senate of the Technion—Israel Institute of Technology

Heshvan, 5766 HAIFA November 2005
The Project Thesis Was Done Under The Supervision of Prof. Eliezer Kantorowitz in the Faculty of Computer Sciences

I would like to express my utmost gratitude to Prof. Kantorowitz for sharing his experience and wisdom and being much much more than an academic advisor.

# Table of Contents

**Abstract** 1

**List of Abbreviations** 3

1 **Introduction** 4
   1.1 High-Level Declarative Languages 4
   1.2 The Software Development Process 5
   1.3 Frameworks 7
   1.4 Our Project 9
   1.5 Document Outline 9

2 **Specification-Oriented Programming** 10
   2.1 Original Motivation 10
   2.2 Specification-Oriented Frameworks 11
   2.3 The Interaction Style Component 14
   2.4 Previous Work—SI 15
   2.5 SI+ 17

3 **Specification-Oriented Programming Model** 18
   3.1 System Specification 18
      3.1.1 Data Flow-Centered Use-Case Specifications 19
   3.2 Relational Database 20
   3.3 Use-Case Classes and Interaction Methods 20
   3.4 Activity Classes 22
   3.5 Communicating with the User 22
   3.6 Database Manipulation 25
   3.7 Use-Case Relationships 27
   3.8 The Use-Case Displayer 29
   3.9 Interaction Styles and Automatic User Interface Generation 30
   3.10 Managing Multiple Users 30
   3.11 Security 32
   3.12 Verification 33
   3.13 Modification and Extension 33

4 **Detailed Programming Model** 34
   4.1 Use-Case Classes 34
   4.2 Interaction Methods 34
   4.3 Communicating with the User 35
      4.3.1 The Output Activity Class 35
      4.3.2 The UserComposed Activity Class 36
4.3.3 The Select Activity Class .............................................. 37
4.3.4 The Fetch Activity Class ............................................ 38
4.4 Manipulating the Database .................................................. 39
4.4.1 Reading the Database ................................................... 40
4.4.2 Writing to the Database ............................................... 41
4.4.3 Recommended SQL Datatypes ......................................... 42
4.4.4 Database Transactions ................................................. 42
4.5 Multi-User Support ............................................................ 43
4.5.1 User-Defined Session Objects ....................................... 44
4.6 Implementing the UML <<include>> Relationship ..................... 44
4.7 Debug Printouts .................................................................. 45
4.8 WebSI Security .................................................................... 45
4.9 The TableImpl Class ............................................................ 46
4.10 The ClientProgrammerTools Class ....................................... 47

5 Example ................................................................................. 48
5.1 Use-Case Specification ......................................................... 48
5.2 Security Considerations ....................................................... 52
5.3 Database Design ................................................................. 53
5.4 Coding and Verifying the “Add Product” Use Case ..................... 56
5.5 Coding and Verifying the “Select Offers” Use Case ..................... 63
5.6 The Rest of the Code ........................................................... 71

6 The Structure of the WebSI Framework ...................................... 76
6.1 Overview ............................................................................. 76
6.1.1 The Kinds of Requests a WebSI Application Handles ............... 78
6.1.2 The Structure of the Response XML File ............................. 79
6.1.3 Generating XML Content by WebSI Components ..................... 82
6.1.4 Managing Multiple Users ................................................. 83
6.1.5 Storing Common Application Data in a ServletContext ............ 84
6.1.6 Passing Data between Various WebSI Components .................. 86
6.1.7 How an Activity Class Obtains Correct Objects ...................... 88
6.1.8 Handling One Request of a User at a Time ............................ 91
6.2 The WebSI Interaction Style Component ................................ 92
6.2.1 The Structure of the XSL Stylesheet ................................... 92
6.2.2 The form Element in the Final HTML File ........................... 95
6.2.3 Indirect Fetchers ............................................................ 97
6.2.4 Submitting the Form ....................................................... 99
6.2.5 Communication Between the Sub-Stylesheets ....................... 100
6.2.6 Combining the Sub-Stylesheets into an Interaction Style ............ 101
6.2.7 Handling Input Values Supplied by the User ........................ 101
6.3 The DBProxy Bean ............................................................. 103
6.3.1 The Remote and the Local Interface .................................. 104
6.3.2 The Home and the Local Home Interface ............................ 105
6.3.3 The DBProxyBean Class ................................................. 106
6.3.4 The DB Activity Class .................................................... 106
6.3.5 The DBVendorSpecificSupport Class ................................. 107
6.4 The Table Interface and its Implementations ............................ 108
6.4.1 The StaticCopyResultset2TableAdapter Adapter .................... 111
B.1.4 The output_elegantWideTables Style .................. 193
B.1.5 The output_sky Style .................................. 194

B.2 The Action Activity Class Stylesheets ..................... 195

B.3 The UserComposed Activity Class Stylesheets .......... 196
B.3.1 The user_composed_default Style ..................... 196
B.3.2 The user_composed_default2 Style ................... 199
B.3.3 The user_composed_elegant Style ..................... 201
B.3.4 The user_composed_sky Style .......................... 202

B.4 The Select Activity Class Stylesheets .................... 203
B.4.1 The select_default Style ............................... 203
B.4.2 The select_default2 Style ............................. 206
B.4.3 The select_default3 Style ............................. 210
B.4.4 The select_elegant Style ............................... 211
B.4.5 The select_elegant2 Style ............................ 212
B.4.6 The select_elegant3 Style ............................. 213
B.4.7 The select_elegantWideTables Style .................. 215
B.4.8 The select_sky Style .................................. 218
# List of Figures

2.1 The architecture of a use case-oriented framework .......................... 14  
3.1 WebSI multi-user support ..................................................... 31  
4.1 WebSI activity classes that support communicating with the user .... 35  
4.2 Database constraint violations exception classes ....................... 41  
4.3 WebSI multi-user support ..................................................... 43  
5.1 The use-case diagram of the “Offers” system ............................ 49  
5.2 Executing the “Add Product” use case ..................................... 61  
5.3 Executing the “Select Offers” use case ..................................... 68  
6.1 The EngineServlet class ....................................................... 77  
6.2 Handling GET and POST HTTP requests by EngineServlet .......... 77  
6.3 The WebSI LoginLogout use-case class ................................... 79  
6.4 The UseCaseDisplayer class .................................................. 80  
6.5 Java DOM API classes ......................................................... 83  
6.6 The SecurityRolesSetup class ............................................... 86  
6.7 The ApplicationUseCasesSetup class .................................... 86  
6.8 Processing a request by various WebSI components ................... 88  
6.9 Storing data required by activity classes in a UseCase ................ 90  
6.10 Associating a Thread with a UseCase .................................... 91  
6.11 Accessing the database via DBProxy beans ............................. 104  
6.12 The DB activity class ......................................................... 107  
6.13 The Adapter design pattern ................................................ 109  
6.14 The Adapter design pattern with an abstract adapter class .......... 109  
6.15 Adapting a ResultSet to a Table ......................................... 112  
6.16 The members of the UseCase class, related to the inclusion mechanism .. 113  
7.1 Traditional software development approach vs. WebSI approach ...... 127  
7.2 Common user interface controls and their functionalities ............. 131  
7.3 The Model-View-Controller (MVC) design pattern ..................... 132  
7.4 Plasticity domain and plasticity threshold of a user interface. Picture taken from [7] ............................................................ 142  
7.5 Designing a user interface for a single context. Picture taken from [6] .. 143  
7.6 Designing a plastic user interface for two different contexts. Picture taken from [7] ............................................................ 144  
7.7 An ideal design process for a plastic user interface. Picture taken from [7] 145  
A.1 Starting the J2EE server from NetBeans IDE ........................... 171  
A.2 Adding Cloudscape JARs to the J2EE server CLASSPATH ............ 172  
A.3 Creating a connection pool—step 1 ...................................... 172
| A.4 | Creating a connection pool—step 2 | 173 |
| A.5 | Creating a connection pool—step 3 | 173 |
| A.6 | Creating a JDBC resource | 173 |
| A.7 | Deploying the DBProxy component—step 1 | 174 |
| A.8 | Deploying the DBProxy component—step 2 | 174 |
| A.9 | The contents of the WebSI Web component | 175 |
| A.10 | “Mounting” a JAR file in NetBeans IDE | 176 |
| A.11 | The WebSI war directory structure | 177 |
| A.12 | Setting the default J2EE server | 179 |

| B.1 | The twocolumns_disp display, before “Login” is performed (top) and after the “Login” (bottom) | 184 |
| B.2 | The twocolumns_elegant_disp display, before “Login” is performed (top) and after the “Login” (bottom) | 185 |
| B.3 | The twocolumns_sky_disp display, before “Login” is performed (top) and after the “Login” (bottom) | 186 |
| B.4 | The topscreen_uc_links_disp display, before “Login” is performed (top) and after the “Login” (bottom) | 187 |
| B.5 | The listbox_disp display, before “Login” is performed (top) and after the “Login” (bottom) | 188 |
| B.6 | The dynamic_menu_disp display, before “Login” is performed (top) and after the “Login” (bottom) | 188 |
| B.7 | The imagebuttons_disp display, before “Login” is performed (top) and after the “Login” (bottom) | 189 |
| B.8 | The output_default style, implementing the Output.scalar and Output.table methods | 190 |
| B.9 | The output_default2 style, implementing the Output.scalar and Output.table methods | 191 |
| B.10 | The output_elegant style, implementing the Output.scalar and Output.table methods | 192 |
| B.11 | The output_elegant-wide_tables style, implementing the Output.scalar and Output.table methods | 193 |
| B.12 | The output_sky style, implementing the Output.scalar and Output.table methods | 194 |
| B.13 | Implementing the Action.action method by various stylesheets. From top to bottom: action_default, action_default2, action_elegant, action_sky | 195 |
| B.14 | The user_composed_default style implementing the UserComposed.any[Optional], UserComposed.anyLong[Optional] and UserComposed.secret methods (from top to bottom) | 196 |
| B.15 | The user_composed_default style implementing the UserComposed.yesNo and UserComposed.yesNoOptional methods (from left to right) | 196 |
| B.16 | The user_composed_default style implementing the UserComposed.time and UserComposed.timeOptional methods (from top to bottom) | 197 |
| B.17 | The user_composed_default style implementing the UserComposed.date and UserComposed.dateOptional methods (from left to right) | 198 |
B.18 The `user_composed.default2` style implementing the following methods: `UserComposed.any`, `UserComposed.anyLong`, `UserComposed.secret`, `UserComposed.yesNo`, `UserComposed.yesNoOptional`, `UserComposed.time` and `UserComposed.timeOptional`.

B.19 The `user_composed.default2` style, implementing the `UserComposed.date` and `UserComposed.dateOptional` methods.

B.20 The `user_composed.elegant` style.

B.21 The `user_composed.sky` style.

B.22 The `select_default` style, implementing `Select.one[Optional]` methods, for single-column and multiple-column tables.

B.23 The `select_default` style, implementing `Select.oneOrMore[Optional]` methods, for single-column and multiple-column tables.

B.24 The `select_default` style, implementing `Select.action` method, for single-column and multiple-column tables.

B.25 The `select_default2` style, implementing `Select.one[Optional]` methods, for single-column and multiple-column tables.

B.26 The `select_default2` style, implementing `Select.oneOrMore[Optional]` methods, for single-column and multiple-column tables.

B.27 The `select_default2` style, implementing `Select.action` method, for single-column and multiple-column tables.

B.28 The `select_default3` style, implementing `Select.one` and `Select.oneOrMore` methods, for multiple-column tables.

B.29 The `select_elegant` style, implementing the following methods for multiple-column tables: `Select.oneOptional`, `Select.oneOrMoreOptional` and `Select.action`.

B.30 The `select_elegant2` style, implementing the following methods for single-column tables: `Select.oneOptional`, `Select.oneOrMoreOptional` and `Select.action`.

B.31 The `select_elegant3` style, implementing the following methods for single-column tables: `Select.one`, `Select.oneOptional`, `Select.oneOrMore` and `Select.oneOrMoreOptional`.

B.32 The `select_elegant.wide_tables` style, implementing `Select.oneOptional` method, for multiple-column tables.

B.33 The `select_elegant.wide_tables` style, implementing `Select.oneOrMoreOptional` method, for multiple-column tables.

B.34 The `select_elegant.wide_tables` style, implementing `Select.action` method, for multiple-column tables.

B.35 The `select.sky` style, implementing the methods of `Select.activity` class.
List of Tables

3.1 Input and output activities classification ........................................... 23
3.2 The WebSI output methods .......................................................... 23
3.3 The WebSI input methods ............................................................. 24
3.4 WebSI database manipulation methods ........................................... 26
3.5 Comparison of use-case relationships ............................................. 28
4.1 Recommended SQL datatypes ....................................................... 42
5.1 The schema of the category table .................................................. 53
5.2 The schema of the product table .................................................... 53
5.3 The schema of the supplier table ................................................... 54
5.4 The schema of the offer table ....................................................... 55
5.5 The schema of the customer table .................................................. 55
5.6 The schema of the interested table ................................................ 56
5.7 The schema of the operator table ................................................... 56
5.8 Verifying the implementation of the “Add Product” use case against its
    specification ........................................................................... 62
5.9 Missing specifications and the Java statements related to them in the “Add
    Product” use case .................................................................. 63
5.10 Verifying the implementation of the “Select Offers” use case against its
    specification ........................................................................... 69
5.11 Verifying the implementation of the “Select Offers” use case against its
    specification ........................................................................... 70
5.12 Missing specifications and the Java statements related to them in the “Se-
    lect Offers” use case ............................................................... 71
6.1 WebSI XML namespaces and namespace prefixes ............................. 80
6.2 String conversion methods used by the Fetch activity class ............. 103
6.3 Dividing the inclusion-related data members into four categories ........ 113
7.1 Basic list box and combo box functionalities .................................... 130
7.2 User interface construction approach comparisons summary .......... 145
7.3 The timetables of student laboratories ............................................ 147
7.4 Hours invested into projects, as reported by the students (Winter 2003–
    2004 semester) ..................................................................... 153
7.5 Hours invested into projects, as reported by the students (Spring 2004
    semester) ............................................................................ 154
B.1 Select activity class methods implementations by select_default style 203
B.2 Select activity class methods implementations by select_default2 style 206
B.3 Select activity class methods implementations by select_default3 style 210
B.4 Select activity class methods implementations by select_elegant3style 213
B.5 Select activity class methods implementations by select_elegant_wide_tables style . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 215
Abstract

In order to reduce the program development costs, frameworks of ready-made software modules are often employed. In the traditional object-oriented (OO) approach, these modules are designed as abstractions of the objects of the application domain. This study investigates the feasibility of the specification oriented (SO) approach for framework design. Following this approach, introduced by Kantorowitzz and Tadmor, the modules of the framework are designed to enable a direct coding of a use-case specification. It is expected that the produced code will be “close” to the specification, which should facilitate verifying that the code implements the specification precisely. The goal of the SO approach is thus to facilitate code verification. Kantorowitz and Tadmor compared code of small, stand-alone information systems, produced using a traditional OO framework to code produced using a framework developed by the SO approach. That early study suggested that the SO approach facilitates code verification, as was expected. In addition, the SO approach required significantly less code than the traditional OO approach. The goal of the current study was to attempt to achieve a higher level of similarity between the specifications and the code. For this purpose we developed an experimental framework called WebSI (Web Simple Interfacing). WebSI was designed to enable the manufacturing of Web-based, interactive, multi-user information systems. WebSI enables a direct manual translation of English language use-case specifications of such systems into Java code. In order to make the code similar to use-case specifications, we designed WebSI to implement some of the abstractions employed in the use-case specifications. It was assumed that the essence of these abstractions is that an information system is modelled solely by the flow and processing of data. The components of WebSI, therefore, attempt to hide code that is not directly related to the data flow and processing. WebSI hides the code for the construction of the user interface (UI), database access code and Web-related technicalities. The result of these high-level abstractions is that WebSI services are declarative rather than operative. The WebSI programmer specifies what is the desired functionality, e.g., selecting a single item out of a set of items, rather than how this functionality should be implemented. The implementation of a declarative WebSI service is accomplished by a WebSI component, called the interaction style (IS). A particular IS may, for example, implement the item selection by a combo box, while another IS may implement it by a set of radio buttons. Exchanging the employed IS with an alternative one is fast and requires no recompilation of the use-case implementation code. The programmer may experiment with different IS’s and select the one that produces the most appropriate UI. WebSI IS’s are not application-dependent and, therefore, may be reused in different applications. Developing a new IS from scratch, may, however, be quite labor-intensive. WebSI comes with a repository of ready-made IS’s. Employing such a ready-made IS saves the UI development effort and shortens the time required to deliver an application. Students used WebSI for the development of eleven Web-based information systems. In one case, the students estimated that WebSI saved about 30% of the development effort, compared to the traditional OO approach. One of the systems
was constructed to support the needs of a real Web community; this system was tested by real users. An unexpected result was that the UI’s produced by WebSI in these student projects seemed to have a satisfactory usability. This promising observation calls for a further research, in order to map the applications, in which our technology can produce acceptable UI’s. Students that employed WebSI observed that the structure of the produced code was easy to understand and that it was easy to trace the code related to each use case. We believe, therefore, that their code was relatively easy to verify. We believe also that the high-level WebSI abstractions enabled the students to better focus on the essence of their applications, rather than on technicalities, and helped producing quality systems.
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>AL</td>
<td>Application logic</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical user interface</td>
</tr>
<tr>
<td>HCI</td>
<td>Human-computer interaction</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>IO</td>
<td>Input-Output</td>
</tr>
<tr>
<td>IS</td>
<td>Interaction style</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>OO</td>
<td>Object-oriented</td>
</tr>
<tr>
<td>SO</td>
<td>Specification-oriented</td>
</tr>
<tr>
<td>SOP</td>
<td>Specification-oriented programming</td>
</tr>
<tr>
<td>UI</td>
<td>User interface</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The primary challenge of software engineering is to produce a quality software at low costs. By quality software we mean software products that possess some of the following properties:

- The product is usable. Usability is both the degree by which the system meets the users’ needs, and the degree, to which the system enables the users to accomplish their tasks with a minimal human effort and in a pleasant way [51, 50].

- It has been verified that the code of the software implements its specification precisely.

- Extending or modifying the product can be performed with a reasonable effort. For example, the effort of extending a system with new functionality may be independent of the number of already existing functionalities. Another way to formulate this property is by saying that the extension complexity [33] of the software is $O(1)$.

Achieving the properties above involves considerable costs, when it is required to manufacture and manage a large and complex software system. Some of the commonly applied techniques to reduce these costs include:

- Employing high-level, declarative languages
- Employing software development processes
- Employing frameworks

In further sections we will discuss each one of the listed techniques.

1.1 High-Level Declarative Languages

The philosophy of many traditional programming languages, such as C and Pascal, is that any somewhat complex operation should be explicitly coded by the programmer. In somewhat higher-level object-oriented (OO) programming languages, such as C++ and Java, some of the low-level operations are hidden from the programmer. An example is an invocation of a C++ virtual member function. Such function is said to have dynamic binding rather than static binding, i.e., the decision which function to invoke is performed at run-time rather than at compile time. The code that at run-time locates the function to be invoked based on the actual (dynamic) type of an object is hidden from the programmer. However, even in such quite high-level languages, the programmer in many cases codes
how things should be done, rather than what should be done, i.e., the languages are more operative than declarative.

An example of a high-level declarative language is the Structured Query Language (SQL). This language is designed to manipulate relational data, i.e., data that are logically organized in tables (relations). SQL queries specify what is the desired outcome of an operation, but not how this operation should be performed. The physical organization of data, and the implementations of operations on it, are hidden from the programmer.

Consider, for example, the following SQL query:

```sql
SELECT EMPL_NR, EMPL_NAME FROM EMPLOYEE WHERE SALARY < 10000
```

In this query the programmer requests to see the names and the employee numbers of all the employees, whose salary is less than 10,000. The programmer here operates on a logical table, which contains the details of all employees. The outcome of the query is also a logical table. The programmer is required to know neither how the employee data are stored physically, nor which algorithm is used to fetch the requested data.

Declarative notations, such as SQL, allow the programmer to focus on the system’s requirements, rather than on low-level implementation details. The required low-level work is performed by the underlying software, and this work is hidden from the programmer. This saves coding and verification of potentially complex data structures and manipulation routines.

### 1.2 The Software Development Process

A software development process is a set of recommended activities, guidelines and milestones leading the developers to the desired software product. Some of the most established processes are the Unified Software Development Process (USDP) [22] and the Rational Unified Process (RUP) [40]. Typically, a software process includes activities such as:

- Requirements elicitation
- Specification development and validation
- Specification analysis
- Software design
- Unit coding and integration
- Verification, including testing

The way, in which these basic activities are performed, differs among the various software processes. The traditional waterfall process [57], for example, recommends the activities to be performed sequentially. The USDP, on the other hand, organizes the development into four phases— inception, elaboration, construction and transition. Each phase in USDP is furthermore divided into several iterations. An iteration potentially includes all of the basic process activities listed above. During early iterations, more time is typically devoted to the requirements and specification activities. Later iterations may concentrate on software design, coding and verification.

In general, the activities suggested by the various software processes can be divided into two major parts. During the first, requirements and specification part, the developers
devote most of their effort to understanding the users’ needs and developing and validating the system specification. During the second, construction part, the developers concentrate on coding the software and verifying that it corresponds to the specifications.

The USDP and some other software processes employ use cases [21] to specify the developed system. In the following we consider only the processes that employ use-case specifications.

A use case is a single application of a system. A system may be specified by a detailed specification of all its use cases [21]. Use cases are often described in a natural language, e.g., English, and may be accompanied by drawings of the user interfaces (UI). Consider, for example, a use case in a sales management system, which specifies the addition of a new sales offer by a supplier:

1. The system presents the set of products, for which the supplier has no given offers. For each product, its name and description are presented.
2. The supplier selects a single product to give an offer for.
3. The supplier specifies the price.
4. The supplier approves.
5. The system registers the new offer and presents a success message.

The above use-case specification does not tell how the system locates the products, for which the supplier has no given offers. The specification also provides no information on where and how the details of the new offer are stored. The above specification is essentially only a specification of user’s input data and the corresponding system’s output data. The specification does not describe how the system computes its outputs from the user’s inputs.

The use of natural language and UI descriptions enable application domain experts and human-computer interaction (HCI) experts, which may not be familiar with formal specification notations, to validate the use cases. The domain experts validate the completeness and correctness of system functionalities, while the HCI experts validate the system’s friendliness. The purpose of this validation by both domain and HCI experts is thus to ensure the usability of the system. The product of the specification development activities is, therefore, a use-case specification of the system, that is validated for its usability.

Requirements elicitation and specification development are inherently difficult. The potential users of the system may not be able to express their needs or not be aware of what a computer system may do. There exists evidence that many software failures originate from incorrect requirements or an incomplete specification. The promising Play-In/Play-Out approach [19] is intended to automate the specification development and validation activity. Following this methodology, first a graphical user interface (GUI) of the system is constructed. Then the developer operates the GUI as an end-user, and specifies the desired reactions of the system—also with the help of the GUI. In this way, the developer specifies the system by “playing-in” various scenarios. At the “play-out” stage, the developed specification is directly executed, again by operating the system’s GUI. During this stage, the specified system may also be tested with scenarios, which are different from the “played-in” ones. In some cases, the “play-out” may actually serve as the final implementation.
The second, construction part of a software process involves an analysis of the validated specifications. This includes developing methods for computing the system outputs from the user’s inputs. Based on this analysis, the developers design and code the software. The construction part involves also a verification, that the produced code implements the use-case specifications precisely. Since the use-case specifications were validated for their usability, the verified code is expected to retain this usability.

A common way of performing verification is by testing. When employing a use-case specification of the system, it is a good idea to derive the various test scenarios from the system’s use cases. Each test scenario may test a particular use case, or it may span several use cases, testing a complex workflow. For each scenario a set of test data and expected results are prepared.

An important feature of a good software process is modelling, i.e., developing a set of artifacts that provide different views on software, facilitating its analysis and design. USDP was developed in connection with the popular standardized Unified Modelling Language (UML) [58]. This graphical language provides notations for describing such aspects of the system as static structure, dynamic behavior and physical arrangement of system components on various hardware nodes. Employing models requires, however, an additional effort during the various verification and validation activities, because all the models must be assessed for verity and validity [9].

The bottom line is that the second, construction part of a software process such as USDP is labor-intensive.

### 1.3 Frameworks

A possible way of alleviating the difficulties of the construction part of a software process is by employing a framework. We may define a framework as:

> A set of reusable software components, and a methodology of combining these components together in order to produce custom software in a specific domain.

The developer uses a framework in two main ways:

1. By extending the framework’s components and overriding or customizing their default behavior. This approach is employed with so called white-box frameworks [56]. For example, the developer defines a class that inherits from one or more classes of the framework. Then the developer overrides the implementations of some of the inherited classes’ (virtual) methods by custom implementations. The developer may completely re-implement an inherited method or she may customize it by calling the parent-class version of the method at some point in the overriding method.

2. By composing the framework’s components in various ways. This approach is employed with so called black-box frameworks [56]. An example for this approach would be the case when the developer instantiates one or more of the framework’s classes and uses the resulting objects as data members of her class. Then the developer employs the existing interfaces of these framework’s classes to operate the instantiated objects. The behavior of the framework’s components is not customized in this case.
It is suggested in [56], that black-box frameworks are more mature ones. Most of the frameworks, however, are “gray-box” frameworks, i.e., their components can be both composed and extended.

The power of frameworks essentially stems from two factors:

**Code reuse** The code encapsulated in the components of the framework is reused in different applications. Assuming that the framework’s code has been verified, the programmer can benefit both from less coding and from relying on the correctness of the framework’s code.

**Design reuse** Frameworks are built to solve design problems, common to software in the target domain, i.e., in order to avoid redesigning and revalidating common solutions to recurring requirements and design challenges [14, 5]. As a result, the framework usually comes with a specific methodology of how the programmer should combine the framework’s components together.

Frameworks are usually designed to construct software with useful software quality properties, such as the ease of verification, modification and extension.

A special kind of frameworks are **application frameworks**, which provide most of the services necessary to build complete applications. Using an application framework, however, does not replace the software development process. Instead, the framework shortens the development process by complementing its general guidelines with a domain-specific, concrete advice [5]. A well-designed framework can reduce the cost of developing an application by an order of magnitude [56]. Since software systems are becoming more and more complex, some researchers believe, that frameworks will be at the core of software technology [13].

Let us now bring examples of several well-known frameworks.

**Swing [27]** The framework of Java programming language for constructing GUI’s. This is a typical OO framework. Its domain are stand-alone interactive Java applications with a GUI. Swing provides classes for such domain objects as push buttons, combo boxes, radio buttons, check boxes, text fields, panels, frames, dialogs and events. To construct typical GUI’s, it is sufficient to combine the ready Swing components. If, however, a non-standard GUI is required, Swing provides specific extension rules for its components. Therefore, the Swing framework may be considered a “gray-box” framework.

**MPICH [47]** A portable implementation of MPI (Message Passing Interface) [46], an industry standard for message-passing libraries. MPI defines and MPICH implements a collection of routines for facilitating communication (exchange of data and synchronization of tasks) among the processors in a distributed-memory parallel program. The MPI routines include point-to-point message passing, collective communication and group management.

**Java Servlets [30]** Are a part of J2EE [24] technology. Servlets provide a component-based, platform-independent method for building Web-based applications. The servlets API (application programming interface) is a typical example of a white-box framework. To construct a Web application, the programmer subclasses the `javax.servlet.http.HttpServlet` class. Then the programmer overrides its various `doXXX` methods, such as `doGet` and `doPost`, to handle the different kinds of HTTP requests.
**Microsoft Access** May be considered an application framework, which helps to construct small, stand-alone, data-driven information systems. A database schema can be designed and filled with data. Afterwards, it is possible to design a GUI and write database queries, which are performed, when the user operates the system.

1.4 Our Project

The purpose of our project is to develop and validate a tool, which supports the innovative *specification-oriented* (SO) software construction methodology, introduced by [61, 39]. We will elaborate on this methodology in the next chapter. We would like to try to overcome the disadvantages of the tool, developed in the previous work, as we will also describe in detail in the next chapter.

Our tool will be based on the combination of techniques described above. We will develop an application framework, targeted at construction of software products in a specific domain. The components of our application framework will be declarative rather than operative. In other words, the programmer will employ an API delivered by our framework, whose methods enable her to perform high-level, semantic-specified tasks, rather than specifying how a certain task should be accomplished. Our framework will be designed to fit into the construction part of the USDP process, assuming that the requirements and specification part has already been performed.

The domain, at which we decided to target our framework, are Web-based, interactive, multi-user information systems, specified with use cases.

1.5 Document Outline

The rest of this document is organized as follows. In Chap. 2 we present the main ideas of the SO programming methodology. We also discuss the original motivation behind it and the previous work related to it.

Chapter 3 introduces the principles of the programming model, implemented in our application framework, which supports the SO methodology. Here we discuss the main framework’s components and services. We also explain how the framework’s facilities should be combined to produce systems in our domain. The main design decisions of our framework are justified in “Rationale” paragraphs at appropriate places. This chapter discusses in an abstract way essentially everything the programmer needs to know.

Chapter 4 gives the complete API of our framework.

Chapter 5 demonstrates the construction of a small information system with the help of our application framework.

In Chap. 6 we describe the internals of our framework. We also give guidelines for its extension and modification.

Chapter 7 evaluates our work.

Chapter 8 discusses the current limitations of our tool and desired future improvements.

Chapter 9 suggests directions for further research.

Chapter 10 concludes the thesis.

Appendix A brings technical information on using our tool: installing and configuring the required software, and compiling and deploying the ready information systems.

Appendix B lists and demonstrates all the IS’s available in our framework’s repository.
Chapter 2

Specification-Oriented Programming

In this chapter we describe the main ideas behind the specification-oriented programming (SOP) methodology. Section 2.1 brings the original motivation for the SOP approach. Section 2.2 presents the theory, based on which our tool has been designed. The theory described in this section has been developed in this project. Section 2.3 focuses on an important concept of SOP—the interaction style (IS) [38, 61, 39] concept. It has been introduced in [61, 39] and further elaborated by the author. Section 2.4 discusses the work, which first introduced SOP and Sec. 2.5 mentions the pre-thesis project, performed by the author, which served as a base for the current project.

2.1 Original Motivation

One of the labor-intensive stages of the software development process is verifying that the code implements the specifications precisely. One of the reasons for verification being difficult stems from the use of a number of different notations in the program development process. Natural languages are often employed in the specifications. The validated specifications may be translated into a formal specification language. The structure of the program designed to meet the specification may be modelled by UML diagrams. Finally, we have the code written in a programming language. Each one of these different kinds of languages may potentially affect the correctness of the verification of the code. The validated natural-language specification may be ambiguous or incomplete. The translation of natural-language specifications to a formal language specification can be both labor-intensive and error-prone. Showing that the code of a program implements formal specifications correctly is difficult, as programming languages can be quite different from the formal specification language employed.

It may be expected that the difficulties mentioned above can be alleviated by employing code that is “similar” to the specifications. This expectation was the original motivation for the SOP approach, introduced in [61, 39]—to reduce the verification costs of the software development process.

One may consider two different approaches for producing code that is “similar” to the specifications. One approach is to develop a special specification-oriented (SO) programming language. The code written in such a language should be similar to a specification. The second approach is to develop a SO software framework for a general-purpose programming language such as Java.
2.2 Specification-Oriented Frameworks

A SO application framework is designed to produce the system’s implementation directly from its specifications. The possibility of deriving an implementation of a system directly from its specifications was demonstrated for reactive systems, in which a state chart [58] specification of such a system can be executed [18]. The developer of such a system has thus only to specify it by its state chart, and needs not invest further time in designing and implementing the system.

A state chart can be executed, because it explicitly specifies all the computations in the system. For data-centered information systems considered in our study, it is difficult and unpractical to produce such complete explicit specification of the entire system. Therefore, we decided to employ use-case specifications, which specify the system as a black box, from the user’s point of view, rather than trying to capture its explicit computations. Our framework then becomes use case-oriented.

A use case-oriented application framework is designed to enable a direct manual translation of natural-language use-case specifications into code. During this translation, the design of the framework should strive to keep the programmer as close as possible to the use-case specification. Two following criteria may be suggested for this:

- The code should have the same structure as the use-case specifications. For example, each one of the system’s use cases is coded in a separate software component of the framework. Such component may be called use-case component.

- The functions (or methods) of the use case component should correspond to the steps of the message flow in the use-case description. Each stage of the use-case description should be mapped into few, easy-to-locate code statements.

Use-case specifications specify the application logic (AL). We define application logic as a description of services, provided by the application to the user, without such technicalities as user interface (UI) details, database access details and handling multiple users. The AL is specified in use cases from the user’s point of view. This means, that the user sees the system as a black box, to which she supplies inputs and from which she receives outputs. Therefore, a typical use-case description defines the flow of data from the user to the system (input) and from the system to the user (output). If we are to directly translate a use-case description into code, we must strive to write code that only involves the activities described in the use case, which means that our code must be mostly around input and output (IO). Obviously, just the IO is not enough, since we also need to perform the processing of data, i.e., to perform the various computations required by the use case. Our goal, therefore, is to enable the programmer to write code that deals only with AL of a specific system, which includes:

- Input requests
- Fetching the input values supplied by the user
- Data manipulations and computations
- Output presentations

All the other activities, typical for information systems in our domain, must be hidden from the programmer and performed automatically by the framework. These other activities include:
• Construction and manipulation of the UI

• Database access technical details, such as connecting, disconnecting, managing database resources and transactions

• Managing the multiple system users

• Handling the various security-related details, such as identifying the user and granting access to system resources according to the user’s credentials

• Handling the various environment-specific details (such as HTTP communication details)

We understand now, that the programmer’s code should include only the four AL-related activities mentioned above. The question is how to derive such code directly from the use-case specification. Let us take a look again at the use case presented in Sec. 1.2:

1. The system presents the set of products, for which the supplier has no given offers. For each product, its name and description are presented.

2. The supplier selects a single product to give an offer for.

3. The supplier specifies the price.

4. The supplier approves.

5. The system registers the new offer and presents a success message.

Now imagine that this description must be translated directly into code. Which services should the framework provide to enable this?

Let us take a look at steps (1) and (2). Here the user is required to select a product out of a set of products. It seems that the programmer should be able to directly code something like this:

(a) Compute the set of products, for which the current supplier has no given offers
(b) Request the user to select a single product out of this set

Let us continue to steps (3) and (4):

(c) Request the user to supply a user-composed value (the price)
(d) Enable the user to approve

Now the system waits for the user to supply the required inputs and to approve. After the user approves, the following should be executed:

(e) Fetch the details of the product selected by the user
(f) Fetch the price specified by the user
(g) Store the details of the new offer in the system’s database
(h) Present a success message

Steps (a) and (g) obviously involve computations. Here the programmer must be familiar with the database schema, and her code for these steps cannot be derived directly from the use case, since the use case does not mention implementation details. All the other steps, however, involve only IO.

The use-case description we gave is, however, not complete. For example, what if the current supplier has already given offers for all the available products? This is an exceptional situation, and for that purpose UML use-case descriptions can contain “exceptions”, which describe how the system should react in such exceptional situations. Here is a description of our exception:
**Exception** There are no products, for which the supplier has not given offers. The system presents an error message.

How this exception should be coded with the help of our framework? A step (a1) can be added to the code, like this:

(a) Compute the set of products, for which the current supplier has not given offers
(a1) If this set is empty, present an error message, otherwise proceed to step (b)
(b) Request the user to select a single product out of this set

We see that the pseudo-code we have written comes quite close to the original use-case specification. We now can conclude that the framework services that perform the various activities, should be high-level and declarative, rather than operative. Furthermore, the framework services should enable the programmer to directly employ the abstractions, which are frequently seen in use-case specifications of systems in our domain. For example, there should be a method, which enables the user to select a single item out of several items. Indeed, it is not unfrequent to see a statement in a use-case description, which reads: “The user selects a …”. But this method should not specify, which UI technique should be used to perform this operation. Possible techniques for the graphical user interface (GUI) include a set of radio buttons, a combo box and a single-selection list box. A possible technique for a command-line UI is listing the available items, assigning a number to each item, and waiting for the user to type the number of the desired item.

The small example above gave us some basic understanding of how the framework’s services should look like. Now we can formulate the goals that should drive the design of our use case-oriented application framework:

- We must strive to hide all the kinds of code, which are common to systems in our domain.
- The programmer will have to write only the code that is specific to the information system being currently constructed.
- The system-specific code will involve only: input requests, fetching the user input values, data manipulations and computations, and output presentations.
- The system-specific code should be as much declarative as possible. For example, an input request should specify which data the user is required to supply. The request will not specify which specific UI controls are used to enable the user to supply the data.

As an additional goal, the framework should be designed to make the programmer’s code resemble the natural language use-case specifications.

Let us now mention some immediate benefits of a proposed framework. First of all, because of the declarative nature of the framework’s services, the programmer might actually be able to narrow the gap between the use-case specification and the code, i.e., she will be able to derive most of the code directly from the use cases. By this, the programmer skips or considerably alleviates the difficulties of the specification analysis and software design activities. In addition, code resembling the specifications should contribute to:

- Direct system verification by code inspection
• Tracing the specifications backwards and forwards
• Retaining the usability of the use-case specification

A further benefit is that by employing the framework’s declarative services, the implementation of the use cases has the potential of being environment-independent. The next section brings more on this.

2.3 The Interaction Style Component

When employing a use case-oriented framework, the programmer codes the AL using the framework’s application programming interface (API). As mentioned before, this API delivers high-level services, which are declarative, rather than operative, i.e., they specify what has to be done, rather than how this should be done.

The implementations of the declarative services are hidden in a ready-made framework component, called the interaction style (IS) [38, 61, 39]. This component is environment-specific. By “environment” we mean the whole software and hardware environment, in which the AL has to operate, such as:

• The underlying hardware and middleware, e.g., a stand-alone PC with a certain operating system, or the Web and a Web application server.

• The UI platform, e.g., a desktop GUI, a GUI of a standard Web browser, a UI of a certain mobile device or a command-line UI.

• The employed database engine.

Therefore, the IS component implements the declarative operations of the framework’s API for a certain environment. The described framework architecture is illustrated on Fig. 2.1.

![Figure 2.1: The architecture of a use case-oriented framework](image)

A use case-oriented application framework should provide a repository of the ready-made IS components, developed for different environments. In addition, several IS’s may
be designed for the same environment; they may differ, for example, in their UI properties, such as UI controls, colors, graphics and fonts. Ideally, it should be possible to combine the AL with any IS from the framework’s repository at deployment time. This means that the AL code should neither involve any environment-specific details, nor specify which IS should be employed. At deployment time, the deployer specifies which IS should be combined with a certain AL. Exchanging the IS should ideally require no changes or recompiations of the AL code.

The suggested software architecture has several potential benefits:

**A clear separation of development responsibilities** The AL is developed by the programmers, which have sufficient knowledge in the application’s domain. The IS is developed by experts familiar with a certain environment, and by UI and usability experts.

**The IS is not application-specific** Therefore, its development and maintenance can be performed independently of a certain application. The same IS may be reused in many applications.

**The AL is environment-independent** The same AL can operate in all the environments, for which an environment-specific IS component has been developed. For example, the same AL code may be operated through a standard Web browser, through a PDA (Personal Digital Assistant) and through a mobile Web-enabled phone. The AL code may be considered as the code of an application. Any maintenance activities, such as new releases and bug corrections, are done solely in the AL code. Afterwards, the same new AL can be deployed on all the required environments.

For the sake of simplicity, when employing the term *interaction style* throughout this document, we do not consider the whole software and hardware environment. Instead, we use this term in its original meaning from [38], which considers only the UI platform and appearance. Therefore, when we say “exchanging the employed IS with another one”, we mean exchanging the UI properties of a system, such as the visual appearance or the UI platform, e.g., GUI vs. command-line UI. We do not mean by this, e.g., exchanging the employed database engine.

### 2.4 Previous Work—\(SI\)

The feasibility of a use case-oriented framework has been first explored in [61, 39]. In this work, a use case-oriented framework named \(SI\) (*Simple Interfacing*) has been developed. This framework is targeted at small, stand-alone information systems.

In one experiment, an English statement in the use-case specification was translated into 2–3 Java statements, which resembled the specification to a degree that made it relatively easy to verify, that the code corresponds to the specification. One system was implemented twice. Once using the Java’s GUI Swing [27] framework and the Java’s database connectivity package JDBC [26]. The second implementation employed the components of \(SI\). There were 2.4 as many lines of code in the Swing- and JDBC-based implementation, than in the \(SI\)-based code. This suggests, that using components that model use cases, as done in \(SI\), saves coding, when compared to using components that model the objects of the domain in a classical object-oriented (OO) design (such as Swing and JDBC). The suggestion supports the observation made by [41], that a classical OO component design does not necessarily produce the most appropriate components.
The remarkable code savings obtained with $SI$ were achieved mostly by hiding much of
the GUI and database access code in the components of $SI$.

The $SI$ framework has shown the feasibility of the use case-oriented approach. However,
being a very first raw prototype, it had several drawbacks:

- The API for communication with the user (which is a part of the interface between
  the AL and the IS) was not completely declarative. It contained methods, such as
  `addButton` and `addLabel`, mentioning specific GUI controls, which made the
  framework environment-dependent. More important, the methods of this API were
  not directly driven by the typical abstractions of use-case specifications. The meth-
  ods were mostly selected on the add-hoc basis.

- In some cases, the programmer’s code depended on the employed IS. Therefore,
  switching to an alternative IS might have required code changes.

- There was no clear separation in the various IO activities between the user and the
  system. For example, the `display` method (which is a very successful declarative
  method) presented the results of an SQL query to the user, but it could also have
  been used for input (depending on the employed IS)—the user could select items in
  the presented results.

- Fetching the user’s inputs was messy and error-prone. There was no clean model
  for specifying which input value to fetch, and the programmer relied on the order
  of input requests and on some other $SI$ internals to correctly fetch the input values.

- The programmer had to manually implement the initiation of a use case. A possible
  way of doing this was by defining artificial “Menu” use cases, which presented the
  set of choices to the user, each choice leading to a particular use case.

  We believe that the initiation of a use case is quite an important point. The UML
  use-case specification divides the application into a set of atomic services, each
  one specified by a use case. The UML use-case specification does not define how
  an execution of a certain use case is initiated. Therefore, the programmer’s code
  should contain no details concerning use-case initiations.  

- The IS component to be employed was selected in the programmer’s code, rather
  than being specified at deployment time. Therefore, to employ an alternative IS,
  the code had to be changed and recompiled. Furthermore, the selection of the IS
  for different situations appeared in several places in the code. As a result the actual
  IS was defined by a set of the made selections, rather than in one place, outside the
  programmer’s code.

- The programmer’s code mentioned such objects as panes and frames, which are
  quite low-level.

- Systems produced with $SI$ were single-user applications. The database was not
  accessed through database transactions. To be more exact, each single SQL query
  was executed in a separate transaction.

- The $SI$ framework had serious limitations in the kind of GUI’s it could produce.

\footnote{Except when a use case explicitly references another use case, such as in the \texttt{<<include>>} relationship.}
It must, of course, be mentioned, that the drawbacks of $SI$ originate from the very basic understanding of use case-oriented frameworks feasibility problems in that pioneer project.

2.5 $SI+$

As a pre-thesis project, the author, together with another student, developed a new version of $SI$, called $SI+$ [36, 37]. The principle goal of $SI+$ was to enable the construction of more sophisticated GUI's, than with $SI$. We consider the most important contribution of $SI+$ to be the concept of a use-case displayer. This is a ready-made framework component, which attacks the problem of use-case initiation. Recall, that a use-case specification defines a system as a set of atomic services, each one specified by a use case. The specification does not describe how a certain use case is initiated. Therefore, in $SI+$, the implementation of the specification is by a set of components, each one implementing a certain use case. These components are then submitted to the use-case displayer component. The use-case displayer “glues” all the use-case components together into one complete application; it visualizes all the use cases to the user and enables the user to initiate and execute them. Therefore, the use-case displayer component closes the gap between a set of developed use-case components and the final system.

Further contributions of $SI+$ are improved facilities for database access and presentation of tables, obtained by SQL queries.

As for the drawbacks, the $SI+$ project had several wrong design decisions:

- The programmer’s code mentioned specific GUI controls, even more than $SI$-based code. Furthermore, in many cases the programmers employed Swing to manipulate specific GUI controls. All this made the code non-declarative and environment-dependent. This also hindered the possibility of direct code verification by comparing the code to the natural language use-case specification.

- The programmer had to manually manage the consistency between the GUI’s of the various use cases. This was especially complex and error-prone. Many problems in $SI+$-based systems originated from this drawback.

- The selection of the IS was still performed in the code, in several places.

- $SI+$ employed a sub-use case concept, which is not a standard UML term.

The $SI+$ framework has been tested in the lab, in which the students employed it to implement several stand-alone information systems. All in all, the $SI+$ project helped us significantly to understand the challenges of our work. Many design decisions, which proved to be correct and successful in our current application framework, originated from our evaluation of $SI+$.
Chapter 3

Specification-Oriented Programming Model

In this chapter we introduce the principles of our suggested programming model. We also describe the basics of the tool we have developed, which partly supports the programming model. This tool is a use case-oriented application framework, which is targeted at construction of interactive, multi-user, computation and data-centered information systems, operating over the Web. Our framework is called WebSI (Web Simple Interfacing) [34, 35]. The code for our framework is written in Java programming language.

3.1 System Specification

It is assumed, that the software development process is initiated by a thorough requirements elicitation, which is followed by system specifications development. It is expected that the requirements and specifications have been validated, to ensure that the developers understand the users’ needs precisely. In addition, we assume that the specifications are evaluated for their usability, which means that the specified system is designed to enable the users to accomplish their work in a pleasant and efficient way.

In this project we employ use-case specifications [21]. A use case is a single application of a system, described in a natural language from the user’s point of view, and possibly accompanied by some description of the UI.

Rationale The reason, that we decided to work with use-case specifications, is that use cases are a well-known and popular specification methodology for interactive software. It is stated in [21], that an interactive software can be fully specified by all its use cases. The use-case concept together with additional important concepts from other software processes, were later elaborated into the Unified Software Development Process (USDP) [22], which was developed in connection with the popular standardized Unified Modelling Language (UML) [58].

The natural language in use-case specification enables its validation by domain, usability and UI experts, who may not be familiar with formal specification notations. Therefore, development processes, that employ use-case specifications, such as USDP, are expected to produce systems with high usability.
3.1.1 Data Flow-Centered Use-Case Specifications

For our programming model we suggest to employ use-case specifications that concentrate on the data that flows from the system to the user (output) and from the user to the system (input). We expect the use-case specifications not to mention any implementation details, UI details and other platform- and hardware-specific technicalities, such as database access details.

A typical description of a use case in an information system of our interest looks as follows:

1. The system presents a certain information to the user, and requests inputs from the user.
2. The user supplies the requested inputs.
3. The system computes the output and presents it to the user.
4. Possibly, the system requests additional inputs.
5. The user supplies the requested additional inputs.
6. The system computes the output and presents it to the user.
7. Possibly, the system requests additional inputs.
8. . . .

A use-case specification need only specify what output should be provided for certain inputs. The use-case specification must not define how the output should be computed from the inputs. We generalize it by saying that use-case specifications should define the semantics of the application, i.e., what services the system gives to the users, but not the syntax, i.e., how the services are provided, which specific UI will be employed and other specific implementation details.

**Rationale** By limiting ourselves to pure input and output of data in specifications, we benefit from the following factors:

- The specifier focuses on defining a data flow between the user and the system. This directly contributes to understanding and specifying the application logic (AL), which it is required to implement. The specifier is not distracted by UI issues and other details, not directly related to the AL.

- The resulting specification is independent of the UI platform. The UI, that will be generated automatically, may be a desktop-based GUI, as well as an audio-based UI, command-line UI or a mobile device UI.

- The specification is not limited to a certain hardware, database engine or a network communication protocol.
3.2 Relational Database

The important aspect of our programming model is a relational database, which is manipulated by a Structured Query Language (SQL). The programmer sees the database as \textit{logically single}, i.e., the database may be physically distributed, but appears to the programmer as if it were a single local database.

When constructing an information system, we assume that the database schema has been already designed and validated, and will not undergo any fundamental changes. This assumption is supported by the observation of [21], that the schema of the database changes much slower than its applications. The programmer, therefore, needs only be familiar with the schema, but she will not alter the schema.

**Rationale** The theories of relational databases design and querying are well-found. The relational database management systems (RDBMS) technology is a mature technology, whose usefulness has been proven over the years. In addition, in most cases there is no pressing need to develop a database schema from scratch. There exist ready-designed and validated schemas for many industry types, for example, those suggested by [59]. Finally, current database products provide useful services beyond querying, such as concurrent transactions support and database constraints.

3.3 Use-Case Classes and Interaction Methods

Each use case of the specification is implemented in a separate class, derived from the \texttt{websi.UseCase} class. We call these classes \textit{use-case classes}. At runtime, instances of these classes—the \textit{use-case objects}—are created automatically, on-demand by \textit{WebSI}.

Each use-case class defines one or more \textit{interaction methods}, that contain the translation of the natural language use-case specification into Java, employing \textit{WebSI} services. Interaction methods of a use-case class together implement all the specified interactions between the user and the system for a particular use case.

A single interaction method implements one \textit{input-output sequence} (IO sequence). We define an IO sequence to be a sequence of the following actions:

1. The system fetches the input values supplied by the user, as requested by the previous IO sequence.\footnote{In case there was no previous sequence, i.e., this IO sequence is the first one, this step does not apply.}

2. The system computes the output.

3. The system updates the database if needed.

4. The system presents the output to the user.

5. Optionally, the system requests additional inputs from the user. In that case, an additional IO sequence will occur.

\textit{WebSI} automatically invokes the required interaction method during an IO sequence, as the following describes:

1. The user supplies the inputs, requested by the previous IO sequence.
2. The required interaction method is automatically invoked by WebSI on a required use-case object.

3. The interaction method fetches the input values supplied by the user.

4. The interaction method computes the output.

5. The interaction method updates the database if needed.

6. Optionally, the interaction method requests additional inputs.

7. The interaction method ends.

8. WebSI sends the output and the input requests (if any) to the user.

9. If additional input requests have been issued, the user supplies the required input values. By this the user starts an additional IO sequence.

An interaction method must have the following special signature:

```
public void methodName() throws Exception;
```

A use-case class must implement at least one interaction method, called start. This method is invoked by WebSI, when the execution of the use case begins. In other words, the start method serves as a starting point for IO sequences, which will happen during the use-case execution.

A typical interaction method does the following:

1. Fetches the input values supplied by the user.

2. Computes the output, based on the supplied inputs, on the database and, if needed, with the help of external software components.

3. Updates the database if needed.

4. Presents the output to the user and requests additional inputs if necessary.

**Rationale**  
It may be noted, that a typical use-case description, presented in Sec. 3.1.1, can be decomposed into several IO sequences. Each such IO sequence will be implemented by an interaction method. Therefore, the structure of a use-case class resembles the natural language use-case specification of the AL.

The “`throws Exception`” clause in the signature of an interaction method enables the programmer not to handle any system exceptions in the code. These exceptions are passed out of the interaction method and eventually handled by WebSI. Some of the WebSI system exceptions may be caught and handled by the programmer (otherwise a default handling action is taken by WebSI). At present, these are the database constraints violation exceptions, as we will describe in Sec. 3.6. The programmer may, of course, be required to handle various exceptions from third-party software components.

The programmer must catch only specific exceptions in her code. Writing:

```
catch (Exception exc) { ... }
```

will catch any exception and prevent WebSI from correctly handling system errors.
3.4 Activity Classes

An interaction method, during its execution, employs WebSI services for performing activities, which include:

- Presenting outputs to the user and requesting inputs from the user.
- Fetching the input values supplied by the user.
- Manipulating the database.
- Obtaining the user’s identification details.

To perform these activities, WebSI provides several activity classes, each class supporting a certain activity. An activity class is an abstract class, which implements a set of static methods, that support a certain activity. For example:

- The websi.DB activity class provides methods, such as DB.write and DB.read for manipulating the database.
- The websi.Fetch activity class provides methods that fetch input values supplied by the user.
- The websi.Output activity class provides methods that present outputs to the user.

The set of methods of all WebSI activity classes together comprise a declarative, high-level interface between the AL and the interaction style (IS) component, as described in Sec. 2.3.

3.5 Communicating with the User

An interaction method’s responsibilities include, as discussed above, presenting the computed output to the user and requesting inputs from the user. These two activities obviously require a UI, which enables the user to view the output and to supply the required inputs.

Our programming model, supported by WebSI, employs a semantic user interface specifications approach [53]. The idea is that the programmer essentially specifies what data should be presented to the user and what kind of data the user is requested to supply. The programmer does not, however, specify any UI properties, such as which specific UI controls should be employed, which colors, fonts and graphics should be used, and what will be the geometrical layout of the UI.

The programmer defines what the UI should do, i.e., the semantics, but not how this should be done, i.e., the syntax. For example, the programmer issues an SQL SELECT query and wishes to present the resulting table to the user. The programmer requests this table to be presented to the user (i.e., what), but it is up to WebSI to decide how the table should be presented, e.g., as a tabular structure or as a set of separate records. WebSI will also automatically determine which colors and fonts will be employed and so on.

WebSI provides a programmer with a set of IO activity classes. These activity classes provide methods for coding input requests and output presentation requests. Since our programming model is centered over a relational database, one of WebSI’s main abstractions is a table (relation), i.e., an object that contains columns and rows. This abstraction
is represented through a `websi.Table` interface. Consequently, `WebSI` IO facilities, provided via the IO activity classes, are designed to handle tabular data.

In Tab. 3.1 we suggest a classification of IO activities, which are frequent in the information systems domain. The classification is suggested based on our experience in student projects, and on our everyday experience with interactive software. The classification is accompanied by the names of `WebSI` IO activity classes, which support the various IO activities.

<table>
<thead>
<tr>
<th>IO Activity</th>
<th>Description</th>
<th>IO Activity Class</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Output</strong></td>
<td>Present a single value to the user.</td>
<td><code>websi.Output</code></td>
</tr>
<tr>
<td></td>
<td>Present a table to the user. The table is possibly a result of SQL SELECT query.</td>
<td></td>
</tr>
<tr>
<td><strong>Input</strong></td>
<td>Request the user to supply a user-composed value of a certain type. The user can supply any value of the type, rather than being limited to a predefined values set. The input request is mandatory or optional. A default input value may be specified.</td>
<td><code>websi.UserComposed</code></td>
</tr>
<tr>
<td></td>
<td>Request the user to supply value(s) out of a predefined values set. The user is limited to the values in the set and cannot supply a value, that is not in the set. The input request is mandatory or optional. A default input value may be specified.</td>
<td><code>websi.Select</code></td>
</tr>
<tr>
<td></td>
<td>Enable the user to initiate an action in the system.</td>
<td><code>websi.Action</code></td>
</tr>
</tbody>
</table>

Note that an input request is qualified by the programmer as mandatory or optional. Mandatory input requests cannot be ignored by the user, i.e., `WebSI` forces the user to supply a value. Optional input requests can be ignored. Therefore, prior to fetching a value of an optional input request, the programmer must ensure (using a special `WebSI` facility) that the value has actually been supplied.

Table 3.2 presents `WebSI` output methods. Table 3.3 presents `WebSI` input methods. Each such IO method defines a certain output semantics or input semantics.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Output.scalar</code></td>
<td>Present a single value to the user</td>
</tr>
<tr>
<td><code>Output.table</code></td>
<td>Present a table to the user. The table is possibly a result of SQL SELECT query.</td>
</tr>
<tr>
<td><code>Output.error</code></td>
<td>Present a system error message to the user.</td>
</tr>
<tr>
<td>Method Description</td>
<td>Mandatory</td>
</tr>
<tr>
<td>-------------------------------------------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>Request a short user-composed value of any type</td>
<td>UserComposed.any</td>
</tr>
<tr>
<td>Request a long user-composed value of any type</td>
<td>UserComposed.anyLong</td>
</tr>
<tr>
<td>Request an answer to a yes-no question</td>
<td>UserComposed.yesNo</td>
</tr>
<tr>
<td>Request a user-composed time value</td>
<td>UserComposed.time</td>
</tr>
<tr>
<td>Request a user-composed date value</td>
<td>UserComposed.date</td>
</tr>
<tr>
<td>Request a short “secret” user-composed value (e.g., a password)</td>
<td></td>
</tr>
<tr>
<td>Request the user to select a single row out of a table. The table is possibly a result of SQL SELECT query</td>
<td>Select.one</td>
</tr>
<tr>
<td>Request the user to select one or more rows out of a table. The table is possibly a result of SQL SELECT query</td>
<td>Select.oneOrMore</td>
</tr>
<tr>
<td>Enable user to select a single cell in a table. The act of selecting a cell initiates an action in the system. The table is possibly a result of SQL SELECT query</td>
<td></td>
</tr>
<tr>
<td>Enable the user to initiate an action in the system</td>
<td></td>
</tr>
</tbody>
</table>
The un-typed user-composed values are divided into “short” and “long” values. This is done for the sake of simplicity of storing the un-typed user-composed values as strings in the database. Short values are stored as VARCHAR(50) SQL data type and long values are stored as VARCHAR(200). This simplifying decision still proved flexible enough.

It can be noted, that the Select.action semantics does not fit into the IO activities classification, suggested in Tab. 3.1. Actually, this semantics combines two input activities: value selection and action initiation. Still, this semantics proved useful. This suggests that the classification in Tab. 3.1 should be taken only as a general guide. The actual set of IO semantics should be defined by the developers of WebSI, based on their experience of using WebSI for construction of various information systems.

For each IO method WebSI generates a UI, that corresponds to the method semantics. One can, of course, imagine many possible UI implementations for each one of the methods. We have already discussed some of the possible implementations of the Output.table method, the one that presents a table to the user. Another example is asking the user to (mandatorily) select a single value out of the set of values, using the Select.one method. The possible generated UI may employ a combo box, a single-selection list box or a set of radio buttons. If a single-selection request is qualified as optional, it is possible, for example, to employ a set of radio buttons, accompanied by an extra radio button, labelled, e.g., as “unspecified”. By selecting this extra radio button the user indicates that she has decided not to supply a value for this input request.

Rationale Following our software construction methodology, we expect the use-case specification to define the flow of data between the user and the system. The programmer then merely translates the IO activities, described in the use cases, into calls to the IO methods provided by WebSI. Thus the programmer’s code for IO is driven by the specification and remains close to the specification. The benefits of code resembling the specification, such as the ease of direct verification, have already been mentioned.

By employing semantic UI specifications we fully decouple the AL from the presentational issues. The calls to the IO methods are driven by the specification and thus are related to the AL. The UI implementations of WebSI IO methods are hidden from the programmer and, therefore, can be altered without touching the use cases implementation. As a result, WebSI can, in principle, generate a UI of any type (GUI, audio-based, command-line etc.) for any platform (desktop, mobile device etc.) without changing or even recompiling the use cases implementation code. To conclude, WebSI makes the AL implementation platform- and UI type-independent.

Currently, WebSI provides an ordered way of altering the set of IO methods, in order to accommodate future needs.

3.6 Database Manipulation

Manipulating the database is performed via two methods of the DB activity class: read and write. Technical details of connecting and disconnecting from the database, managing database resources and transactions are hidden from the programmer. Table 3.4 summarizes the WebSI database manipulation methods.

Since the database is intended to store business data, it is imperative, that these data are legal and consistent at any time. For this purpose, the database is accessed only via database transactions, that are opened and committed automatically by WebSI. It is, however, the programmer’s responsibility to design the SQL queries in a transaction in
Table 3.4: WebSI database manipulation methods

<table>
<thead>
<tr>
<th>Method</th>
<th>SQL Queries, that Can Be Issued</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB.read</td>
<td>SELECT</td>
<td>websi.db.Table, that contains the results of the query</td>
</tr>
<tr>
<td>DB.write</td>
<td>INSERT, DELETE, UPDATE</td>
<td>Number of rows, affected by the query</td>
</tr>
</tbody>
</table>

such a way, that they transform the database from a given consistent state (at the beginning of the transaction) to a new consistent state (at the end of the transaction).

Each interaction method execution, i.e., each IO sequence is associated with a separate transaction:

- When an interaction method is invoked, the first SQL query it issues opens a new database transaction.
- When the interaction method ends, WebSI tries to commit the transaction, i.e., to commit all the SQL queries, the programmer has issued during the interaction method execution.
- Should the commit fail, WebSI rolls the transaction back and sends an error message to the user. It is also desirable to have a possibility of letting the programmer handle a commit failure in the code (currently not supported).

To conclude, all the SQL queries, issued during an interaction method execution, are performed in a separate database transaction.

Current database systems are capable of checking various integrity constraints on the database schema, such as PRIMARY KEY, UNIQUE, FOREIGN KEY or CHECK constraints. In case such a constraint is violated by an SQL query, issued during the execution of an interaction method, WebSI encapsulates the violation into a Java exception object. The exception is of one of WebSI exception classes, according to the violated constraint type. The exception is then delivered to the programmer, so that an appropriate action can be coded. WebSI exception classes for database constraints violations include:

- websi.exceptions.PrimaryKeyOrUniqueViolationException
- websi.exceptions.ForeignKeyViolationException
- websi.exceptions.CheckViolationException
- websi.exceptions.NotNullViolationException

In case a database constraint violation occurs, and the resulting exception is not caught and handled by the programmer, WebSI automatically rolls back the whole ongoing transaction and sends an error message to the user. If the programmer handles the database constraint violation exception, only the violating SQL query is rolled back; the ongoing transaction proceeds normally.
Rationale SQL is a powerful query language, that is declarative, rather than operative, i.e., a semantic, rather than a syntactic language. Using SQL to implement the AL allows the programmer to stay focused on application semantics, rather than on low-level implementation and data management details. The SQL queries, of course, are not part of a use-case specification, they are the implementation of the specification. Still, the fact that SQL is a high-level declarative notation contributes to the ease of verification by code inspection.

To perform an SQL query the programmer needs to specify only the query itself. Technical database manipulation details are hidden, and that is in line with our attempts the make the programmer’s code deal only with AL.

Associating each interaction method execution (i.e., each IO sequence) with a separate transaction proved to be the simplest way of database transactions management. The programmer does not have to manually indicate when a new transaction should start and when it should be committed—all these, including the handling of commit failure, are performed by WebSI. We believe, that this model is sufficient for many systems.

Sometimes it may be desirable to have “long” transactions, that span several IO sequences, or even transactions that span over the execution of several different use cases. In that case, however, the programmer would have to manually demarcate the transactions. We decided to prefer programming simplicity over a full flexibility, and provided a simple transactions model, that is still flexible enough.

Database constraints are a simple and efficient way of ensuring the integrity and correctness of the data stored in the database. The constraints are checked automatically by the database engine and the programmer benefits from less coding. The programmer also has the ability to react to a database constraint violation in her code, which is supported through a standard exceptions mechanism.

3.7 Use-Case Relationships

UML defines several kinds of relationships between use cases: <<include>> , <<extend>> and <<generalize>>. Since we strive to make the programmer’s code as close as possible to the specification, we attempt to map these relationships into the programmer’s code. Currently WebSI supports a single relationship between use cases. This relationship is intended to be similar to the <<include>> relationship, although the semantics is not identical.

Let us first briefly describe the semantics of the UML <<include>> relationship, as defined in [58]. In this relationship, a use case, called a base use case, completely incorporates the behavior of another use case, called an inclusion use case. The execution of the inclusion use case starts at a point in time, defined by the base use case. When the inclusion use case ends, the control is returned to the base use case. The inclusion use case in this relationship cannot be instantiated, instead it explicitly describes an additional behavior that is inserted into the base use case instance. The inclusion use case may access attributes and operations of the base use case. The base use case cannot access attributes or operations of the inclusion use case. The inclusion must happen always.

The WebSI “<<include>>” relationship differs from the UML’s one in the following points:

- An inclusion use case may be initiated directly by the user, not necessarily by a base use case.
• An inclusion use case cannot access attributes or operations of a base use case. A base use case can pass a parameter value to an inclusion use case.

• An inclusion use case, when its behavior ends, can return a result value to a base use case.

• The behavior of the inclusion use case can be executed any number of times, including zero.

The UML <<include>>, <<extend>>, <<generalize>> and the WebSI <<include>> relationships have many similarities. For example, they all add behavior to the initial use case. Table 3.5 compares some key properties of the four relationships. Most of this table originates from the UML reference [58].

<table>
<thead>
<tr>
<th>Property</th>
<th>&lt;&lt;extend&gt;&gt;</th>
<th>&lt;&lt;include&gt;&gt;</th>
<th>&lt;&lt;generalize&gt;&gt;</th>
<th>WebSI &lt;&lt;include&gt;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base behavior</td>
<td>Base use case</td>
<td>Base use case</td>
<td>Parent use case</td>
<td>Base use case</td>
</tr>
<tr>
<td>Added behavior</td>
<td>Extension use case</td>
<td>Inclusion use case</td>
<td>Child use case</td>
<td>Inclusion use case</td>
</tr>
<tr>
<td>Direction of reference</td>
<td>Extension use case</td>
<td>Base use case</td>
<td>Base use case</td>
<td>Base use case</td>
</tr>
<tr>
<td>Which use case is instantiated?</td>
<td>The base use case. It must be well-formed without the extension</td>
<td>The base use case. It may or may not be well-formed without the inclusion</td>
<td>To obtain the effects of the addition, the child must be instantiated. The parent may be abstract.</td>
<td>The base use case, unless it is an inclusion use case itself. In that case it may be included by another base use case</td>
</tr>
<tr>
<td>Can the addition be instantiated?</td>
<td>No</td>
<td>No</td>
<td>Not necessarily. The child may be abstract</td>
<td>Not necessarily. An inclusion use case may have to be initiated by a base use case only</td>
</tr>
<tr>
<td>Can the addition access attributes and operations of the base?</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes (by usual inheritance mechanisms)</td>
<td>No. The base use case may supply a parameter value to the inclusion use case</td>
</tr>
<tr>
<td>Can the base see the addition?</td>
<td>No</td>
<td>The base sees the inclusion and may depend on its effects but may not access it</td>
<td>No</td>
<td>No. The inclusion use case may return a result value to the base use case</td>
</tr>
<tr>
<td>Repetition</td>
<td>Depends on the condition at the base use case</td>
<td>Exactly one. Loops in the behavior of the base use case can achieve other multiplicities</td>
<td>Child controls its own execution</td>
<td>Any number, depends on the base use case</td>
</tr>
</tbody>
</table>

When starting the inclusion of a use case, the programmer usually specifies the interaction method of the base use case, which will be invoked when the inclusion use case ends. We call this method a return-to method. In this way, the inclusion use case returns control back to the base use case. If a return-to method is not specified, then, when the inclusion ends, the control will not return to the base use case, i.e., the base use case will end as well.
The execution of an inclusion use case begins from its start interaction method.

The interaction method of a base use case, that initiates the inclusion, may issue some SQL queries. The start method of an inclusion use case may do so as well. Each one of these two interaction methods will be associated with its own separate database transaction. The transaction of the base use case’s interaction method will be committed before the start interaction method of the inclusion use case is invoked.

In the code of an inclusion use case, at a desired point, the programmer indicates that the inclusion should end. If a return-to interaction method of the base use case has been specified, WebSI will pass the control back to the base use case by executing the return-to method. In any case, the programmer does not indicate a specific base use case to return to.

WebSI supports inclusion chains of arbitrary depth, however, it does not support circles in these chains.

Rationale Mapping standard UML relationships directly into the programmer’s code contributes to the closeness of the code to the specification. The benefits of this closeness have been already discussed.

Despite the differences in semantics, both WebSI and UML <<include>> relationships are intended to encapsulate some common behavior in an inclusion use case, and to reuse that behavior from several base use cases.

### 3.8 The Use-Case Displayer

Since a WebSI-based application is essentially a set of use-case classes, we obviously need a kind of a component that “glues” all the use cases together into one application. For that purpose, WebSI provides a ready-made component, called a use-case displayer [37]. The use-case displayer’s primary function is to visualize the use cases of the application to the user, and to enable the user to execute them.

At deployment time, it is required to specify which use-case classes should be deployed. Based on this information, WebSI uses the use-case displayer to group all the use-case classes into a ready-to-use information system.

Rationale The UML use-case specification does not mention how the execution of use cases is initiated. When employing a traditional development process, such as USDP, it is required to design an environment and a UI, which enable the execution of all the use cases. WebSI strives to keep the implementation as close as possible to the use-case specification. Therefore, the programmer has only to implement the use-case classes and she does not have to develop an environment that supports the use-case execution. The ready-made use-case displayer is such an environment.

Currently, WebSI groups all the use cases into a single use-case displayer. For the future research it may be suggested to divide all the use cases into packages [58] and to provide a separate displayer for each package. Another possibility is to provide a separate use-case displayer for each actor [58], containing only the use cases relevant to this actor. The latter possibility fits well the current WebSI security model (see Sec. 3.11).
3.9 Interaction Styles and Automatic User Interface Generation

As mentioned in Sec. 3.5, it is up to WebSI to decide what UI will be employed for each one of its IO methods. These decisions are encapsulated in a WebSI component, called interaction style (IS) [38].

The IS component specifies all the properties of the UI, such as: which specific controls should be employed, the geometrical properties of these controls, their color, their layout on the screen, the fonts and the colors of the text and so on. The programmer has only to select one of a number of different IS’s offered by WebSI, and the system will automatically produce a UI using the selected IS.

If a customer is unhappy with the UI appearance, it is required to exchange the employed IS with another one. This operation involves only changing the application’s deployment settings. Use-case implementations need not be altered or recompiled.

Rationale Separating UI definitions from the AL code, by employing IS’s, makes WebSI-based code environment-independent. In order to employ another kind of hardware, middleware or another UI platform for WebSI applications, it is required to develop an IS, that employs the new environment’s UI facilities. It is thus possible, in principle, to employ WebSI for platforms, that use an audio-based or a command-line UI rather than a GUI, when appropriate IS’s have been developed.

Developing a completely new IS is not a trivial task. It includes providing UI implementations for each one of the IO methods and defining the appearance of the use-case displayer. The IS developer must, therefore, be well familiar with the presentational facilities of the platform, for which the IS is being developed. For example, if a standard Web browser is employed, the developer must be familiar with such Web technologies as HTML [20], CSS [10] and JavaScript [31]. In addition, the IS developer must have good UI design skills. Finally, a usability expertise is required to pick the appropriate colors, fonts, graphics, to define a proper geometrical layout and so on.

Once developed, the IS can be reused in many applications. Software manufacturers may develop IS’s that support their UI and usability standards, ensuring a unified appearance of all their products.

It is noted, that we bias our programming model towards the client programmer. We strive to make the client programmer’s job of coding the AL as simple as possible and free of anything not related to the AL. Developing a new IS is, indeed, labor-intensive, but this activity can be performed independently of a specific information system, by UI specialists.

3.10 Managing Multiple Users

The WebSI programming model strives to provide a single-user abstraction. This means, that the programmer writes most of her code as if a single user is operating the system. The various details of managing multiple system users are hidden.

For each user that starts operating the information system, WebSI creates (on-demand) a separate set of use-case objects (i.e., instances of the use-case classes), which are associated with this user only. Therefore, the programmer can design her use-case classes as if a single user accesses them, from within a single thread. For example, data members
of a use-case class can be manipulated without any need for synchronization, since each WebSI user will have her own instances of these data members.

In WebSI, each user that operates the system, is given her separate private “world”, called a session [25]. We see a session as a container for objects, which serve a specific user. Use-case objects of a certain user reside in her session. In addition, WebSI allows to store in a session helper session objects of custom classes, defined by the programmer. Each user will then have an on-demand, separate instance of the custom class, i.e., her own session object. Accessing session objects is also performed without any need for synchronization.

The only place, where the programmer has to be aware of the multi-user nature of the system is, of course, the database. When performing a transaction, the programmer, still can act as if she is the only user of the database. Here we rely on database engines’ ability to manage concurrent transactions. However, between two different transactions (even on behalf of the same user), the programmer cannot assume, of course, that the database state has not changed.

The described architecture is illustrated on Fig. 3.1.

![Diagram](image)

Figure 3.1: WebSI multi-user support

**Rationale**  The single-user abstraction proves useful because of its simplicity, and because programmers that employ WebSI services are not required to have any parallel programming skills.

The data members of a use-case class can be used by the programmer to hold intermediate data, that should be preserved during the use-case execution, i.e., between IO sequences. Therefore, the use-case class data members’ values comprise a state of a use-
case object. This state can of course be different for each user that is executing the use case. Thus it seems natural to us to have a separate set of use-case objects for each user. This supports well the abstraction of a single user operating the system.

Sessions and session objects are, of course, not our invention. Most of the support for these we receive from the underlying J2EE software, on top of which the WebSI programming model is implemented. It proved very natural to see a session as an abstraction of the user’s private “world”.

3.11 Security

In many cases some of the information system’s services should be given only to authorized users. As with other aspects of the information system, not directly related to AL, WebSI attempts to hide all the code, that handles authorization and security issues.

WebSI takes the Role-Based Access Control (RBAC) approach [15, 49], that in our case is use case-oriented. The idea is, that the use cases are divided into public and secure use cases. Public use cases can be executed by anyone. Secure use cases can be executed only by authorized users, according to their security roles. During the deployment, all the security roles are specified. In addition, for each secure use case it is specified, users of which security roles are authorized to execute this use case. Public use cases are not associated with security roles. Usernames and passwords for the various security roles are stored in the database.

Given all this information, WebSI performs the following:

- Generates the UI required for “Login” and “Logout” operations to and from the various security roles.
- Verifies usernames and passwords during “Login” operations.
- Enables the user to execute only the use cases applicable to the security role, the user has logged in to.
- WebSI enables the programmer to obtain at any time the username and the security role of the logged-in user (if any).

Rationale  We believe that in most cases the security aspect of the information system is completely orthogonal to its AL. The exception is when some AL task requires the user’s identity. Consider for example, a theater events management system, that includes storing event orders for its users. In order to obtain all the orders of a certain user, the programmer must write code, that obtains the user’s identification details and uses them to search for the orders of that user.

In any case, the details of logging in to the system (authentication) and logging out of the system, and the details of granting access to the secure use cases according to the security role are not related to the AL, and, therefore, are hidden by WebSI.

It is up to the programmer to decide, which security roles to define, and how to divide the users between the security roles. One possible way is to define a security role for each actor of the use-case specification.
3.12 Verification

One way to perform verification is by testing. This includes executing test scenarios prepared in advance and comparing the actual outputs to the desired ones. A test scenario can span one or several use cases. The use-case model is usually a starting point to developing the various test scenarios.

In some cases, it is useful to perform a direct verification of a WebSI-based code. The closeness of the code to the natural language use-case specification makes it possible to perform code inspections, comparing the code to the specifications. The closeness of the code to the specifications also makes it easier to correct bugs, which are discovered during testing.

3.13 Modification and Extension

A use-case object, at runtime, usually does not interact with other use-case objects (except for the <<include>> relationship). It usually interacts only with the user, the database and, possibly, with session objects. Therefore, the programmer, that implements the use case, has to be familiar with the use-case specification and with the database schema, but not with the details of other use cases. As a result, the effort of modifying a use case or extending the information system with an additional use case, can be expected to be independent of the number of already existing use cases. Another way to formulate this result, is to say that the extension complexity \[O(1)\] of the architecture is \[O(1)\].

Rationale The achievement of an \(O(1)\) extension complexity is based on two assumptions. The first assumption is that the database schema is not changed. This assumption is probably often met, as observed by [21] that the schema of a database changes much slower than applications employing the schema. The second assumption is that it is possible to design the use cases such that they need not interface with other use cases. This assumption was met in our student projects, but we do not know whether it is true in general.

In many cases, the use cases, although not interfacing directly, do communicate implicitly, through common database data. Consider, for example, a work-flow, in which a worker in a company issues a purchase request (one use case). The request must be approved by two financial clerks, who may work in parallel (another use case, executed twice by different users). The approved request is executed by a purchasing clerk (an additional use case). Our experience suggests, that such use cases have to be designed, implemented and verified together, so that they cooperate correctly.
Chapter 4

Detailed Programming Model

4.1 Use-Case Classes

Each use case of the system is implemented in a separate class, called a use-case class. A use-case class is derived from the abstract websi.UseCase class. A use-case class must be public and must have a public default constructor. Instances of use-case classes are called use-case objects.

4.2 Interaction Methods

An interaction method is a method of a use-case class that handles one IO sequence. An interaction method is invoked automatically by WebSI when an IO sequence begins. An interaction method must have the following signature:

```java
public void methodName() throws Exception;
```

Each use-case class must implement an interaction method, named start. This method is invoked automatically by WebSI when the use-case execution starts.

The code of interaction methods employs WebSI services in order to perform the various activities, such as:

- Communicating with the user, i.e., requesting inputs, presenting outputs and fetching input values supplied by the user
- Manipulating the database
- Manipulating session objects
- Obtaining the user’s identification details
- Including an inclusion use case and returning to the base use case

The WebSI services for performing the various activities are accessed in the following ways:

- By invoking static methods of WebSI activity classes. An activity class is an abstract class providing static methods that support a certain activity, such as manipulating the database or communicating with the user
- By invoking the inherited methods of the UseCase class
- By using WebSI helper classes from the websi.util package
In the following sections we will describe all WebSI services available to the client programmer. Additional information is available in WebSI JavaDocs.

4.3 Communicating with the User

WebSI provides several activity classes that enable the programmer to communicate with the user. We call these activity classes IO activity classes. These activity classes follow the classification of the input and output activities suggested in Sec. 3.5. There is an websi.Output activity class that supports the activities of presenting outputs to the user. The websi.Input activity class does not provide any methods, but rather serves as a marker, i.e., other classes that support input activities inherit from the Input class. These other classes are:

- websi.UserComposed—provides methods that request the user to supply user-composed values
- websi.Select—provides methods that request the user to select values out of predefined value sets
- websi.Action—provides a method that enables the user to initiate actions

In addition, the websi.Fetch activity class enables the programmer to fetch the input values supplied by the user. (We do not consider this class an IO activity class, since its methods do not generate any UI content.) Figure 4.1 shows the relationships between the activity classes described above. In the following sections we will describe the API of the mentioned activity classes.

4.3.1 The Output Activity Class

This class provides methods that present outputs to the user:

![Figure 4.1: WebSI activity classes that support communicating with the user](image-url)
public static void scalar(String text, String caption);
Presents a scalar value (i.e., some text) to the user.

public static void table(Table table, String[] visible_columns, String caption);
Present the contents of a websi.Table to the user.
visible_columns—the names of table's columns, that will be visible. Column names in WebSI are case-insensitive.

public static void error(String msg, Exception exc);
Presents a system error message to the user. This method is not supposed to be used by the client programmer.

4.3.2 The UserComposed Activity Class

This activity class provides methods that enable the user to supply user-composed values. Supplying a user-composed value means that the user can specify any value, rather than being limited to a predefined values set. In some cases, however, the user is limited to a certain values type.

All input request methods in WebSI share some common properties:

- An input request is associated by the client programmer with a textual ID, called a fetch ID. A fetch ID must be unique within all input requests of an IO sequence (this is checked by WebSI). A fetch ID is used later (in the next IO sequence) by the programmer in order to fetch the input value(s), supplied by the user.

- An input request can be mandatory or optional. When an input request is mandatory, the user must supply an input value. The employed IS will ensure this and the client programmer can be sure that an input value will be supplied. If an input request is optional, the employed IS will give the user an option of not supplying a value. Therefore, for an optional input request, prior to fetching the input value, the programmer must ensure that a value has been supplied. This is performed by calling a special method of the Fetch activity class (see Sec. 4.3.4).

- An input request may be supplied with a default input value.

Here are the methods of the UserComposed activity class:

public static void any(String fetch_id, String default_value, String caption);

public static void anyOptional(String fetch_id, String default_value, String caption);

Enables the user supply a short user-composed value of any type. The length of the supplied value is up to websi.PublicConstants.SHORT_TEXT_LENGTH (currently 50) characters.

public static void anyLong(String fetch_id, String default_value, String caption);

public static void anyLongOptional(String fetch_id, String default_value, String caption);
Enables the user supply a long user-composed value of any type. The length of the supplied value is up to `PublicConstants.LONG_TEXT_LENGTH` (currently 200) characters.

```java
public static void secret(String fetch_id, String caption);
```

Enables the user to specify a “secret” (short) user-composed value (e.g., a password). This input request has only a mandatory version and no default value.

```java
public static void yesNo(String fetch_id,
    Boolean default_value, String caption);
public static void yesNoOptional(String fetch_id,
    Boolean default_value, String caption);
```

Enables the user to answer a yes-no question.

```java
public static void date(String fetch_id,
    java.sql.Date default_value, String caption);
public static void dateOptional(String fetch_id,
    java.sql.Date default_value, String caption);
```

Enables the user to supply a date. Note: if the input request is mandatory and no default value is specified, the default value will be the current date.

```java
public static void time(String fetch_id,
    java.sql.Time default_value, String caption);
public static void timeOptional(String fetch_id,
    java.sql.Time default_value, String caption);
```

Enables the user to supply time. Note: if the input request is mandatory and no default value is specified, the default value will be the current time.

### 4.3.3 The Select Activity Class

This activity class provides methods that enable the user to select values out of predefined sets. The user is limited to the predefined set and cannot supply a value outside the set. Here are the methods:

```java
public static void one(String fetch_id, Table table,
    String[] visible_columns,
    String return_values_column,
    String default_value, String caption);
public static void oneOptional(String fetch_id, Table table,
    String[] visible_columns,
    String return_values_column,
    String default_value, String caption);
```

Enables the user to select a single row out of the supplied `Table`. For a mandatory request, the `Table` must contain at least one row (`WebSI` checks this).

- `visible_columns`—the names of the `Table`'s columns that will be visible to the user.
- `return_values_column`—the name of the `Table`'s column, that will be used for return values, i.e., if the user selects the `i`-th row of the `Table`, the programmer will fetch the value, that is stored in the `i`-th row, in the column, whose name is `return_values_column`.
- `default_value`—a value that identifies the row that will be selected be default. This
value must come from a column that serves for return values, as specified by the previous parameter.

```java
public static void oneOrMore(String fetch_id, Table table,
               String[] visible_columns,
               String return_values_column,
               String[] default_values, String caption);
```

```java
public static void oneOrMoreOptional(String fetch_id, Table table,
               String[] visible_columns,
               String return_values_column,
               String[] default_values, String caption);
```

Enables the user to select one or more rows out of the supplied `Table`. For a mandatory request, the `Table` must contain at least one row (WebSI checks this).

```java
public static void action(String fetch_id, Table table,
              String[] visible_columns,
              String[] choice_columns,
              String[] return_values_columns,
              String[] interaction_method_names,
              String caption);
```

Enables the user to select a single cell (not a row) out of the supplied `Table`. The act of selecting a cell also initiates an action in the system.

- `visible_columns`—the names of the table’s columns that will be visible to the user
- `choice_columns`—columns of the table, whose cells can be selected. These columns are always visible, even if not listed in the `visible_columns` parameter
- `return_values_columns`—columns of the table, whose values will serve as return values. For example, if the user selects a cell in the 3rd row, in the column, whose name is the 4th element of the `choice_columns` array, the programmer will fetch the value in the 3rd row, in the column, whose name is the 4th element of the `return_values_columns` array. The size of `return_values_columns` array must be identical to the size of the `choice_columns` array
- `interaction_method_names`—specifies the interaction method, which will be invoked when the user selects a cell. For example, if the user selects a cell in the column, whose name is the 4th element of the `choice_columns` array, WebSI will invoke the interaction method (of the current use-case class), whose name is the 4th element in the `interaction_method_names` array. The size of `interaction_method_names` array must be equal to the size of the `choice_columns` array

### 4.3.4 The Fetch Activity Class

This activity class provides methods for fetching input values supplied by the user.

Prior to fetching a value of an optional input request, the programmer must ensure that the value has actually been supplied. This is performed with the following method of the Fetch activity class:

```java
public static boolean inputSpecified(String fetch_id);
```

If this method returns `true`, the value for the passed `fetch_id` can be fetched. For mandatory input requests, this method always returns `true`, and, therefore, may not be used.

Fetching input values in `WebSI` is type-safe. When a programmer wishes to fetch an input value for a certain `fetch ID`, the programmer specifies the type, in which she wishes
to obtain the value. WebSI then tries to convert the value to the desired type. Should the conversion fail, WebSI aborts the currently executing interaction method and sends an automatic error message to the user, requesting her to resupply the inputs. (In addition, WebSI rolls back the ongoing database transaction.) The programmer does not check or handle type conversion errors, she just calls the desired fetch method, knowing that the method will succeed.

Now we list the methods of the Fetch activity class that fetch user input values and convert them to the desired type. Each method comes in two versions: for single-value inputs and for multiple-value inputs. When the user can supply more than one value for an input request (e.g., for Select.oneOrMore request), the multiple-value version should be used. If the single-value version is used in this case, it will return one of the multiple values. When the user can supply just one value for an input request, both versions can be used.

```java
public static String string(String fetch_id);
public static String[] strings(String fetch_id);

public static int Int(String fetch_id);
public static int[] Ints(String fetch_id);

public static long Long(String fetch_id);
public static long[] Longs(String fetch_id);

public static double Double(String fetch_id);
public static double[] Doubles(String fetch_id);

public static boolean Boolean(String fetch_id);
public static boolean[] Booleans(String fetch_id);

public static java.sql.Date date(String fetch_id);
public static java.sql.Date[] dates(String fetch_id);

public static java.sql.Time time(String fetch_id);
public static java.sql.Time[] times(String fetch_id);

public static java.sql.Timestamp timestamp(String fetch_id);
public static java.sql.Timestamp[] timestamps(String fetch_id);
```

The methods listed above never return null and arrays returned by multi-value input fetch methods never contain nulls. The Strings returned by string and strings methods are always trimmed.

### 4.4 Manipulating the Database

The websi.DB activity class provides two methods that enable the client programmer to read and write the database:

```java
// For SELECT queries
public static Table read(String sql_select_query);
// For INSERT, UPDATE and DELETE queries
public static int write(String sql_query)
    throws DBConstraintViolationException;
```

The client programmer is not expected to issue any other SQL queries from the code, other than SELECT, INSERT, UPDATE and DELETE queries. This means that the programmer cannot, for example, modify the database schema.
### 4.4.1 Reading the Database

The `read` method returns an object of a class, that implements a `Table` interface. This interface represents a table with rows and columns, and defines methods that allow inspecting the table. The methods of the `Table` interface follow.

```java
// columns and rows are numbered starting from 1
// column names are case-insensitive

public String getString(int row, int column);
public String getString(int row, String columnName);

public int getInt(int row, int column);
public int getInt(int row, String columnName);

public long getLong(int row, int column);
public long getLong(int row, String columnName);

public double getDouble(int row, int column);
public double getDouble(int row, String columnName);

public boolean getBoolean(int row, int column);
public boolean getBoolean(int row, String columnName);

public Date getDate(int row, int column);
public Date getDate(int row, String columnName);

public Time getTime(int row, int column);
public Time getTime(int row, String columnName);

public Timestamp getTimestamp(int row, int column);
public Timestamp getTimestamp(int row, String columnName);

public Object getObject(int row, int column);
public Object getObject(int row, String columnName);

public int getColumnsNr();
public int getRowsNr();
public String getColumnName(int column);

// Column labels may potentially be different from column names,
// and are used in various printouts
public String getColumnLabel(int column);
public String getColumnLabel(String columnName);
```

The various `getSomeType` methods of the `Table` interface try to convert the table values to a requested type. In case such a conversion fails, a runtime exception is thrown. Tables returned by the `read` method are immutable and can be stored locally, i.e., in data members of a use-case class.

Currently, any two `Table` objects in `WebSI` can be tested for equality, using the standard Java method:

```java
public boolean equals(Object otherObj);
```

Two `Table` objects are considered equal if:

- they have the same number of rows and columns and
  - the objects in their corresponding cells are equal, when applying the `equals` method on them, or
– the string representations of these objects (obtained by applying `toString`) are identical

We are employing such non-standard definition for equality based on feedback from the students.

### 4.4.2 Writing to the Database

The `write` method returns the number of rows affected by the query. In addition, the `write` method can throw an exception, indicating that a *database constraint* has been violated.

Database constraints are defined over a database schema in order to preserve the integrity of the data. Database constraints include NOT NULL, PRIMARY KEY, UNIQUE, FOREIGN KEY and CHECK constraints. In case a database constraint has been violated, the database engine raises an error, which is noted by *WebSI* and delivered to the client programmer as a `websi.exceptions.DBConstraintViolationException` object.

The `DBConstraintViolationException` class has currently four subclasses (all located in `websi.exceptions` package), indicating specific violated constraints:

- `PrimaryKeyOrUniqueViolationException`
- `CheckViolationException`
- `ForeignKeyViolationException`
- `NotNullViolationException`

The described hierarchy is presented on Fig. 4.2. The constraint-specific classes have an API, that provides some more information on the violated constraint. This API is quite simple and is presented on Fig. 4.2. The only thing we mention here is that all Strings, returned by the constraint-specific classes API’s, are guaranteed to be in lower case.

It should be noted, that database constraints violation exceptions are the only *WebSI* system exceptions, that the programmer may handle in the code. The programmer should
not try to handle such system exceptions as websi.exceptions.WebSIException or java.sql.SQLException.

4.4.3 Recommended SQL Datatypes

For the sake of simplicity and database vendor independence, WebSI recommends to employ only the SQL datatypes presented in Tab. 4.1. Appropriate fetch methods of Table and Fetch classes are also presented. We believe, that for academic needs the presented datatypes set is sufficient.

Note, that all the textual data (VARCHAR) is treated either as a short data or as a long data. Short data is up to PublicConstants.SHORT_TEXT_LENGTH (currently 50) characters long. Long data is up to PublicConstants.LONG_TEXT_LENGTH (currently 200) characters long. When requesting input with the help of UserComposed.any, UserComposed.anyOptional and UserComposed.secret methods, WebSI ensures that the values supplied by the user are of a proper length (currently 50 characters), so that inserting them into VARCHAR(50) columns does not cause a database error. How this is performed, depends on the IS, e.g., by not allowing the user to supply more than 50 characters or by truncating strings that are too long. Same is true for the UserComposed.anyLong and UserComposed.anyLong methods and VARCHAR(200) columns.

The names of the datatypes are as in the database product we use currently—Cloudscape 4.0.6, supplied together with J2EE 1.3.1.

<table>
<thead>
<tr>
<th>Name</th>
<th>Java class or datatype</th>
<th>Table methods</th>
<th>Fetch methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOLEAN</td>
<td>boolean</td>
<td>boolean boolean(...)</td>
<td>boolean Boolean(...)</td>
</tr>
<tr>
<td>VARCHAR(50)</td>
<td>String</td>
<td>String getString(...)</td>
<td>String string(...)</td>
</tr>
<tr>
<td>VARCHAR(200)</td>
<td>int</td>
<td>int getInt(...)</td>
<td>int Int(...)</td>
</tr>
<tr>
<td>INTEGER</td>
<td>long</td>
<td>long getLong(...)</td>
<td>long Long(...)</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>double</td>
<td>double getDouble(...)</td>
<td>double Double(...)</td>
</tr>
<tr>
<td>DECIMAL (precision, scale)</td>
<td>double</td>
<td>double Doubles(...)</td>
<td>double Doubles(...)</td>
</tr>
<tr>
<td>DATE</td>
<td>java.sql.Date</td>
<td>Date getDate(...)</td>
<td>Date date(...)</td>
</tr>
<tr>
<td>TIME</td>
<td>java.sql.Time</td>
<td>Time getTime(...)</td>
<td>Time time(...)</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>java.sql.Timestamp</td>
<td>Timestamp getTimestamp(...)</td>
<td>Timestamp timestamp(...)</td>
</tr>
</tbody>
</table>

Table 4.1: Recommended SQL datatypes

4.4.4 Database Transactions

All SQL queries, issued by the programmer during an interaction method execution, are performed in a single separate transaction. The transaction is opened automatically by
*WebSI* when a first SQL query during the interaction method execution is issued. When the interaction method ends, *WebSI* commits the transaction. Should the commit fail, *WebSI* rolls the transaction back and sends an error message to the user.

When a database constraint is violated and an appropriate exception is caught by the client programmer, only the violating query is rolled back, the ongoing transaction can continue normally. If the exception is not caught by the programmer, *WebSI* catches it and rolls the whole ongoing transaction back, sending an error message to the user.

In some situations, the client programmer wishes to manually roll the ongoing transaction back. This is performed by the following method, provided by the *DB* activity class:

```java
public static void rollbackTransaction();
```

When this method is called, the whole ongoing transaction is rolled back. Issuing an additional SQL query will automatically open a new transaction.

### 4.5 Multi-User Support

In *WebSI*, each user that operates the information system has its own private “world” inside the system. This “world” is called a *session*. Session can be thought of as a container of objects, that belong to a specific user operating the system.

Each user has her own set of use-case objects (i.e., instances of the use-case classes) inside her session (see Fig. 4.3). The use-case objects are created on-demand by *WebSI* and stored in a session that belongs to a specific user. The data members of the use-case

![Figure 4.3: WebSI multi-user support](image-url)
objects can, therefore, be used safely by the programmer, without a need to synchronize an access to them.

A new session is created automatically when a new user starts operating the system. There are two ways of ending a session:

- When a timeout (set by a J2EE server) expires, the session ends.
- The client programmer can end a session in the code, using the following method of the `websi.Session` activity class:

  ```java
  public static void end();
  ```

  After this method is invoked, and the currently executing interaction method ends, the current session is destroyed.

When a session ends, all the data stored in it, such as the use-case objects, becomes inaccessible.

### 4.5.1 User-Defined Session Objects

Besides the use-case objects, it is possible to store in the session various helper objects, of user-defined classes. These objects are called session objects (see Fig. 4.3).

As with use-case objects, each user will have in her session her own object of a particular user-defined class.

User-defined session object classes are derived from the `websi.SessionObject` class. As with use-case classes, session object classes must be public and they must have a public default constructor. Session objects can employ all `WebSI` services, just as use-case objects.

In order to access the session object, the client programmer uses the following method, provided by the `Session` activity class:

```java
public static SessionObject object(Class obj_class);
```

This method returns the session object of the supplied class. The returned session object belongs to the same user as the use-case object, whose code invokes the method. If a session object of the supplied class has not been created yet for this user, `WebSI` creates it automatically, by instantiating the session object class. The resulting `SessionObject` should be downcasted to the needed class.

It can be noted from the method signature, that at most one object of a class can be stored in a session.

### 4.6 Implementing the UML `<include>` Relationship

According to UML, the `<include>` relationship involves two use cases: the base use case and the inclusion use case. The base use case, at some point during its execution, completely incorporates the behavior of the inclusion use case. The inclusion must happen always. In addition, the inclusion use case cannot stand alone, i.e., its execution cannot be directly initiated by an actor.

In `WebSI` we adopt a more relaxed model (although we still call it `<include>`). We see the inclusion as an extended function call. The base use case initiates an execution of the inclusion use case, possibly passing a parameter to it. When the inclusion use case execution ends, the inclusion use case can return some result to the base use case. The
inclusion does not have to happen always. It is not forbidden for the inclusion use-case execution to be initiated directly by an actor.

In order to include a use case, the programmer uses the following method of the UseCase class:

```java
protected void includeUseCase(Class uc_class,
                               Object inclusion_param,
                               String interaction_method_name);
```

The parameters to this method are as follows:
- `uc_class`—the class of the use case to be included
- `inclusion_param`—an inclusion parameter (or `null` if not needed)
- `interaction_method_name`—the name of an interaction method of the current use-case class, that will be invoked when the inclusion ends. If the base use case is not required to continue its execution after the inclusion ends, this parameter should be `null`

Typically, this method is called right before the current interaction method ends.

The execution of the inclusion use case starts from its `start` method. The inclusion use case can fetch the inclusion parameter passed to it, via the following method of the UseCase class:

```java
protected Object getInclusionParam();
```

When the inclusion use case wishes to return control to the base use case, it uses the following method of the UseCase class:

```java
protected void endInclusion(Object inclusionResult);
```

This method is also typically called right before the current interaction method ends. Note, that the included use case does not have to know specifically who is the base use case, i.e., which use case has included it.

When the base use case gets control back, it can fetch the inclusion result passed by the inclusion use case, using the following method of the UseCase class:

```java
protected Object getInclusionResult();
```

`WebSI` does not support cycles in inclusion chains.

### 4.7 Debug Printouts

In order to print a debug message the following method should be used:

```java
Debug.print("A debug message");
```

If the debug printouts are enabled (this is specified at deployment time), the message is printed to the standard output of the J2EE server.

### 4.8 WebSI Security

`WebSI` adopts a RBAC (Role-Based Access Control) [15, 49] security model, which is, in addition, use case-oriented. During the deployment of the application, a set of `security roles` is defined (see Sec. A.6.2). Then, for each use case it is specified, users of which security roles are authorized to execute this use case. Use cases that are not associated with security roles, are considered public and can be executed by anyone. After the application is deployed, `WebSI` automatically generates the UI for “Login” and “Logout” operations, checks the usernames and passwords, and grants access to the use cases, according to the user’s security role.

The client programmer, in the code, can obtain the username and the security role of the user, that is currently logged in. This is performed using the following methods of the
websi.UserID activity class:
   // returns null if no user is logged in currently
   public static String getUsername();

   // returns null if no user is logged in currently
   public static String getUserSecurityRole();

4.9 The TableImpl Class

The websi.util.TableImpl class implements the Table interface in order to enable
the client programmer to construct and manage mutable Tables. The instances of the
TableImpl class, can be used in WebSI input and output methods in the same way as
Tables returned by the read method.

   A TableImpl has a fixed set of columns, specified at construction time:
   public TableImpl(String[] column_names);

   An alternative Constructor is used to create a single-column TableImpl and fill it with
data:
   public TableImpl(String column_name, Object[] values);

   The list of methods introduced by the TableImpl class follows.

   public void addRow(Object[] values);
      Adds a row to the TableImpl. If the supplied array is shorter than the number of
columns in the table, missing values of the new row will be nulls. If the supplied array
is longer than the number of columns in the table, the suffix of the array is ignored. If the
supplied array is null, it is considered a zero-length array.

   public void addRow(Table table, int row, String[] columns_to_add);
      Adds a row to the TableImpl. The row values are taken from the supplied Table, from
its row whose number is supplied in row, from its columns, whose names are passed in
columns_to_add array. If this array is null or empty, the values will be taken from all the
columns of the supplied Table. If the number of values to be added is less than the number
of columns in this TableImpl, nulls will be added to the last columns. The number of
values that will be added, will not exceed the number of columns in this TableImpl. The
order, in which column values are added to this TableImpl, is the same as the column
names order in columns_to_add array or, if this array is null or empty, the order is the
same as in the supplied Table.

   public void removeRow(int row);
      Removes the row, whose number is row. Rows are numbered starting from 1.

   public void removeAllRows();
      Removes all the rows of this TableImpl.

   public void setTableValue(int row, String column_name, Object value);
      Sets the value of this TableImpl in the (row, column_name) cell.

   public TableImpl project(String[] columns);
      Returns a projection of this TableImpl on columns, whose names are passed in the
columns array. If the columns array is null or empty, the new TableImpl will contain
all the columns of this TableImpl. The order of columns of the new TableImpl is the same as the order of columns names in the supplied columns array or, if this array is null or empty, the order is the same as the columns order in this TableImpl. The resulting TableImpl is independent of this TableImpl.

```java
public int[] findRowsByColumnValue(String columnName, Object value);
```

Returns the numbers of all rows of this TableImpl, containing a value, that equals() to value, in the column, whose name is columnName. Never returns null, but may return an empty array.

### 4.10 The Client Programmer Tools Class

The websi.util.ClientProgrammerTools class contains static helper methods for the client programmer:

```java
public static String buildValuesList(int[] arr);
public static String buildValuesList(double[] arr);
public static String buildValuesList(long[] arr);
public static String buildValuesList(Object[] arr);
public static String buildValuesList(Table table, String columnName, Class valuesClass);
```

These methods all build a string, that contains a list of values to be used in SQL WHERE clause, e.g., “(value1, value2, ... )”. Those lists are used, for example, with IN or NOT IN operators. The values in the resulting list are properly quoted, e.g., if the values are numbers, they are not quoted and if the values are TIMESTAMPs, they are quoted, like this: “(‘value1’, ‘value2’, ... )”. The last method builds a list out of values of the supplied Table, in the column whose name is columnName. In that case, the type of the column needs also be specified.
Chapter 5

Example

In this chapter we will bring an example information system, developed with the help of **WebSI**. The purposes of bringing the example are:

- To demonstrate our software construction methodology
- To demonstrate the usage of **WebSI**
- To make the reader familiar with the **WebSI**-based code
- To demonstrate the properties of a **WebSI**-based application, such as declarative code, ease of modifications and ease of verification

In our short example, we will not be able to demonstrate all the features of **WebSI**. But we will, of course, demonstrate the most important ones, such as use-case classes and interaction methods, manipulating the database, communicating with the user and security.

Our example is a small sales management system, called “Offers”. It stores information on the various products for sale, allowing product suppliers to add and modify their sale offers, and customers to browse the offers. We start with the use-case specification of the “Offers” system. Then we discuss some security considerations of the system. After that, we proceed to designing the database and coding the use cases. Then we demonstrate the UI's generated by **WebSI** for the “Offers” system. Finally, we will relate the Java statements of the implementation to the English statements of the use-case specification, in order to demonstrate direct verification.

5.1 Use-Case Specification

Figure 5.1 presents the use-case diagram of the “Offers” system. Three actors have been identified. **Suppliers** can add offers for existing products and manage their existing offers, i.e., update and remove the offers. **Customers** can browse the offers and register their interest in certain offers via the “Select Offers” use case. Customers can also state that they are already not interested in certain offers, via the “Remove Interests” use case. We can see that this use case is <<include>>d from the “Select Offers” use case, but it is also accessible directly, i.e., without first browsing for the offers. Finally, **system operators** can add new kinds of products to the system.

We now present the descriptions of the use cases.
The “Add Product” Use Case

**Goal** To enable the operator to add a new product to the system.

**Actors** Operator

**Description**

1. The operator selects the product’s category out of the available categories.
2. The operator specifies the product’s model.
3. Optionally, the operator specifies a description for the product.
4. The operator approves.
5. The system stores the details of the new product and presents a success message.
6. The system presents the details of the added product: category, model and description (if specified).

**Post-condition** The new product’s details are successfully stored by the system.

**Exception at step 5** A product of the same category and of the same model already exists. The system presents an error message.

The “Add Offer” Use Case

**Goal** To enable the supplier to add an offer for a product

**Actors** Supplier

**Description**

1. The system presents the set of products, for which the supplier has not given offers. For each product, its category, model and description are presented.
2. The supplier selects a single product to give an offer for.
3. The supplier specifies the price.
4. The supplier approves.
5. The system registers the new offer and presents a success message.
6. The system presents the details of the newly registered offer: product’s category, model and price.

Post-condition The new offer is successfully added.

Exception at step 1 The supplier has already given offers for all the products available. The system presents an appropriate message.

Exception at step 5 The supplier has specified a non-positive price. The system presents an error message.

The “Manage Offers” Use Case

Goal To enable the supplier to update the price of an offer or to remove the offer.

Actors Supplier

Description

1. The system presents all the offers the supplier has given. For each offer, the product’s category, product’s model and price are presented.
2. The supplier selects a single offer.
3. The supplier indicates to the system that she wishes to update the price of the selected offer.
4. The supplier specifies the new price for the offer.
5. The supplier approves.
6. The system updates the price of the offer and presents a success message.
7. The system presents the details of the updated offer: product’s category, model and price.

Post-condition An offer’s price has been updated or the offer has been removed.

Exception at step 1 The supplier has given no offers. The system presents an appropriate message.

Exception at step 6 The supplier has specified a non-positive price. The system presents an error message.

Variation at step 3

3. The supplier indicates to the system that she wishes to remove the selected offer.
4. The system removes the selected offer and presents a success message.

Exception at step 4 The selected offer cannot be removed, e.g., because there is a customer that registered an interest in it. The system presents an error message.
The “Select Offers” Use Case

Goal  To enable the customer to browse the offers of the different suppliers. To enable the customer to express interest in some of the offers

Actors  Customer

Description
1. The customer selects a city out of the set of cities that have suppliers.
2. The customer selects a product category out of the available categories.
3. The customer approves.
4. The system presents the suppliers in the chosen city, that have offers for products of the chosen category. For each supplier, its name and address are presented.
5. The customer selects a single supplier.
6. The customer approves.
7. The system presents the offers of the chosen supplier for the products of the chosen category. For each offer, the product’s model and price are presented.

Post-condition  The details of the selected offers are presented to the customer.

Exception at step 1  There are no cities that have suppliers or no product categories. The system presents an error message.

Exception at step 4  There are no suppliers in the chosen city that have offers for products of the chosen category. The system presents an error message.

Variation after step 7
8. Optionally, the customer selects one or more offers, to express her interest in them.
9. The customer approves.
10. The system registers the customer’s interest in the chosen offers.
11. The system <<include>>s the “Remove Interests” use case.

Variation at step 10  The customer has already expressed her interest in some of the selected offers. The system registers the customer’s interest in the remaining offers.

The “Remove Interests” Use Case

Goal  To enable the customer to view the offers she has expressed her interest in. To enable the customer to indicate that she is no longer interested in certain offers.

Actors  Customer

Description
1. The system presents all the offers, the customer has expressed her interest in. For each offer, the product category, model, description, price, supplier name, city and address are presented.

2. The customer selects one or more offers, in which she is no longer interested.

3. The customer approves.

4. The system updates the customer’s interests.

5. The use case starts from the beginning.

**Post-condition** The customer is no longer registered as interested in the selected offers.

**Exception at step 1** The customer has not expressed her interest in any offer. The system presents an appropriate message.

### 5.2 Security Considerations

Prior to designing the database schema, we should decide which security roles our system will have. Recall that in WebSI usernames and passwords are stored in the database.

The use-case specification has identified three actors for the “Offers” system: Customer, Supplier and Operator. Still, we should not immediately decide that these will also be the security roles. A security role is defined for a group of users, that:

- Need to access non-public services of the system, or
- Need to be identified by the system in order to perform their tasks

Consider the Supplier actor. Her tasks include adding, modifying and removing her offers. The “Offers” system must obviously know, to which supplier a certain offer belongs. Therefore, the Supplier actor needs to be identified by the system in order to perform her tasks, e.g., adding an offer. In addition, we would, of course, not like non-registered users to perform Supplier’s tasks. As a result, we define a separate security role for Suppliers.

The Operator actor accesses the “Add Item” use case, which obviously is a non-public service. Therefore, we have a separate security role for Operators.

The “Select Offers” use case may be considered as a public service. But the “Remove Interests” use case requires an identification of its actor—in order to manage her interest expression requests. Since the “Select Offers” <<include>>s the “Remove Interests” use case, it is best to define them both as non-public (otherwise, we would have to check manually in the code, whether it is legal to allow the inclusion of “Remove Interests” from “Select Offers”). We have a third security role, for Customers.

Here are the security settings for the “Offers” system:

- The “Add Product” use case can be executed by members of the “Operator” security role.
- The “Add Offer” and “Manage Offers” use cases can be executed by members of the “Supplier” security role.
- The “Select Offers” and “Remove Interests” use cases can be executed by members of the “Customer” security role.
• There are no public use cases.

In our case, we have a one-to-one mapping from actors to security roles. But, of course, this is not always the case.

## 5.3 Database Design

We now present the tables of the database for the “Offers” system.

We start with a category table that stores product categories. Its schema is presented in Tab. 5.1. The code column is the table’s primary key and it serves as a unique identifier.

### Table 5.1: The schema of the category table

<table>
<thead>
<tr>
<th>Column name</th>
<th>Datatype</th>
<th>Constraints</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>VARCHAR(50)</td>
<td>UNIQUE, NOT NULL</td>
<td></td>
</tr>
<tr>
<td>code</td>
<td>INTEGER</td>
<td>PRIMARY KEY</td>
<td>Generated by the database</td>
</tr>
</tbody>
</table>

for a category. Since we would like to avoid generating unique values manually, we employ a database mechanism for this. The database engine we use currently, supports defining a column as “AUTOINCREMENT” with an initial value and a step value. The programmer is not allowed to specify values for an “AUTOINCREMENT” column in her queries; instead the database engine generates the values, using the supplied initial value and adding to it a step value each time. Most current database engines support such value generation. As for the name column, we decide that category names should be unique. Here is the SQL statement that creates the category table (using our database engine SQL syntax):

```
CREATE TABLE category
(name VARCHAR(50) NOT NULL,
 code INTEGER DEFAULT AUTOINCREMENT INITIAL 1 INCREMENT 1 NOT NULL,
 CONSTRAINT category_pk PRIMARY KEY(code),
 CONSTRAINT category_name_unique UNIQUE(name));
```

Next we describe the product table, which stores the products available for sale. Its schema is presented in Tab. 5.2. The code column here serves the same purpose as the

### Table 5.2: The schema of the product table

<table>
<thead>
<tr>
<th>Column name</th>
<th>Datatype</th>
<th>Constraints</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>model</td>
<td>VARCHAR(50)</td>
<td>NOT NULL, UNIQUE (together with category.code)</td>
<td></td>
</tr>
<tr>
<td>category_code</td>
<td>INTEGER</td>
<td>NOT NULL, FOREIGN KEY to category(code), UNIQUE (together with model)</td>
<td></td>
</tr>
<tr>
<td>description</td>
<td>VARCHAR(200)</td>
<td>Default value is NULL</td>
<td></td>
</tr>
<tr>
<td>code</td>
<td>INTEGER</td>
<td>PRIMARY KEY</td>
<td>Generated by the database</td>
</tr>
</tbody>
</table>
The code column in the category table—it uniquely identifies the product. The model column stores the name of the model, and the category_code column identifies the category, to which the product belongs. The latter column is defined as a FOREIGN KEY to the code column of the category table. This means that the values of the category column must at any time be a subset of values stored in the code column of the category table.

In particular, the database engine will not allow removing a category, of which products exist, or adding a product of an illegal category. In addition, the model and the category_code columns are together constrained with a UNIQUE constraint. This means that a certain model-category combination can appear at most once in the product table. Effectively, this states that there cannot be two products, that have the same model and the same category, but a different product code. Here is the SQL statement that creates the product table:

```sql
CREATE TABLE product
(model VARCHAR(50) NOT NULL,
category_code INTEGER NOT NULL,
description VARCHAR(200) DEFAULT NULL,
code INTEGER DEFAULT AUTOINCREMENT INITIAL 1 INCREMENT 1 NOT NULL,
CONSTRAINT product_pk PRIMARY KEY(code),
CONSTRAINT model_categ_unq UNIQUE(model,category_code),
CONSTRAINT categ_fk FOREIGN KEY (category_code) REFERENCES category(code));
```

Before we can present the table that stores the details of the sales offers, we need to present the table of the suppliers. This is because each offer is related to a particular supplier. The schema of the supplier table is presented in Tab. 5.3. The id column uniquely identifies a supplier. It is also used as the supplier’s username. The name-city-address combination is unique across the supplier table. This means that there cannot be two suppliers that have the same name, are located in the same city, at the same address, but have a different ID. Ideally, the names of the cities should be stored in a separate table, and the supplier table should store their unique codes, referencing the cities table with a FOREIGN KEY constraint. In that case, potential spelling errors in the names of the cities would be avoided. For the sake of simplicity we ignore this issue. As for the password column, it would be desirable to have its values encrypted by the database-specific encryption mechanism, but for the simplicity of the example we store them as-is. Here is this table’s creation query:

```sql
CREATE TABLE supplier
(name VARCHAR(50) NOT NULL,
city VARCHAR(50) NOT NULL,
address VARCHAR(50) NOT NULL,
password VARCHAR(50) NOT NULL,
id INTEGER DEFAULT AUTOINCREMENT INITIAL 1 INCREMENT 1,
CONSTRAINT supplier_pk PRIMARY KEY (id),
CONSTRAINT name_city_addr_unq UNIQUE (name, city, address));
```

Table 5.4 illustrates the schema of the offer table. The id column uniquely identifies an offer. The product_code and the supplier_id columns identify the product, for
Table 5.4: The schema of the *offer* table

<table>
<thead>
<tr>
<th>Column name</th>
<th>Datatype</th>
<th>Constraints</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>product_code</td>
<td>INTEGER</td>
<td>NOT NULL, FOREIGN KEY to product(code), UNIQUE (together with supplier_id)</td>
<td></td>
</tr>
<tr>
<td>supplier_id</td>
<td>INTEGER</td>
<td>NOT NULL, FOREIGN KEY to supplier(id), UNIQUE (together with product_code)</td>
<td></td>
</tr>
<tr>
<td>price</td>
<td>DECIMAL(8,2)</td>
<td>NOT NULL, CHECK(price &gt; 0)</td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>INTEGER</td>
<td>PRIMARY KEY</td>
<td>Generated by the database</td>
</tr>
</tbody>
</table>

which the offer was given, and the supplier that gave the offer, respectively. They are both constrained by FOREIGN KEY constraints that reference appropriate columns in appropriate tables. In addition, product-supplier combination is constrained as UNIQUE. This means that a certain supplier can give at most one offer for a certain product. Finally, the price column is CHECKed to be positive. The creation query follows:

```
CREATE TABLE offer
(product_code INTEGER NOT NULL,
supplier_id INTEGER NOT NULL,
price DECIMAL(8,2) NOT NULL,
id INTEGER DEFAULT AUTOINCREMENT INITIAL 1 INCREMENT 1 NOT NULL,
CONSTRAINT offer_pk PRIMARY KEY (id),
CONSTRAINT prod_suppl_unq UNIQUE (product_code, supplier_id),
CONSTRAINT product_fk FOREIGN KEY (product_code) REFERENCES product(code),
CONSTRAINT supplier_fk FOREIGN KEY (supplier_id) REFERENCES supplier(id),
CONSTRAINT positive_price CHECK(price > 0));
```

The next table stores the details of the customers, its schema is demonstrated in Tab. 5.5. The *username* column uniquely identifies a customer and serves as her username. The name-city combination is defined as UNIQUE to state that there exists at most one customer with a certain name from a certain city. (The *city* column is not used in this example.) The table creation query is:

```
CREATE TABLE customer
(username VARCHAR(50) NOT NULL,
name VARCHAR(50) NOT NULL,
city VARCHAR(50) NOT NULL,  
password VARCHAR(50) NOT NULL,)
```
The `interested` table stores customers’ interests in various offers. Its schema is presented in Tab. 5.6. The primary key of this table contains its both columns. In particular,

<table>
<thead>
<tr>
<th>Column name</th>
<th>Datatype</th>
<th>Constraints</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>cust_username</td>
<td>VARCHAR(50)</td>
<td>PRIMARY KEY (together with offer_id), FOREIGN KEY to customer(username)</td>
<td></td>
</tr>
<tr>
<td>offer_id</td>
<td>INTEGER</td>
<td>PRIMARY KEY (together with cust_username), FOREIGN KEY to offer(id)</td>
<td></td>
</tr>
</tbody>
</table>

this means that a certain customer can register her interest in a certain offer at most once. In addition, both columns reference appropriate tables using the FOREIGN KEY constraints. This ensures that these columns contain only legal usernames and offer ID’s. Another consequence of these constraints is that an offer cannot be removed, if there exist customers that are interested in it.

The last table of our small database is called `operator` and it stores identification details for system operators. Its schema is depicted in Tab. 5.7. This table is currently used only for operators identification purposes.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Datatype</th>
<th>Constraints</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
<td>VARCHAR(50)</td>
<td>PRIMARY KEY</td>
<td></td>
</tr>
<tr>
<td>password</td>
<td>VARCHAR(50)</td>
<td>NOT NULL</td>
<td></td>
</tr>
</tbody>
</table>

### 5.4 Coding and Verifying the “Add Product” Use Case

Let us start with a relatively simple use case “Add Product”. First, we must define an appropriate use-case class:

```java
import websi.*;
import websi.exceptions.*;
import websi.util.*;

public class AddProduct extends UseCase {
    ...
}
```

The three imports always should be performed. They bring in the `WebSI` packages that are relevant to the client programmer. Now we must code the `start` interaction method of the use-case class, according to the first IO sequence of the use case, which is:
1. The operator selects the product’s category out of the available categories.
2. The operator specifies the product’s model.
3. Optionally, the operator specifies a description for the product.
4. The operator approves.

Here is the code:

```java
public void start() throws Exception {
    Table avail_categories =
        DB.read("SELECT code, name FROM category");
    String[] categories_vis_cols = {"name");
    Select.one("category", avail_categories,
                categories_vis_cols, "code", null,
                "Select a category to add product for:");
    UserComposed.any("model", null, "Specify the product’s model:");
    UserComposed.anyLongOptional("descr", null,
                                "You may add a description for the new product:");
    Action.action("addProduct", "Add the new product");
}
```

Let us briefly explain the code. First we obtain all the available categories; for each category we have its name and its code. (The user will see and select category names, while we will be manipulating codes.) Then we enable the user to select a single category, out of the categories Table. The visible column is category name, this is stated in the third parameter to the Select.one method. When the user selects a category and approves, we will (in the next interaction method) fetch this category’s code value. This is stated in the fourth parameter to the Select.one method, i.e., that return values should be taken from the code column. Default values to all methods are left unspecified (null). When the user approves, the addProduct interaction method will be invoked.

The code presented above looks very natural and correct, however, it has a hidden pitfall. The Select.one method requests the user to select a single row out of the supplied Table. The request is mandatory. But what if the supplied Table is empty? That is, if it contains zero rows? In that case, the Select.one method will throw an exception. From our observations, a programmer that has a modest experience with WebSI would have noticed this problem right away. Being familiar with the Select.one method, the programmer knows that, in any case, prior to calling it, she must ensure that the Table is non-empty. But this situation is, in fact, related to application logic (AL), and, therefore, should be described in the use-case specification:

**Exception at step 1** There are no product categories defined. The system presents an appropriate message.

It seems, that in some cases, by only being familiar with WebSI methods, the programmer can discover defects in the use-case specification and correct them. This property is interesting and we will be seeing more of it later. By now, let us write the correct version of the start method:

```java
public void start() throws Exception {
    Table avail_categories =
        DB.read("SELECT code, name FROM category");
    if (avail_categories.getRowsNr() == 0) {
        Output.scalar(
            "Technion - Computer Science Department - M.Sc. Thesis MSC-2006-09 - 2006"
        )
        "Exception at step 1 There are no product categories defined. The system presents an appropriate message.

It seems, that in some cases, by only being familiar with WebSI methods, the programmer can discover defects in the use-case specification and correct them. This property is interesting and we will be seeing more of it later. By now, let us write the correct version of the start method:

```java
public void start() throws Exception {
    Table avail_categories =
        DB.read("SELECT code, name FROM category");
    if (avail_categories.getRowsNr() == 0) {
    Output.scalar("Exception at step 1 There are no product categories defined. The system presents an appropriate message.
```
"No product categories are defined in the system.",
null);
return;
}

String[] categories_vis_cols = {"name");
Select.one("category", avail_categories,
categories_vis_cols, "code", null,
"Select a category to add product for:"
);

UserComposed.any("model", null, "Specify the product’s model:"
);
UserComposed.anyLongOptional("descr", null,
"You may add a description for the new product:"
);

Action.action("addProduct", "Add the new product")
}

Now let us code the next IO sequence of the use case:

5. The system stores the details of the new product and presents a success message.

6. The system presents the details of the added product: category, model and description (if specified).

Let us first write the code that updates the database and presents a success message:

```java
public void addProduct() throws Exception {
    int category_code = Fetch.Int("category");
    String model = Fetch.string("model");
    String descr = null;
    // Description input request is optional
    if (Fetch.inputSpecified("descr"))
        descr = Fetch.string("descr");

    DB.write("INSERT INTO product(model, category_code, description) VALUES" +
             "('" + model + "/", + category_code + "," +
             "(" + descr == null ? "NULL": ("" + descr + ")) +
             ");
    Output.scalar("The new product successfully added.", null);
}
```

This code is very typical to an interaction method: first it fetches the input values supplied by the user, then it updates the database, and, finally, presents the output to the user. The output, however, is not complete. We must, in addition, present the product category, model and description. The model and description are available, but the category name is not, we have the category code only. Let us obtain the category name and present the required outputs:

```java
public void addProduct() throws Exception {
    int category_code = Fetch.Int("category");
    String category_name = null;
    String model = Fetch.string("model");
    String descr = null;
    // Description input request is optional
    if (Fetch.inputSpecified("descr"))
        descr = Fetch.string("descr");

    DB.write("INSERT INTO product(model, category_code, description) VALUES" +
             "('" + model + "/", + category_code + "," +
```
We believe, the reader now can see a potential hidden pitfall in obtaining the category name. What if the read method returned an empty Table? This means, that the category does not exist! But how could this happen? The category existed when we requested the user to select a single category. The user could not supply an illegal code, because the user was requested to select a category name out of a predefined set. So how could this happen?

Here comes the multi-user nature of our system. Remember that there may be other users operating the system and between two separate database transactions we cannot assume that the database does not change. The category might be removed by some other user of the database. (Although the current version of the “Offers” system does not allow adding and removing categories, we cannot, of course, rely on this. Future extensions may add additional functionalities.)

Fortunately, in that case the read method cannot return an empty Table. Remember, that we have a FOREIGN KEY constraint on the category_code column of the product table. So if the category has been removed, the write method would have failed (with a ForeignKeyViolationException) and the read method would not be invoked.

Indeed, we have not considered the various constraint violation exceptions that can be thrown by the write method. Here we return again to the discussed WebSI property: a programmer familiar with WebSI knows, that any update to the database must be checked for database constraint violation exceptions.

Which database constraint violation exceptions can be thrown in our case? The product table has three constraints defined. The PRIMARY KEY constraint, defined on the code column, cannot be violated, because the values of this column are generated by the database and guaranteed to be unique. The UNIQUE constraint defined on (model, category_code) columns, if violated, means that the product with the same model of the same category already exists. We have foreseen this case in our use-case specification. The FOREIGN KEY constraint on the category_code column, if violated, means that the category has been removed. We have not foreseen this problem in our use-case specification. We see again, that just by being familiar with WebSI, we can find defects in a use-case specification and correct them.

Here is the complete and correct code of the addProduct method:

```java
public void addProduct() throws Exception {
    int category_code = Fetch.Int("category");
    String category_name = null;
    String model = Fetch.string("model");
    String descr = null;
    // Description input request is optional
    if (Fetch.inputSpecified("descr"))
        descr = Fetch.string("descr");

    try {
        ((descr == null) ? "NULL": ("'" + descr + "'")) + "");

        category_name =
            DB.read("SELECT name FROM category WHERE code=") +
            category_code).getString(1,1);
        Output.scalar("The new product successfully added.", null);
        Output.scalar(category_name, "Product category:"");
        Output.scalar(model, "Product model:"");
        if (descr != null)
            Output.scalar(descr, "Product description:");
    }
}
```
DB.write(
    "INSERT INTO product (model, category_code, description) VALUES" +
    "(" + model + "," + category_code + "," +
    ((descr == null) ? "NULL" : ("" + descr + ")") +
    ")");

    // Cannot fail if write succeeded
    category_name =
        DB.read("SELECT name FROM category WHERE code=" +
                 category_code).getString(1,1);

    Output.scalar("The new product successfully added.", null);
    Output.scalar(category_name, "Product category:`);
    Output.scalar(model, "Product model:`);
    if (descr != null)
        Output.scalar(descr, "Product description:`);
    } catch (PrimaryKeyOrUniqueViolationException exc) {
        Output.scalar("The product of model " +
                     model + " already exists!",
                     "Failed adding product:`);
    } catch (ForeignKeyViolationException exc) {
        Output.scalar("The specified category no longer exists!",
                     "Unexpected error:`);
    }

Note, that in case of a constraint violation exception, we do not have to manually roll
back the transaction. This is because the transaction has a single modifying query, the one
issued by the write method. This query is rolled back by the database in case a constraint
is violated. If there were more than a single update query, we might have to manually roll
back the ongoing transaction (using DB.rollbackTransaction).

We have seen in this example that using database constraints helped us to implement
the use-case specification, and, in general, to keep our code correct. In principle, we could
have manually checked, whether a product of the same model in the same category exists,
but in many cases database constraints make it simpler and safer, in our opinion.

Figure 5.2 demonstrates the GUI generated by WebSI for the “Add Product” use
case, during the various stages of its execution. Here, the default WebSI IS has been used.
Figure 5.2: Executing the “Add Product” use case
Let us now demonstrate how a direct verification of a WebSI-based code works. We will look again at the specification of the “Add Product” use case, and we will map the various parts of the specification to the code. This will give us an idea of how close a WebSI-based code is to the specification.

Table 5.8 relates the natural language statements of the specification to the Java code statements.

Table 5.8: Verifying the implementation of the “Add Product” use case against its specification

<table>
<thead>
<tr>
<th>Specification</th>
<th>Related code statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. The operator selects the product’s category out of the available categories.</td>
<td>Table avail_categories = DB.read(&quot;SELECT code, name FROM category&quot;); String[] categories_vis_cols = {&quot;name&quot;}; Select.one(&quot;category&quot;, avail_categories, categories_vis_cols, &quot;code&quot;, null, &quot;Select a category to add product for:&quot;);</td>
</tr>
<tr>
<td>2. The operator specifies the product’s model.</td>
<td>UserComposed.any(&quot;model&quot;, null, &quot;Specify the product’s model:&quot;);</td>
</tr>
<tr>
<td>3. Optionally, the operator specifies a description for the product.</td>
<td>UserComposed.anyLongOptional(&quot;descr&quot;, null, &quot;You may add ...&quot;);</td>
</tr>
<tr>
<td>4. The operator approves.</td>
<td>Action.action(&quot;addProduct&quot;, &quot;Add the new product&quot;);</td>
</tr>
<tr>
<td>5. The system stores the details of the new product and presents a success message.</td>
<td>DB.write(&quot;INSERT INTO product ...&quot;); Output.scalar(&quot;... successfully added.&quot;, null);</td>
</tr>
<tr>
<td>6. The system presents the details of the added product: category, model and description (if specified).</td>
<td>Output.scalar(category_name, &quot;Product category:&quot;); Output.scalar(model, &quot;Product model:&quot;); if (descr != null) Output.scalar(descr, &quot;Product description:&quot;);</td>
</tr>
<tr>
<td>Exception at step 5: A product of the same category, of the same model already exists. The system presents an error message.</td>
<td>catch (PrimaryKeyOrUniqueViolationException exc) { Output.scalar(&quot;The product of model &quot; + model + “ already exists!&quot;, &quot;Failed adding product:&quot;); }</td>
</tr>
</tbody>
</table>

It can be noted, that we can successfully relate all the use-case specification statements to Java statements. On the other hand, some Java statements remained not related to any statement of a use-case specification. Some of these statements, however, clearly deal with AL, that should be specified in the use case. Those statements and the missing specifications are presented in Tab. 5.9.

The rest of the Java statements are those that fetch the input values supplied by the user:

```java
int category_code = Fetch.Int("category");
String model = Fetch.string("model");
String descr = null;
// Description input request is optional
if (Fetch.inputSpecified("descr"))
    descr = Fetch.string("descr");
```

and the statement that obtains the name of the selected category:

```java
String category_name = null;
...
    category_name = DB.read("SELECT name FROM category WHERE code=" +
```
Table 5.9: Missing specifications and the Java statements related to them in the “Add Product” use case

<table>
<thead>
<tr>
<th>Missing Specification</th>
<th>Related code statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exception at step 1:</td>
<td>if (availableCategories.getRowCount() == 0) {</td>
</tr>
<tr>
<td>There are no product</td>
<td>Output.scalar(</td>
</tr>
<tr>
<td>categories defined.</td>
<td>“No product categories...” ,</td>
</tr>
<tr>
<td>The system presents</td>
<td>null);</td>
</tr>
<tr>
<td>an appropriate message.</td>
<td></td>
</tr>
<tr>
<td>Exception at step 5:</td>
<td>catch (primaryKeyException exc) {</td>
</tr>
<tr>
<td>The selected category</td>
<td>Output.scalar(</td>
</tr>
<tr>
<td>does not exist anymore.</td>
<td></td>
</tr>
<tr>
<td>The system presents</td>
<td>“The specified category no longer exists!” ,</td>
</tr>
<tr>
<td>an appropriate message.</td>
<td></td>
</tr>
</tbody>
</table>

Obviously, statements that fetch the input values supplied by the user will appear in almost any interaction method. They should be verified against the appropriate input requests.

We can see that in this case most of the Java statements are clearly mapped to the natural language statements in the use-case specification. Out of total 24 statements (counting also the if conditions) 13 are related to the existing use-case specification statements, and 4 more are related to the use-case specification statements that are missing but should exist. Out of the remaining 7 statements, 5 fetch the input values supplied by the user. We can also conclude that in this example, specifications (including the missing ones) are implemented with few Java statements. One can, of course, rearrange in many ways the natural language specification statements and also the Java statements. We believe, however, that this example still gives a general idea on the closeness of the WebST-based code to the use-case specification.

5.5 Coding and Verifying the “Select Offers” Use Case

The first IO sequence of this use case is:

1. The customer selects a city out of the set of cities that have suppliers.
2. The customer selects a product category out of the available categories.
3. The customer approves.

We define the appropriate use-case class and code its first interaction method start:

```java
import websi.*;
import websi.exceptions.*;
import websi.util.*;

public class SelectOffers extends UseCase {
    public void start() throws Exception {
        Table cities = DB.read("SELECT DISTINCT city FROM supplier");
        Table categories = DB.read("SELECT code, name FROM category");
    }
```
We have foreseen this exception

```java
if (cities.getRowsNr() == 0 || categories.getRowsNr() == 0) {
    Output.scalar(
        "There are no suppliers or no product categories " +
        "in the system.", null);
    return;
}
```

Select.one("city", cities, null, null, null,
         "Select the desired city:");

String[] categ_visible_cols = {"name"};
Select.one("category_code", categories, categ_visible_cols,
         "code", null, "Select a product category:");

Action.action("seeSuppliers", "See suppliers");

As in the previous use case, prior to calling Select.one we check that the appropriate Tables are non-empty. This exceptional situation has been foreseen in the use-case specification:

**Exception at step 1** There are no cities that have suppliers or no product categories. The system presents an error message.

Let us continue to the next IO sequence:

4. The system presents the suppliers in the chosen city, that have offers for products of the chosen category. For each supplier, its name and address are presented.

5. The customer selects a single supplier.

6. The customer approves.

This IO sequence is coded in the `seeSuppliers` interaction method:

```java
public void seeSuppliers() throws Exception {
    _city = Fetch.string("city");
    _category_code = Fetch.Int("category_code");

    Table suppliers =
        DB.read(
            "SELECT DISTINCT S.id, S.name, S.address, C.name AS categ_name" +
            " FROM supplier S, product P, offer O, category C" +
            " WHERE S.city=\"" + _city + \\
            "\"" +
            " AND O.supplier_id=S.id " +
            " AND P.category_code=\"" + _category_code +
            " AND O.product_code=P.code AND P.category_code=C.code";

    if (suppliers.getRowsNr() == 0) {
        Output.scalar("No suppliers found!", null);
        return;
    }

    String[] visible_cols = {"name", "address"};
    Select.one("supplier_id", suppliers, visible_cols, "id", null,
              "Suppliers for product category " +
              suppliers.getString(1, "categ_name") +
```
First of all we fetch the input values supplied by the user, i.e., the city name and the product category code. We save those values in the data members of the SelectOffers class, rather than in local variables, because these values will be needed also in the next IO sequence. Here is the data members declaration:

```java
private int _category_code = -1;
private String _city = null;
```

Next we perform an SQL SELECT query, which searches for suppliers in the selected city, that have offers for products of the selected category. Note that this query also fetches the category names of the offered products. All these names are identical, since we are searching for a specific product category. We do not really need the category name, but we want to print it in a caption, rather than the category code, with which the customer is not familiar.

Then we check for an exceptional situation, which has been foreseen in the use-case description:

**Exception at step 4** The are no suppliers in the chosen city, that have offers for products of the chosen category. The system presents an error message.

After we have taken care about this exception, we enable the user to select a single supplier and to approve. Note how we print the category name in the caption: we fetch the name from the first row of the suppliers Table. We know that this table is non-empty (we have just checked it), but it may contain only one row, so we use the first row.

The next IO sequence is:

7. The system presents the offers of the chosen supplier for the products of the chosen category. For each offer, the product’s model and price are presented.

To code the use case correctly, we must also consider the variation, which may occur at this step:

**Variation after step 7**

8. Optionally, the customer selects one or more offers, to express her interest in them.

9. The customer approves.

Here is the code:

```java
public void seeOffers() throws Exception {
    String supplier_id = Fetch.string("supplier_id");

    Table offers_of_selected_supplier =
        DB.read("SELECT O.id, C.name AS category, P.model, O.price," +
                " S.name AS supplier, S.address " +
                " FROM offer O, product P, category C, supplier S" +
                " WHERE O.supplier_id=" + supplier_id +
                " AND O.product_code=P.code AND P.category_code=" +
                _category_code +
                " AND P.category_code=C.category AND S.id=O.supplier_id");
    if (offers_of_selected_supplier.getRowsNr() == 0) {
        Output.scalar("No offers of the selected supplier" +
```
First we fetch the ID of the selected supplier.

Then we issue an SQL query, which obtains the offers of the selected supplier for the selected product category. Note that, as before, we fetch the category names, which are all identical. We also fetch the name and the address of the supplier, which are also identical in all the rows, because we consider a specific supplier here. All these details will be used in a caption.

Now here comes an interesting question. Is it possible, that there are no offers of the selected supplier for the selected product category? After all, in the previous IO sequence we have shown only the suppliers that do have offers for the selected product category. Yes, it is indeed possible. Our system is a multi-user one, and the supplier could have meanwhile removed all her offers. Alternatively, the supplier could be removed by a system operator (in that case, all her offers are removed also, because of the FOREIGN KEY constraint). Or, the product category could have been removed (in which case, all the products of this category and all the offers for these products do not exist also, because of FOREIGN KEY constraints).

So we have here an additional exceptional situation, which has not been foreseen in the use-case description.

Finally, we enable the user to optionally select one or more offers, in which she is interested. Pay attention to the caption of this input request; we use here the category name and the supplier’s name and address.

We still have to code the rest of the variation:

10. The system registers the customer’s interest in the chosen offers.

11. The system <<include>>s the “Remove Interests” use case.

Here is the code:

```java
public void expressInterest() throws Exception {
    // We must check, since the input request was optional
    if (!Fetch.inputSpecified("offer_ids")) {
        Output.scalar(
            "You haven’t selected any offers to express interest in.",
            null);
        return;
    }

    String[] offers_vis_cols = {"model", "price"};
    // Here we also consider the variation
    Select.oneOrMoreOptional("offer_ids", offers_of_selected_supplier,
        offers_vis_cols, "id", null,
        "Offers of " +
        offers_of_selected_supplier.getString(1,"suppl_name") +
        ", located at " +
        offers_of_selected_supplier.getString(1, "address") +
        ", in " + _city +
        " for products of category " +
        offers_of_selected_supplier.getString(1, "category") + ":");

    Action.action("expressInterest",
        "Express interest in the selected offers");
}
```
int[] offer_ids = Fetch.Ints("offer_ids");

boolean all_offer_ids_ok = true;
for (int idx = 0; idx < offer_ids.length; ++idx) {
    try {
        DB.write("INSERT INTO interested VALUES (" +
                "" + UserID.getUsername() + "," +
                offer_ids[idx] + ");
    }
    catch (ForeignKeyViolationException exc) {
        // There are two FOREIGN KEY constraints defined
        if ("cust_usern_fk".equals(exc.getViolatedConstraintName())) {
            // If happens, this happens on the first query, which is
            // rolled back when the exception occurs
            // So we can just exit
            Output.scalar("You are not an authorized user anymore.",
                          "Unexpected error: ");
            return;
        }
        // Otherwise, the offer id does not exist...
        // We just proceed to the next offer
        all_offer_ids_ok = false;
    }
    catch (PrimaryKeyOrUniqueViolationException exc) {
        // The interest for this offer has already been expressed
        // We just proceed...
    }
}

// end of for

if (!all_offer_ids_ok)
    Output.scalar("Some of the offers, you have expressed interest in, do not exist anymore", null);

includeUseCase(RemoveInterests.class, null, null);

First we obtain the ID’s of the offers, in which the user is interested. Since the input
request for these offers was optional, we must check whether the request was ignored by
the user or not.

Then we just have to insert the selected ID’s into the interested table, together with
the ID of the supplier. This is done in the body of the for loop.

Each INSERT in this loop might violate some database constraints. First of all, there
are two FOREIGN KEY constraints defined for the interested table. The constraint
named cust_usern_fk ensures that only legal customers express their interest. If the
current customer is not a legal one anymore, this constraint will be violated on the first
INSERT. This exceptional situation has not been foreseen in the use-case description.

The second FOREIGN KEY constraint ensures that the customer is interested only
in legal offers. Some of the offers selected by the customer might not be legal anymore.
We use a boolean flag to check this and issue an appropriate warning. Our use-case
specification has not foreseen this either.

Finally, there is a PRIMARY KEY constraint, which ensures that a customer expresses
her interest in a certain offer at most once. In case this constraint is violated, we just move
to the next offer.

At the end of the method we <<include>> the “Remove Interests” use case, as re-
quired.
Figure 5.3 presents some of the GUI’s generated by WebSI for the “Select Offers” use case.

![GUI for Select Offers use case]

Let us now, as in the previous use case, relate the English use-case specification statements to the Java statements. This is done in Tab. 5.10 and 5.11.
Table 5.10: Verifying the implementation of the “Select Offers” use case against its specification

<table>
<thead>
<tr>
<th>Specification</th>
<th>Related code statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. The customer selects a city out of the set of cities that have suppliers.</td>
<td>Table cities = DB.read(&quot;SELECT DISTINCT city FROM supplier&quot;); Select.one(&quot;city&quot;, cities, null, null, null, &quot;Select the desired city&quot;);</td>
</tr>
<tr>
<td>2. The customer selects a product category out of the available categories.</td>
<td>Table categories = DB.read(&quot;SELECT code, name FROM category&quot;); String[] categ_visible_cols = {&quot;name&quot;}; Select.one(&quot;category_code&quot;, categories, categ_visible_cols, &quot;code&quot;, null, &quot;Select a product category&quot;);</td>
</tr>
<tr>
<td>3. The customer approves.</td>
<td>Action.action(&quot;seeSuppliers&quot;, &quot;See suppliers&quot;);</td>
</tr>
<tr>
<td>4. The system presents the suppliers in the chosen city, that have offers for products of the chosen category. For each supplier, its name and address are presented.</td>
<td>Table suppliers = DB.read(&quot;SELECT DISTINCT S.id, S.name, S.address...&quot;); String[] visible_cols = {&quot;name&quot;, &quot;address&quot;}; Select.one(&quot;supplier_id&quot;, suppliers, visible_cols, &quot;id&quot;, null, &quot;Suppliers for product category &quot; + suppliers.getString(1, &quot;categ_name&quot;) + &quot;, located at &quot; + suppliers.getString(1, &quot;address&quot;) + &quot;, in &quot; + city + &quot;:&quot;);</td>
</tr>
<tr>
<td>5. The customer selects a single supplier.</td>
<td>for (int idx = 0; idx &lt; offer_ids.length; ++idx) { DB.write(&quot;INSERT INTO interested ...&quot;); ... }</td>
</tr>
<tr>
<td>6. The customer approves.</td>
<td>Action.action(&quot;seeOffers&quot;, &quot;See offers of the selected supplier&quot;);</td>
</tr>
<tr>
<td>7. The system presents the offers of the chosen supplier for the products of the chosen category. For each offer, the product’s model and price are presented. Variation at step 7: 8. Optionally, the customer selects one or more offers, to express her interest in them.</td>
<td>Table offers_of_selected_supplier = DB.read(&quot;SELECT O.id, ..., P.model, O.price...&quot;); String[] offers_vis_cols = {&quot;model&quot;, &quot;price&quot;}; Select.oneOrMoreOptional(&quot;offer_ids&quot;, offers_of_selected_supplier, offers_vis_cols, &quot;id&quot;, null, &quot;Offers of &quot; + offers_of_selected_supplier.getString(1, &quot;suppl_name&quot;) + &quot;, located at &quot; + offers_of_selected_supplier.getString(1, &quot;address&quot;) + &quot;, in &quot; + city + &quot;, for products of category &quot; + offers_of_selected_supplier.getString(1, &quot;category&quot;) + &quot;:&quot;);</td>
</tr>
<tr>
<td>Continuing the variation at 7: 9. The customer approves.</td>
<td>includeUseCase(RemoveInterests.class, null, null);</td>
</tr>
<tr>
<td>Continuing the variation at 7: 10. The system registers the customer’s interest in the chosen offers.</td>
<td></td>
</tr>
<tr>
<td>Continuing the variation at 7: The system &lt;&lt;include&gt;&gt;$ the “Remove Interests” use case.</td>
<td></td>
</tr>
</tbody>
</table>
Table 5.11: Verifying the implementation of the “Select Offers” use case against its specification

<table>
<thead>
<tr>
<th>Specification</th>
<th>Related code statements</th>
</tr>
</thead>
</table>
| Variation at step 10: 10. The customer has already expressed her interest in some of the selected offers. The system registers the customer’s interest in the remaining offers. | `catch (PrimaryKeyOrUniqueViolationException exc) {
   // The interest for this offer
   // has already been expressed
   // We just proceed
   }`                                                                                     |
| Exception at step 1: There are no cities that have suppliers or no product categories. The system presents an error message. | `if (cities.getRowsNr()==0 || categories.getRowsNr()==0){
   Output.scalar("There are no suppliers or " + " no product categories " + " in the system.", null);
   return;
 }`                                                                                     |
| Exception at step 4: The are no suppliers in the chosen city that have offers for products of the chosen category The system presents an error message. | `if (suppliers.getRowsNr()==0){
   Output.scalar("No suppliers found!", null);
   return;
 }`                                                                                     |
As in the previous use case, there are some Java statements that clearly deal with AL, but situations they cover are not specified in the use case. These are all exceptional situations, that stem from the multi-user nature of our system. Whether these situations could be foreseen in the use-case specification is an open question. Table 5.12 lists the missing specification statements and the Java statements related to them. The rest of the

<table>
<thead>
<tr>
<th>Missing Specification</th>
<th>Related code statements</th>
</tr>
</thead>
</table>
| Exception at step 7: No offers of the selected supplier for products of the selected category found. The system presents an error message. | if (offers_of_selected_supplier.getRowsNr() == 0) {                                                     
  Output.scalar("No offers of the selected supplier" + " for the selected product category found!", "Unexpected error");  
  return;                                                                                                                                 |
| Exception at step 10: The supplier, which operates the system is not an authorized supplier anymore. The system presents an error message. | catch (ForeignKeyViolationException exc) {                                                                   
  if ("cust_user_fk".equals(exc.getViolatedConstraintName())) {                                          
    Output.scalar("You are not an authorized...", "Unexpected error");                                 
    return;                                                                                                                                 |
| Exception at step 10: Some of the selected offers do not exist anymore. The system registers the customer’s interest in the remaining offers and issues a warning. | boolean all_offer_ids_ok = true;                                                                                                                              
  catch (ForeignKeyViolationException exc) {                                                                     
    all_offer_ids_ok = false;                                                                                   
    if (!all_offer_ids_ok) {                                                                                    
      Output.scalar("Some of the offers," + " you have expressed interest in," + " do not exist anymore", null);    
    }                                                                                                                                 |

Java statements fetch input values supplied by the user:

```java
    _city = Fetch.string("city");
    _category_code = Fetch.Int("category_code");
    ...
    String supplier_id = Fetch.string("supplier_id");
    ...
    if (!Fetch.inputSpecified("offer_ids")) {                                                                  
      Output.scalar(                                             
        "You haven’t selected any offers to express interest in.", null);                                         
      return;                                                                                                    
    }
    int[] offer_ids = Fetch.Ints("offer_ids");
```

In this use case, out of total 44 Java statements, 24 are related to the existing use-case specification statements, and 11 more are related to exceptional situations unspecified in the use case. Out of the rest 9 statements, 7 are related to fetching input values and the last 2 statements declare data members. We can see that in this case we have also been able to trace most of the Java statements back to AL specifications (whether existing on the original use-case description or not).

### 5.6 The Rest of the Code

In this section we bring the rest of the “Offers” system implementation.
The “Remove Interests” use-case:

```java
import websi.*;
import websi.exceptions.*;
import websi.util.*;

public class RemoveInterests extends UseCase {
    public void start() throws Exception {
        // All the interests of the current supplier
        Table expressed_interest = DB.read("SELECT O.id, C.name AS category, P.model,
            "P.description, O.price, S.name AS supplier,
            "S.city, S.address"
            + "FROM interested I, offer O, product P,"
            + "supplier S, category C"
            + "WHERE I.cust_username='" + UserID.getUsername() + \
            '"' + "AND O.id=I.offer_id AND O.supplier_id=S.id " + \
            "AND O.product_code=P.code AND P.category_code=C.code");
        // A foreseen exception
        if (expressed_interest.getRowsNr() == 0) {
            Output.scalar("You haven’t expressed interest in any offer.",
                null);
            return;
        }
        // Visible columns - according to use-case spec.
        String[] expressed_vis_cols = {"category", "model",
            "description", "price", "supplier", "city", "address"};
        Select.oneOrMore("offer_ids", expressed_interest,
            expressed_vis_cols, "id", null,
            "The offers you have expressed interest in:");
        Action.action("unExpressInterest",
            "I am no longer interested in the selected offers");
    }
    public void unExpressInterest() throws Exception {
        int[] offer_ids_to_remove_interest = Fetch.Ints("offer_ids");
        DB.write("DELETE FROM interested WHERE offer_id IN ") + \
            " // We can be sure the array is non-empty!
            // Because the input request was mandatory
            ClientProgrammerTools.buildValuesList(
                offer_ids_to_remove_interest);
        start(); // Start the use-case from the beginning
    }
}
```

The “Add Offer” use-case:

```java
import websi.*;
import websi.exceptions.*;
import websi.util.*;

public class AddOffer extends UseCase {
    public void start() throws Exception {
        // Products for which the current supplier has no offers
        Table products = DB.read("SELECT P.code, P.model, C.name AS category, P.description" + \
            "FROM products"
            + "WHERE P.supplier_id = S.id AND S.cust_username = '" + UserID.getUsername() + \
            '"' + "AND P.category = C.category"
            + "AND P.price = S.price"
            + "AND P.stock = S.stock"
            + "AND P.delivery_time = S.delivery_time"
            + "AND P.product_code = S.product_code"
            + "AND P.description = S.description");
        // Visible columns - according to use-case spec.
        String[] add_vis_cols = {"code", "model", "category", "description", "price",
            "supplier", "city", "address"};
        Select.oneOrMore("offer_ids", products,
            add_vis_cols, "id", null,
            "The offers you have added:");
```
FROM product P, category C
WHERE P.category_code=C.code + 
AND P.code NOT IN + 
(SELECT product_code FROM offer + 
WHERE supplier_id=" + UserID.getUsername() + ")

// A foreseen exception
if (products.getRowsNr() == 0) {
    Output.scalar("You have given offers for all the products!", 
    null);
    return;
}

String[] visible_cols = {"category", "model", "description"};
Select.one("product_code", products, visible_cols, "code", null, 
"Select a product to add an offer for:");
UserComposed.any("price", "0.0", "Give your price:");
Action.action("addOffer", "Add your offer");

public void addOffer() throws Exception {
    int product_code = Fetch.Int("product_code");
    double price = Fetch.Double("price");

    try {
        DB.write("INSERT INTO offer (product_code, supplier_id, price) VALUES"+ 
        " (" + product_code + "," + UserID.getUsername() + "," + 
        price + ")");
        Output.scalar("Your offer has been added successfully!", null);

        // Obtain the product details as required by the spec.
        // This query cannot return an empty table, if the INSERT succeeded,
        // because of the FOREIGN KEY constraint
        Table product_cat_model = DB.read( 
            "SELECT C.name, P.model FROM category C, product P" + 
            " WHERE P.code=" + product_code + 
            " AND P.category_code= C.code");

        // Output the product details and the price
        Output.scalar(product_cat_model.getString(1, 
            "name"),
            "Category:");
        Output.scalar(product_cat_model.getString(1, "model"),
            "Model:");
        Output.scalar("" + price, "Price:"");
    } catch (CheckViolationException exc) {
        Output.scalar("The price should be positive!", 
            "Invalid price specified: "+ price);
    } catch (PrimaryKeyOrUniqueViolationException exc) {
        // The offer already exists! How did this happen?
        Table offer_details = DB.read( 
            "SELECT O.price, P.model, C.name" + 
            " FROM offer O, product P, category C" + 
            " WHERE O.product_code=" + product_code + 
            " AND O.supplier_id=" + UserID.getUsername() + 
            " AND O.product_code=P.code AND P.category_code=C.code"); 
        Output.scalar( 
            "Category: " + offer_details.getString(1, "name") + 
            ", Model: " + offer_details.getString(1, "model") + 
            ", Price: " + offer_details.getString(1, "price"),
"You have already given an offer for this product!";
}

try {
    throw (ForeignKeyViolationException exc) {
        // Supplier or product removed
        Output.scalar(
            "Illegal supplier ID or the product does not exist!",
            "Offer cannot be added!");
    }
}

The "Manage Offers" use case:
import websi.*;
import websi.exceptions.*;
import websi.util.*;

public class ManageOffers extends UseCase {
    public void start() throws Exception {
        // All the offers the current supplier has given
        Table offers = DB.read(
            "SELECT O.id, C.name AS Category, P.model, O.price" +
            " FROM offer O, product P, category C" +
            " WHERE O.supplier_id=" + UserID.getUsername() +
            " AND O.product_code=P.code AND P.category_code=C.code");

        // A foreseen exception
        if (offers.getRowsNr() == 0) {
            Output.scalar("You have no offers!", null);
            return;
        }

        String[] visible_cols = {"category", "model", "price"};
        Select.one("offer_id", offers, visible_cols, "id", null,
            "Select the offer you wish to manage:");

        Action.action("askForNewPrice", "Update the price");
        Action.action("deleteOffer", "Delete the offer");
    }

    public void askForNewPrice() throws Exception {
        offer_id = Fetch.Int("offer_id");
        // We have the offer ID, we don’t really need all the other details,
        // but it improves usability
        Table offer_details = DB.read(
            "SELECT O.price, P.model, C.name" +
            " FROM offer O, product P, category C" +
            " WHERE O.id=" + offer_id +
            " AND O.product_code=P.code AND P.category_code=C.code");

        if (offer_details.getRowsNr() == 0) {
            Output.scalar("The offer does not exist anymore!",
                "Unexpected error:"");
            return;
        }

        // The price default value is the current price
        UserComputed.any("price", offer_details.getString(1,"price"),
            "Specify the new price for product of category: " +
            offer_details.getString(1, "name") +
            ", whose model is: " +
            offer_details.getString(1, "model");

        Action.action("updateOffer", "Update the offer");
    }
}
public void updateOffer() throws Exception {
    double new_price = Fetch.Double("price");

    try {
        int rows_updated =
            DB.write("UPDATE offer SET price=" + new_price +
                      " WHERE id=" + _offer_id);
        if (rows_updated == 0) {
            Output.scalar("The offer does not exist!",
                          "Unexpected error:");
            return;
        }
        Output.scalar("The offer has been updated successfully!",
                      null);
        // If the update succeeded, this table cannot be empty
        Table new_offer_details = DB.read(
            "SELECT C.name, P.model, O.price" +
            " FROM offer O, product P, category C" +
            " WHERE O.id=" + _offer_id +
            " AND O.product_code=P.code" +
            " AND P.category_code=C.code");

        Output.scalar(new_offer_details.getString(1, "name"),
                      "Category:);
        Output.scalar(new_offer_details.getString(1, "model"),
                      "Model:);
        Output.scalar(new_offer_details.getString(1, "price"),
                      "New price:");
    } catch (CheckViolationException exc) {
        // Foreseen!
        Output.scalar("The price should be positive!",
                      "Invalid price specified: " + new_price);
    }
}

public void deleteOffer() throws Exception {
    try {
        DB.write("DELETE FROM offer WHERE id=" +
                 Fetch.Int("offer_id");
        Output.scalar(
            "The requested offer is no longer in the system.",
            null);
    } catch (ForeignKeyViolationException exc) {
        // Foreseen!
        Output.scalar("There are customers interested in this offer.",
                      "Cannot remove the offer!");
    }

    private int _offer_id = -1;
}
Chapter 6

The Structure of the \textit{WebSI} Framework

The WebSI framework is written in Java programming language. Besides the core Java services provided by J2SE (Java 2 Platform, Standard Edition) \cite{23} technology, WebSI employs the services of J2EE (Java 2 Platform, Enterprise Edition) \cite{24} technology. The enterprise features used by WebSI include:

- Java Servlet Technology \cite{30}
- Enterprise JavaBeans (EJB) Technology \cite{11}
- Java Transaction API (JTA) \cite{32}

In addition, WebSI employs some of the Java XML technologies: the Document Object Model (DOM) API’s and the Extensible Stylesheet Language Transformations (XSLT) API’s. The various XML documents employed by WebSI are constrained by XML schemas \cite{65}.

WebSI generates HTML \cite{20} files for standard Web browsers that support CSS \cite{10} and JavaScript \cite{31} technologies.

When describing the WebSI structure in the following sections, we will provide basic information on the various J2EE and XML concepts. However, the reader is encouraged to get some understanding of the technologies mentioned above, prior to proceeding to the following sections. All the required information can be found in J2EE tutorial \cite{25}.

In the following sections we assume, that the reader has read all the previous sections of this document. In addition, the reader is advised to have WebSI sources unpacked and available, since sometimes we will reference specific source files.

6.1 Overview

Applications constructed with the help of WebSI follow a well-known client-server model. A WebSI application is deployed on a J2EE server and the application’s main activity is to handle client requests and to return responses. The clients may employ any software, as long as requests and responses are sent and received accordingly over the HTTP communication protocol.

An HTTP request typically contains a URL and a set of key-value pairs. The key is the name of an HTTP request parameter and the value is the parameter’s value. Multi-value parameters are also supported, by associating several values with the same key. An HTTP response is typically a file.
When an HTTP request arrives to a J2EE server with a deployed WebSI application, this request is passed to an HTTP servlet and handled by it. HTTP servlets are Java objects, whose classes are derived from the javax.servlet.http.HttpServlet class. WebSI defines a single servlet class, called websi.EngineServlet, that handles all the requests (see Fig. 6.1).

The HttpServlet class defines methods for handling all the possible kinds of HTTP requests. EngineServlet, however, handles only POST and GET HTTP requests (other kinds of requests are rarely used by client software), by overriding the doGet and doPost methods of HttpServlet class. EngineServlet handles both kinds of requests in the same way, by directly delegating to its private method called process_request (see Fig. 6.2).

Figure 6.1: The EngineServlet class

Figure 6.2: Handling GET and POST HTTP requests by EngineServlet
An HTTP GET request is a single string, that is a combination of a URL and a set of key-value pairs:

```
URL?key1=value1&key2=value2&...
```

An HTTP POST request, on the other hand, contains a URL for the request, but the key-value pairs are not submitted in the same string, they are submitted separately.

A J2EE server usually handles several requests concurrently. Each request is handled in a separate thread. A J2EE server usually manages a pool of servlet instances. When an HTTP request arrives, the J2EE server picks one of the servlet instances and lets it handle the request in a separate thread. In addition, each servlet instance may be used by several threads concurrently, to handle several concurrent requests. This means that servlet methods must be properly synchronized, e.g., servlet data members must be accessed in a properly synchronized way. The EngineServlet implementation is, of course, properly synchronized.

The EngineServlet’s response to a request is an XML file. This file holds all the response data, but no presentational details. In general, this XML file may be directly sent to a client. The client then may use any XML-parsing technology and any interaction style (IS) in order to present the contents of the XML file to the user. However, for the sake of simplicity, right before sending the XML file to the client, EngineServlet transforms it to an HTML file. The HTML file is then sent to a client as a response to the client’s request.

The XML-to-HTML transformation is performed by EngineServlet using the popular XSL [66] technology, provided via Java XSLT API’s. In order to transform an XML file to HTML using XSLT, an XSL stylesheet is required. The stylesheet is also an XML file, in a special format, that describes the various transformational instructions. The XSL stylesheet used by EngineServlet is, in fact, the WebSI IS component, since it defines how the final HTML file will look like.

Each EngineServlet instance stores a local javax.xml.transform.Transformer object (see Fig. 6.1), which is a part of Java XSLT API. When an EngineServlet instance is created by the J2EE server, this transformer is configured with an XSL stylesheet, which has been specified during deployment of a WebSI application. Once configured, the Transformer can transform any amount of WebSI XML files to HTML, using the stylesheet specified during deployment. See Sec. 6.2.1 for more information on how the XSL stylesheet is constructed.

### 6.1.1 The Kinds of Requests a WebSI Application Handles

When a WebSI application is running, it can receive several kinds of requests from the client software:

**Use-case initiation request** This request instructs WebSI to start the execution of a use case. In that case, the start interaction method of the required use case is executed.

**Use-case operation request** This request instructs WebSI to execute an interaction method of a use case, whose execution has been already initiated.

**Log-in request** This request tells WebSI that a user wishes to log in to a certain security role. The log-in operation is performed by the inner use case of WebSI, implemented in the websi.LoginLogout use-case class (see Fig. 6.3). A log-in request initiates this use case, i.e., its start interaction method is executed. This method
generates the UI, that enables the user to supply her username and password, and to approve. User approval results in executing the login interaction method, that checks the username and the password, and registers the user as logged in if they are correct.

**Log-out request** This request tells WebSI that a logged-in user wishes to log out. To accomplish the log-out operation, WebSI calls the logout method of the LoginLogout use case. This is not an interaction method, since the log-out operation does not require any UI.

![UseCase](from websi)

<table>
<thead>
<tr>
<th>UseCase</th>
<th>(from websi)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt; interaction method &gt;&gt;+start():void</td>
<td></td>
</tr>
</tbody>
</table>

![LoginLogout](from websi)

<table>
<thead>
<tr>
<th>LoginLogout</th>
<th>(from websi)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt; interaction method &gt;&gt;+login():void</td>
<td></td>
</tr>
<tr>
<td>+logout():void</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.3: The WebSI LoginLogout use-case class

### 6.1.2 The Structure of the Response XML File

In this section we will describe how the WebSI response XML file looks like, right before it is transformed to HTML. Three kinds of WebSI components take part in the construction of the response XML file: the EngineServlet, the use-case displayer and the WebSI IO activity classes, whose methods are called from interaction methods.

The format of the WebSI response XML file is constrained by the XML schema we have specially developed. The schema is composed of several files referencing one another, they all reside in the war directory. The schema defines the XML format precisely; here we will give a short explanation, accompanied with examples.

The root element of a WebSI response XML file is a websi:document element:

```
<websi:document ...>
...
</websi:document>
```

This element resides in the main WebSI XML namespace, defined by the WebSI XML schema. Besides the main namespace, the WebSI XML schema defines a namespace for XML elements, generated by the use-case displayer component. This component is implemented by the websi.UseCaseDisplayer class (see Fig. 6.4).

In addition, for each IO activity class a separate XML namespace is defined, for XML elements generated by the methods of this activity class. Table 6.1 summarizes the WebSI XML namespaces, their URI’s and the namespace prefixes.
The described XML namespaces design allows a convenient addition of IO activity classes: when a new IO activity class is created, a separate XML namespace should be created for it, and a schema file that constraints XML elements, generated by this new IO activity class, has to be developed. The new XML elements will reside in this separate namespace. The new schema file is integrated into the whole WebSI XML schema similarly to the schemas of existing IO activity classes (see the war/use_case_displayer.xsd schema file).

The first element in the WebSI response XML file defines the name of a WebSI application and the application’s URL.

```xml
<disp:app_name>
  <name>Application Name</name>
  <url>Application URL</url>
</disp:app_name>
```

This element is mandatory and it is generated by the use-case displayer. The application’s URL is generated by the build_use_case_URL method of the UseCaseDisplayer class.

The next element defines the use cases that can be initiated by the user—those permitted for her security role, or just the public use cases, in case the user is not logged in for any security role.
<disp:accessible_use_cases>
  <use_case>
    <name>Name of Use Case 1</name>
    <url>HTTP GET request, that initiates Use Case 1</url>
  </use_case>
  <use_case>
    <name>Name of Use Case 2</name>
    <url>HTTP GET request, that initiates Use Case 2</url>
  </use_case>
  ...
</disp:accessible_use_cases>

This element is also generated by the use-case displayer. This element is optional (there may be no accessible use cases), but if it exists, it must contain one or more use_case elements. The url attributes contain HTTP GET requests, that once submitted, start the use-case execution. The strings for these requests are built by the build_use_case_URL method of the UseCaseDisplayer class.

The next element enables the user to log in for a security role or to log out, in case she is already logged in for some security role. The element either contains a single logout element:

<disp:security>
  <logout>
    <title>To be displayed to the user</title>
    <url>HTTP GET request, that performs the log-out operation</url>
  </logout>
</disp:security>

Or the element contains one or more login elements:

<disp:security>
  <login>
    <title>To be displayed to the user</title>
    <url>HTTP GET request, that starts the log-in to Security Role 1 operation</url>
  </login>
  ...
</disp:security>

The disp:security element is generated by the use-case displayer and is optional (there may be no security roles defined). The url attributes contain HTTP GET requests, that once submitted, perform the log-out operation or start performing the log-in operation (i.e., the user is asked to specify her username and password). The strings for these requests are generated by the build_logout_URL and build_login_URL methods of the UseCaseDisplayer class.

Finally, there comes the element, that holds all the XML content, generated by the execution of a certain interaction method of a certain use-case class:

<disp:uc_content target_url="URL for POST requests">
  ...
</disp:uc_content>
This element is generated by the use-case displayer (although its children are generated by an interaction method execution) and is optional (e.g., because an interaction method can generate no UI, i.e., no content). The target_url attribute holds a URL for submitting the future use-case operation requests, which are POST requests.

The reason for using GET HTTP requests for log-in, log-out and use-case initiation requests, is that all key-value pairs for future requests of these kinds are not dependent on user’s inputs. So it is convenient to construct HTTP strings of these future requests in advance and insert them into the response XML, so that they may be used as-is by the client. The key-value pairs of future use-case operation requests, however, cannot be determined in advance, as they depend on the inputs supplied by the user. So for this kind of requests we leave it to the client software to construct the appropriate POST requests.

If the disp:uc_content element exists, it must contain (besides some additional data) one or more websi:ui_elements. A ui_element describes a single UI element, generated by a method of an IO activity class. Each IO activity class defines a set of ui_elements, that can be generated by its methods. Let us see, for example, the ui_element generated by the any method of the UserComposed IO activity class:

```xml
<uc:ui_element>
  <any id="a fetch ID" kind="short"
       maxlength="50" optional="false"
       default="a default value" caption="a caption"/>
</uc:ui_element>
```

The id, default and caption attributes are the same as passed to the any method. Recall, that an input request may be qualified as mandatory or optional, and that user-composed values are divided into short and long values. This ui_element asks the user to specify a short user-composed value that is mandatory. After the user specifies the value, a possible way of returning it to WebSI is to construct a POST HTTP request that contains a key-value pair, in which the key is a fetch ID and the value is the value that the user has specified. Since this input request is mandatory, the client software is also responsible to ensure that the value the user has specified is non-empty.

In the same way, each IO activity class defines its possible ui_elements. The reader is advised to browse the schema files that constrain the ui_elements of the various IO activity classes.

### 6.1.3 Generating XML Content by WebSI Components

When EngineServlet starts processing a client’s request, it creates a org.w3c.dom.Document object (Fig. 6.5), that represents a response XML file. Document is a part of Java DOM API. Document represents its XML content as a tree. The Document object will be filled with XML content by various WebSI components. The first WebSI component that fills the Document with content is the EngineServlet itself—it creates the root websi:document element. Then EngineServlet passes the Document to the use-case displayer component’s processRequest method (see Fig. 6.4 and 6.8) for further processing.

The use-case displayer, in case an interaction method of a certain use case has to be executed, creates a org.w3c.dom.DocumentFragment (Fig. 6.5) object and passes it to the use-case object, requesting the use-case object to execute the required interaction method. DocumentFragment is a “lightweight” XML container, that can hold a fragment of an XML document. The execution of the interaction method will fill the initially empty DocumentFragment with various ui_elements, according to the IO activity classes methods, called during the execution of the interaction method. After the execution of the
interaction method ends, the use-case displayer creates the `disp:uc_content` element and places the contents of the `DocumentFragment` under this element. Then the use-case displayer generates the `disp:app_name`, `disp:accessible_use_cases` and `disp:security` elements, according to the application deployment settings and to the state of the user (logged in or logged out). The use-case displayer inserts all the elements it generated into the `Document`, passed to it by `EngineServlet`, in a correct order. At that point, the `Document` contains all the needed response XML content. The control is returned back to the `EngineServlet`, and the XML content is transformed to HTML, which is then sent to the client.

### 6.1.4 Managing Multiple Users

As described in Sec. 4.5, each user that operates a `WebSI`-based application has its own private “world”, called a `session`. A session in J2EE is represented by a `javax.servlet.http.HttpSession` object. A J2EE server creates a separate session for each user. Each session has a unique ID, given by the J2EE server. We see a session as a container of objects that belong to a specific user. The `HttpSession` API allows various session management operations, such as inspecting the session’s ID, storing objects in a session and removing objects from a session.

Right after a new session is created, `WebSI` configures it, by storing the various required objects in it. Additional objects may be stored and removed from the session during the session’s lifetime. The configuration is performed by `websi.EngineServletHttpSessionListener`, which is registered as a session life-cycle events (creation and destruction) listener (see Sec. A.8.2).

Following is the list of objects stored in a session. Some of these objects permanently reside inside the session (i.e., until the session’s life cycle ends), others are stored and removed according to `WebSI` needs.

- The `UseCaseDisplayer` object. Each user has her own use-case displayer. The use-case displayer manages all the `UseCase` objects for the user, i.e., instantiates them (on-demand) and stores them in its internal containers. As a result, each user has her separate set of use-case objects, so that a code of a use-case class (except for the SQL queries) can be written as if a single user operates the system.
• Helper objects of user-defined classes, derived from the `websi.SessionObject` class, the so-called *session objects* (see Sec. 4.5.1). Each user has her own instance (created on-demand by `WebSI`) of each session object class. As a result, session object classes can be coded (except for the SQL queries) as if a single user operates the system.

• The `DBProxy` bean, which is used to access the database on behalf of a specific user. Each user has her own `DBProxy` bean. More on this bean in Sec. 6.3.

• A `Boolean` indicating that the current session should be finished, as soon as the ongoing request handling is finished. This `Boolean` is either a `Boolean.TRUE` or not stored at all. It is used for ending sessions manually. When the client programmer issues the `Session.end()` call (as described in Sec. 4.5), `WebSI` stores a `Boolean.TRUE` in the session. The `EngineServlet`, after sending the response to the user, checks for this `Boolean` and if it is present (and is `Boolean.TRUE`), invalidates the session.

• A `String`, which is the username of the user, that is currently logged in. When the user starts operating the system, a new session is created for her. Then, at some point in time, the user may log in for a certain security role, in order to access non-public use cases. After the log-in procedure succeeds, the user’s username is stored in the user’s session. When the user logs out, the username is removed from the session, but the session is not destroyed. The user continues to operate the system within the same session. The user may log in again, this time for a different security role, if she wishes.

• A `String`, which is the name of the security role, for which the user is currently logged in. This object is handled in the same way as the user’s username.

• A `String`, which is a *server-side cookie*, generated by `WebSI` for the user that is currently logged in. This cookie is a combination of the log-in time, the user’s username and the client’s IP address. The cookie is generated by `WebSI` and stored in the session when the user logs in, and it is removed, when the user logs out.

We will now explain why the server-side cookie is needed and how it is managed. If the user is logged in for some security role, the use-case displayer, in each XML response file it generates, stores the user’s cookie and the session ID. The cookie and the ID are placed under `uc_content` element (in addition to various `ui_element`s). The client software is responsible to include both these values in each POST request it submits. (The GET request strings, generated by the use-case displayer, already contain those values.) For each request, the use-case displayer checks that the session ID that arrived in the request is identical to the ID of the session, in which this displayer is stored. If the values are different (or the request does not contain a session ID at all), the displayer does not allow the execution of any use case. In addition, for use cases that require the user to be logged in for a certain security role, the displayer compares the cookie that arrived in the request to the cookie that is stored in the session. In case the cookies differ or the request does not contain the cookie, the displayer does not allow the use-case execution.

### 6.1.5 Storing Common Application Data in a `ServletContext`

It is often required to store a piece of common data in a such a location, where it can be accessed by any component of a J2EE application. In `WebSI`, for example, the
security roles specifications and the application name are such common data. For that purpose, J2EE defines a scope, called a Web context. A Web context can be thought of as a container that can be accessed by any J2EE application component. There is a single Web context per J2EE Web application. A Web context is accessed through the javax.servlet.ServletContext API.

When a J2EE Web application is deployed, its Web context is created. In order to manage all the needed information in a Web context, WebSI defines a websi.EngineServletContextListener (see Sec. A.8.2), which is invoked by the J2EE server during Web context life-cycle (creation and destruction) events. When the Web context of a WebSI application is created, the EngineServletContextListener stores all the needed data in it.

Since the objects stored in the Web context are shared by all the application components, these objects must be accessed in a properly synchronized manner. The objects, that WebSI stores in a Web context, are either immutable (e.g., String) or not altered (just inspected) during the runtime of a WebSI application. In addition, the client programmer of a WebSI framework has no access to the Web context data. Therefore, no special synchronization for WebSI Web context objects is required.

We will now describe the data that WebSI stores in a Web context.

- A String, which contains the name of a WebSI application.
- A Boolean, which specifies whether the DBProxy bean uses the remote access interface or the local access interface (see Sec. 6.3.1 and the J2EE tutorial [25]).
- A websi.db.DBVendorSpecificSupport object for the DBProxy bean (see Sec. 6.3.5).
- A websi.SecurityRolesSetup object that describes the properties of all security roles, defined for a WebSI application. For each security role, SecurityRolesSetup holds the name of the table in the database that stores usernames and passwords for the users of this security role. In addition, the names of this table’s columns that store usernames and passwords are held. The API of the SecurityRolesSetup class is quite simple and it is presented on Fig. 6.6.
- A websi.ApplicationUseCasesSetup object that stores information on application’s use cases. For each use case, which may be initiated directly by the user, ApplicationUseCasesSetup holds a mapping from its name to its fully-qualified class name and vice versa. In addition, for such a use case, ApplicationUseCasesSetup holds the names of the security roles, whose users are authorized to execute the use case. For a use case that cannot be directly initiated by the user, i.e., for a use case that is only <<included>> by other use cases, only the security roles are stored. The API of this class is quite simple, and it is presented on Fig. 6.7.

The websi.ApplicationUseCasesSetup object is used by the use-case displayer to obtain the names and the fully-qualified class names of use cases, that can be directly initiated by the user. The names are used for the UI, which enables the user to initiate use cases. The class names are used to instantiate the use-case classes.

The security roles, stored for each use case, are used to authorize the execution of use cases, according to the security role, for which the user is logged in.
6.1.6 Passing Data between Various WebSI Components

When a UseCaseDisplayer object is created, it receives (through a constructor) two objects that identify the environment, in which this displayer will operate. These two objects are stored by the use-case displayer during all its lifetime. These two objects are:

- An HttpSession object that identifies the session, in which the displayer operates. (An HttpSession can always be obtained from an HttpServletRequest, using its getSession method. But since an HttpSession is just a facade [16], each time this method is called, a new HttpSession object, pointing to the same session data, may be created. In order to avoid the creation of unnecessary facades, we store an HttpSession in a use-case displayer.)

- A ServletContext object that identifies the Web application, in which the displayer operates. (A ServletContext can always be obtained from an HttpServletRequest, but potentially may also be just a facade that points to the Web context data. For similar reasons as with HttpSession, we store a ServletContext in a use-case
The same two objects, for the same reasons are stored in every UseCase object when it is instantiated.

When EngineServlet invokes the processRequest method of UseCaseDisplay for further request processing, it passes the following objects to the use-case displayer:

- A Document object. This object has been created by EngineServlet and it represents the response XML document. The reason it is passed between WebSI components, is that the Document interface defines factory methods for creating the various XML content, such as XML elements, text nodes, comments etc. For example, in order to create XML elements, the following Document methods can be used (see Fig. 6.5):

  ```java
  public Element createElement(String tagName);
  // Create an Element, that resides in a certain namespace
  public Element createElementNS(String namespaceURI,
                                  String qualifiedName)
  ```

- A javax.servlet.http.HttpServletRequest object. This object is supplied to the EngineServlet by the J2EE server, and it is passed as-is between the various WebSI components. This object stores various data about the HTTP request, such as request parameters and their values.

- An javax.servlet.http.HttpServletResponse object. This object is supplied to the EngineServlet by the J2EE server. This object assists the use-case displayer and the EngineServlet in constructing the response. For example, EngineServlet uses this object in order to obtain a java.io.PrintWriter, through which the HTML content is sent to the client. A use-case displayer uses the HttpServletResponse in order to encode the various URL’s it generates (see the J2EE tutorial [25] for more details).

When the use-case displayer invokes the execute method on a use-case object, asking it to execute the required interaction method, the following objects are passed:

- A Document object. This is the same object that the use-case displayer has received. It is used for creating XML content by IO activity classes.

- A org.w3c.dom.Node (see Fig. 6.5), that represents a root XML element, under which the XML content generated by IO activity classes should be placed. This Node is actually a DocumentFragment created by the use-case displayer.

- A javax.servlet.http.HttpServletRequest object. This is the same object that the use-case displayer has received. It is used by the websi.Fetch activity class for accessing the HTTP request parameters.

Figure 6.8 demonstrates how the various WebSI components typically handle a request from a client. The figure shows the processing of a POST request; a GET request is processed in the same way.

First the J2EE server invokes the doGet or the doPost method of EngineServlet. Whatever method has been invoked, it directly delegates to the process_request method
of EngineServlet. This method obtains a UseCaseDisplayer object that belongs to the session, on behalf of which the request has arrived. Then EngineServlet creates a Document object, that will hold the XML response for this request. Now EngineServlet invokes the processRequest method of the UseCaseDisplayer object. The use-case displayer typically has to invoke an interaction method of a use-case class. So it obtains a use-case object of the needed use-case class; this use-case object belongs to the same session as the use-case displayer. The use-case displayer creates a new DocumentFragment, which is a Node under which the XML content generated by the interaction method execution will be placed, and invokes the execute method on the use-case object. The execute method is responsible to invoke the required interaction method on the use-case object. More details on the execute method in Sec. 6.10 and in Sec. 6.5.2.

6.1.7 How an Activity Class Obtains Correct Objects

WebSI activity classes are used from within the code of an interaction method, in order to perform such activities as communicating with the user, manipulating the database, obtaining user’s identification details etc. The services of the activity classes are reached through their static methods, for example:

```java
public void someInteractionMethod() throws Exception {
    ...
    // Fetch an input value supplied by the user
    int age = Fetch.Int("age");
    ...
    // Manipulate the database
    DB.read("...");
    ...
    // Obtain user’s identification
    String username = UserID.getUserame();
```
In order to support a certain activity, a static method of an activity class typically needs to obtain objects, which belong to a specific user, i.e., to a session, to which a use-case object, whose interaction method is executing, belongs. For example:

- The `UserId.getUsername` method must access the correct `HttpSession` object, where the logged in user's username is stored.
- The `DB.read` method must access the DBProxy bean that belongs to the user, on behalf of which the interaction method is executing. For that purpose, the method must first access the correct `HttpSession` object, where the bean is stored.
- The methods of the `Fetch` activity class must access the correct `HttpServletRequest` object, in order to obtain HTTP request parameters and their values.
- The methods of IO activity classes need to access a correct `Document` object in order to generate their XML content. Moreover, they need a correct `Node` object, under which the generated XML content should be placed. Both objects must belong to a specific ongoing request-processing sequence (on behalf of which the interaction method is executing).

Another complication is that static methods of activity classes are invoked concurrently from multiple use-case objects, when several requests are processed concurrently by `WebSI`.

We will now explain the general mechanism `WebSI` provides, which allows an activity class to access correct objects, also in case of multiple concurrent requests.

The following objects may potentially be required by an activity class:

- `ServletContext`
- `HttpSession`
- `Document`
- `Node`, under which the XML content should be placed (this `Node` is actually a `DocumentFragment`)
- `HttpServletRequest`

The same `ServletContext` object can be used as long as a `WebSI` application runs. The `HttpSession` object identifies a specific user, and this object can be used as long as the session is active. The last three objects are specific to a request. For each request, new `Document`, new `DocumentFragment`, `Node` and new `HttpServletRequest` are required.

`WebSI` stores all these objects in each `UseCase` object it creates. The first two objects are stored only once, when a new use-case object is created. It is possible to store an `HttpSession` once, since a use-case object belongs to the same single session through all its lifetime. The two objects are stored using the `setConstantData` method of the `UseCase` class (see Fig. 6.9). The remaining objects are passed to a use-case object, when its interaction method has to be executed, by invoking the `execute` method of the `UseCase`
class (Fig. 6.9). In addition, the `UseCase` class provides an API, through which all these objects can be inspected, as presented on Fig. 6.9.

So when an interaction method of a use-case object is executed, the use-case object contains all the data, that potentially may be required by the various activity classes. In order to obtain this data, an activity class must first obtain a reference to the use-case object, and then to use the API on Fig. 6.9. The last question, therefore, is how an activity class obtains a reference to a use-case object, which calls one of its methods.

Right before an interaction method of a certain use-case object is about to execute, `WebSI` associates the use-case object with a thread, that is executing the interaction method. Java provides a standard mechanism for associating a thread with an object, through a `java.lang.ThreadLocal` class (see Fig. 6.10). This class allows each thread to access its own independent object. (A simple way to implement this association is to employ a synchronized `Map`, in which the keys are threads and the values are the desired objects. The problem of this approach is that the `Map` may become a synchronization bottleneck, and that the programmer must ensure that unneeded threads and objects can be garbage-collected. So currently Java provides a better solution, in which each `Thread` object holds its own thread-local variables.)

All `WebSI` activity classes have a common base, which is the `websi.ActivityClass` class, presented on Fig. 6.10. This class holds a single static data member, which is a `ThreadLocal` instance. This object holds all the thread-to-use case associations. In addition, the class has an API that allows all the derived activity classes to obtain the required objects (associated with the current thread) in their static methods.

The `execute` method of the `UseCase` class, prior to invoking the required interaction method, performs the following code:

```java
// Associate the current thread with this UseCase
ActivityClass.setUseCaseForCurrentThread(this);
```

When the interaction method execution finishes, the association is removed:

```java
...
```
Figure 6.10: Associating a Thread with a UseCase

```java
// remove the association
ActivityClass.unsetUseCaseForCurrentThread();

The design described above allows a convenient addition of new activity classes. A new activity class must inherit (not necessarily directly) from the ActivityClass. The new class must define public static methods that perform the various activities. In the code of these static methods all the required objects may be accessed by invoking the inherited methods of the ActivityClass. It may be noted, that adding a new activity class requires only coding the new class. No other WebSI classes have to be changed or recompiled.

To conclude, the design described above allows easy modifications and extensions of WebSI input and output semantics set, database access methods etc.

6.1.8 Handling One Request of a User at a Time

A WebSI application processes at most one request from a certain client at a time. We have taken this design decision in order to simplify WebSI synchronization measures, and to enable the client programmer write her code as if a single user operates the system. Two requests coming from the same user concurrently may touch the same objects, e.g., the data members of the same use-case object. This may require from the client programmer to invest effort in properly synchronizing her code, which may be non-trivial.

The effect of processing at most one request from a certain user at a time is achieved by using the UseCaseDisplayer object that belongs to this user as a lock. The synchronization takes place in the process_request method of EngineServlet:

```java
private void process_request(HttpServletRequest req, ...) {
    HttpSession session = req.getSession();
    UseCaseDisplayer displayer =
        (UseCaseDisplayer) session.getAttribute(SESSION_DISPLAYER_ATTR_NAME);

    // Starting from here - we are processing at most one a request at a time
    synchronized (displayer) {
        // from a specific user
        Process the request
    }
    // Process the request
}
```
Synchronizing on a use-case displayer object, which is unique for each user, ensures that even if several requests are coming from a certain user, and the threads handling these requests employ different EngineServlet objects, at most one thread will proceed, others will be blocked.

### 6.2 The WebSI Interaction Style Component

The IS component is an XSL stylesheet, that transforms WebSI response XML files to HTML. In this section we will describe the structure of a typical XSL stylesheet, thus giving a general methodology of how new IS’s can be developed and integrated into WebSI. The reader is encouraged to browse the available XSL stylesheets, they are located in the war directory.

#### 6.2.1 The Structure of the XSL Stylesheet

The XSL stylesheet, which is used by EngineServlet to transform response XML files into HTML, is combined from several sub-stylesheets. Each sub-stylesheet is capable of handling XML content, generated by a certain WebSI component:

- There exists a sub-stylesheet for the use-case displayer content. This stylesheet handles such XML elements as disp:app:name, disp:security and disp:accessible:use:cases.

- There exists a sub-stylesheet for each IO activity class. This sub-stylesheet handles the various ui:elements, generated by the methods of this activity class.

Typically, a use-case displayer sub-stylesheet creates the basic HTML structure, that contains html, head and body tags:

```xml
<html>
  <head>
    <title>Application Name</title>
    ...
  </head>
  <body>
    ...
  </body>
</html>
```

The head element typically links CSS stylesheets and JavaScript sources employed by the HTML document. The body element contains the visible HTML content. A typical use-case displayer sub-stylesheet, that creates the described HTML structure, looks like this:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  xmlns:websi="http://www.cs.technion.ac.il/~lyakasal/websi"
  xmlns:disp="http://www.cs.technion.ac.il/~lyakasal/websi/UseCaseDisplayer"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="html" indent="yes"/>
  <xsl:template match="/">
    <html>
```
The use-case displayer sub-stylesheet will eventually become the final XSL stylesheet, i.e., the complete IS component. This will happen after the IO activity classes sub-stylesheets are integrated into the use-case displayer sub-stylesheet. This integration happens in the code of the init method of the EngineServlet class, which is invoked by the J2EE server right after it creates a new EngineServlet instance. (The EngineServlet’s init method overrides the init method of HttpServlet).

We will now describe how the various sub-stylesheets are integrated into the final XSL stylesheet that becomes the IS component. During the deployment, it is required to specify the name of the use-case displayer sub-stylesheet, and the names of the IO activity classes sub-stylesheets (see Sec. A.6.1). Then the EngineServlet, in the code of its init
method, dynamically creates `xsl:include` elements and inserts them into the use-case displayer sub-stylesheet. (The sub-stylesheet, being also an XML file, is loaded as a `Document` and this `Document` is modified, not the original XSL file.) The `xsl:include` element includes the contents of one XSL stylesheet into another. For example, suppose that during deployment, an IO activity class sub-stylesheet named `output_default.xsl` has been specified; this sub-stylesheet handles the XML content generated by the methods of the `Output` activity class. The `EngineServlet` will create and insert the following element into the use-case displayer sub-stylesheet:

```
<xsl:include href=".../output_default.xsl"/>
```

(The actual path to the stylesheet is determined by `EngineServlet` at runtime, using the `getRealPath` method of `ServletContext`.)

An IO activity class sub-stylesheet provides HTML implementations for each `ui_element`, that may be generated by the methods of this class. Here is a skeleton sub-stylesheet for the `Output` activity class:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
    xmlns:output="http://www.cs.technion.ac.il/~lyakasa/websi/activities/Output">

    <xsl:template match="output:ui_element/scalar">
        <!--Implement it-->
        ...
    </xsl:template>

    <xsl:template match="output:ui_element/table">
        <!--Implement it-->
        ...
    </xsl:template>

    <xsl:template match="output:ui_element/error">
        <!--Implement it-->
        ...
    </xsl:template>

</xsl:stylesheet>
```

Typically, an IO activity class sub-stylesheet needs to link its own CSS and JavaScript files to the final HTML document. The HTML tags that perform this linkage (the `link` and the `script` tags) must appear inside the HTML `head` element. The templates for the various `ui_elements`, which, in principle, should link the required CSS and JavaScript files, are, however, invoked only when inside the HTML `body` element. To solve this problem, `WebSI` provides the following mechanism.

Each IO activity class sub-stylesheet must define an XSL `named template`, whose name is identical to the name of the sub-stylesheet. For example, the `output_default.xsl` sub-stylesheet must define the following template:

```xml
<xsl:template name="output_default.xsl">
    <!--Link the relevant CSS/JavaScript files-->
</xsl:template>
```

Inside this template the needed CSS and JavaScript files are linked, like this:

```xml
<xsl:template name="output_default.xsl">
    <link href="/websiapp/output_default/output_default.css" rel="stylesheet" type="text/css"/>
    <script type="text/javascript"
        src="/websiapp/output_default/output_default.js"/>
</xsl:template>
```
But how and when this named template will be invoked? It will be invoked when inside the HTML head element, by the use-case displayer sub-stylesheet as explained below.

A use-case displayer sub-stylesheet must also define a named template, whose name is identical to the name of the sub-stylesheet. For example, a use-case displayer sub-stylesheet, named twocolumns_disp.xsl, will define the following template:

```xml
<xsl:template name="twocolumns_disp.xsl"/>
```

This template must be left empty and it must be explicitly invoked in the use-case displayer sub-stylesheet, when inside the HTML head element. A typical invocation of this template looks like this (for the twocolumns_disp.xsl sub-stylesheet):

```xml
<html>
  <head>
    <!--Invoke only if there is some content, generated by an interaction method execution-->
    <xsl:if test="count(/websi:document/disp:uc_content) > 0">
      <xsl:call-template name="twocolumns_disp.xsl"/>
    </xsl:if>
  </head>
  <body>
    ...
  </body>
</html>
```

But how and when the use-case displayer named template is filled by content? This is performed by EngineServlet, by inserting xsl:call-template elements into the named template of the use-case displayer sub-stylesheet. Each xsl:call-template element explicitly invokes the named template from an IO activity class sub-stylesheet.

For example, suppose that a use-case displayer sub-stylesheet is twocolumns_disp.xsl and, in addition, four sub-stylesheets for four WebSI IO activity classes have been specified: output_default.xsl, user_composed_default.xsl, action_default.xsl and select_default.xsl. The EngineServlet will first look for the named template of the use-case displayer sub-stylesheet; the template’s name is similar to the name of the sub-stylesheet, i.e., it is twocolumns_disp.xsl. Then, EngineServlet will insert the following elements into this template:

```xml
<xsl:template name="twocolumns_disp.xsl">
  <xsl:call-template name="output_default.xsl"/>
  <xsl:call-template name="user_composed_default.xsl"/>
  <xsl:call-template name="action_default.xsl"/>
  <xsl:call-template name="select_default.xsl"/>
</xsl:template>
```

In this way, the named templates of the IO activity classes sub-stylesheets will be invoked when inside the HTML head element, as required.

### 6.2.2 The form Element in the Final HTML File

The final HTML file generated by WebSI contains a single HTML form element, generated by the use-case displayer sub-stylesheet. The various ui elements, generated as a result of an interaction method execution, will be transformed to HTML, and this HTML will be placed under the form element. For WebSI input methods, i.e., those that require
the user to supply input, the HTML elements under the form element will hold the input values. All input values will be sent to the WebST application, when the form is submitted. The form submission happens, for example, when the user operates the control, generated by the action method of the Action activity class (e.g., a push button or a hyperlink). The form submission is described in Sec. 6.2.4.

Here is how a form element generated by the use-case displayer typically looks like:

```html
<form method="post" action="\'target_url\' attribute of \'disp:uc_content\'"
    name="websi_use_case_form" id="websi_use_case_form">
    ...
</form>
```

Let us explain this element in brief. The form submission method is set to POST, because POST allows to send any number of values to the server. (The HTTP GET request is sent as a single string and the length of this string is limited.) The action attribute defines the URL, to which the form should be submitted. This URL is specified in the target_url attribute of disp:uc_content element in the response XML file. The form is given a name and an ID.

We will now explain why and how some HTML elements are named. Then we will explain, why we specify both the name and the id attributes.

Any HTML element, that holds a value to be submitted to a server, should have a unique name. The value of the element is submitted as key-value pair, where the key is the HTML element’s name and the value is the value this element holds. Sometimes, several HTML elements share the same name, e.g., a set of check boxes. In that case, their values are submitted as a multi-value parameter, i.e., the same key maps to multiple values.

HTML elements generated by WebST, residing under the form element described above, typically have names that are identical to or derived from fetch ID’s, supplied by the client programmer. For example, suppose the programmer has written the following line of code:

```javascript
UserComposed.anyOptional("age", "20",
    "Optionally, please specify your age:");
```

The following HTML may possibly be generated by WebST:

```html
<form ...> <!--As described before--->
    ...
    <p>
        Optionally, please specify your age:<br>
        <input type="text" name="age" id="age" maxlength="50" size="25"
            value="20">
    </p>
    ...
</form>
```

In this example, the anyOptional input semantics is implemented by a text field. The text field resides inside the form element. The text field’s name is similar to the fetch ID that the programmer has supplied. When the form is submitted, the value of the text field will be submitted in a key-value pair, in which the key is the text field’s name, i.e., the fetch ID.

WebST checks that all fetch ID’s supplied by the client programmer are unique within an interaction method, so that the names of the various HTML elements inside the form
element are also unique. In addition, fetch ID’s that have a prefix _websi_ are reserved for system needs and cannot be used by the client programmer (WebSI checks this also).

In the example, the `form` element has a reserved known name. As a result, it can be referred to from various places, such as from client-side scripts.

The reason we use both the `name` and the `id` attributes is that we strive our HTML to be as close as possible to XHTML [20], which is a standard that tries to make an HTML content also a proper XML content. In XHTML, the `id` attribute replaces the HTML `name` attribute. But since at present many browsers do not support XHTML, it is recommended to use both attributes with identical values.

### 6.2.3 Indirect Fetchers

In many cases, an input value supplied by the user can be passed as-is to the client programmer. In such cases, the value should be submitted in an HTTP key-value pair, where the key is a fetch ID, supplied by the programmer, and the value is the input value, supplied by the user. Typically such a value, before it is submitted, is held in an HTML element, whose name is the fetch ID.

There are some cases, however, in which an input value must undergo some server-side preprocessing, before it can be used by the client programmer. For example, suppose the client programmer uses the following input method:

```java
UserComposed.yesNo("kosher_meal", Boolean.TRUE,
    "Would you like a kosher meal?");
```

Suppose that a sub-stylesheet for the `UserComposed` activity class decides to implement this semantics with a check box:

```html
<form ...> <!--As described before-->
...
<p>
Would you like a kosher meal?
<input type="checkbox" name="kosher_meal" id="kosher_meal"
    value="true" checked>
<!--Wrong!!!-->
</p>
...
</form>
```

This HTML states, that if a check box value is submitted, it is submitted in the following key-value pair:

<table>
<thead>
<tr>
<th>Key</th>
<th>kosher_meal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>true</td>
</tr>
</tbody>
</table>

But the check box value is only submitted, if a check box is checked. So if the check box is checked and a form is submitted, the programmer will fetch `true`, which is correct. But if the check box is unchecked and the form is submitted, then no value will be submitted for the `kosher_meal` key, and a runtime error will occur, when the programmer tries to fetch the value of `kosher_meal` fetch ID. Obviously, in this case some server-side pre-processing is needed: when there is no key-value pair, whose key is `kosher_meal`, and the programmer performs:

```java
boolean cosher = Fetch.Boolean("kosher_meal");
```

the `false` value should be returned.
WebSI provides a general mechanism for such situations. The idea is that the value is not fetched directly from the HTTP request. Instead, an indirect fetcher method of an indirect fetcher class is invoked, and this method returns the correct value. Here is an example for an indirect fetcher method and class, that solve the problem stated above:

```java
package websi.indirect_fetchers;
...

public abstract class UserComposedDefault {
    public static String fetchYesNo(HttpServletRequest req, String fetch_id) {
        if (RequestProcessingTools.parameterExists(req, fetch_id))
            return "true";
        return "false";
    }
}
```

This indirect fetcher class has been designed for the UserComposed activity class, for its default XSL sub-stylesheet, that’s why the class resides in the websi.indirect_fetchers package and the class name is UserComposedDefault. The fetchYesNo method is responsible for handling the input, resulting from the yesNo method of the UserComposed activity class. The method uses a WebSI helper class, called RequestProcessingTools, in order to determine whether the fetch ID is present in the request. If the fetch ID is not present, the method infers that the corresponding check box has been left unchecked and returns the false value to the programmer.

An indirect fetcher class should be a public class, and its fetcher methods should have the following signature:

```java
public static String methodName(HttpServletRequest req, String fetch_id);
// Or, for multi-value parameters:
public static String[] methodName(HttpServletRequest req, String fetch_id);
```

All the Strings returned by a fetcher method should be trim()med.

But how WebSI knows when to invoke the fetcher method? WebSI assumes that if a value for a certain fetch ID has to be obtained through an indirect fetcher, than the HTTP request has to contain the following key-value pair:

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>websi间接_fetch_concatenated with a fetch ID</td>
<td>A string websi间接_fetch_concatenated with a fetch ID that should be used</td>
</tr>
</tbody>
</table>

For the check box problem, the following key-value pair should be present:

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>websi间接_fetch_kosher_meal</td>
<td>websi间接_fetchers.UserComposedDefault.fetchYesNo</td>
</tr>
</tbody>
</table>

In addition, if the check box has been checked, its key-value pair should be also present in the request:

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kosher_meal</td>
<td>true</td>
</tr>
</tbody>
</table>

The following HTML is a possible one for the check box example:
Would you like a kosher meal?
<input type="checkbox" name="kosher_meal" id="kosher_meal" value="true" checked="checked" />
<input type="hidden" name="websi_indirect_fetch_kosher_meal" id="websi_indirect_fetch_kosher_meal" value="websi.indirect_fetchers.UserComposedDefault.fetchYesNo">

This HTML should be generated by a sub-stylesheet of the UserComposed activity class.

When the programmer wishes to fetch an input value that has a certain fetch ID, WebSI checks first whether the key websi_indirect_fetch_concated with the fetch ID is present in the request. If yes, WebSI invokes the fetcher method, according to the value of this special key. Otherwise, WebSI checks for a key that is similar to the fetch ID.

### 6.2.4 Submitting the Form

The disp:uc_content element in the response XML file contains, besides the various ui_element$s, zero or more hidden_variable elements. A client software, when it encounters such an element, must define a variable, whose value is not presented to the user, but rather must be submitted in each use-case operation request. Currently, the following hidden_variable$s are used by WebSI:

```xml
<disp:uc_content target_url="URL for POST requests">
  <hidden_variable id="websi_use_case_class" />
  <hidden_variable id="websi_interaction_method" />
  <hidden_variable id="websi_session_id">session ID</hidden_variable>
  <hidden_variable id="websi_user_cookie">A server-side cookie for the logged-in user
</hidden_variable>

  <!--The various "ui_element"s-->
  ...
</disp:uc_content>
```

The websi_use_case_class and websi_interaction_method variables will hold the name of a use-case class and the name of an interaction method of this class to be executed. Initially their values are empty. The websi_session_id variable holds the ID of the session, to which the user belongs. This ID must be submitted with each request, and it is checked by the use-case displayer, in order to make sure, that the displayer handles only requests, that belong to the same session as the displayer does. The websi_user_cookie variable holds a server-side cookie, generated by WebSI for the user, in case she is logged in for a certain security role. In case the user is not logged in, this variable is not present. Similar to the session ID, the cookie (if exists) must be submitted with each request and is checked by the use-case displayer. It can be noted, that the variable names follow the reserved names pattern.

The HTML implementation for the hidden_variable element is typically with HTML input elements, whose type is hidden:

```xml
<form ...> <!--As described before-->
  ...
</form>
```
When the form is submitted, all the values of `<input type="hidden" ...>` elements are submitted automatically by a browser, together with the values of all the other value-holding HTML elements inside the form.

Submitting the form always results in a use-case operation request, i.e., a request to execute an interaction method of some use-case class. Therefore, the form is typically submitted with the following snippet of JavaScript code:

```javascript
websi_use_case_class.value = 'name of a use-case class';
websi_interaction_method.value = 'name of an interaction method';
document.websi_use_case_form.submit();
```

The code first sets the values of hidden variables that denote the use-case class and the interaction method to be executed. Then the code refers to the form element using its name and submits the form.

For example, suppose the client programmer in the code of AddOffer use-case class wrote the following line:

```javascript
... Action.action("addOffer", "Add the new offer!"); ...
```

The following XML will be generated by WebST:

```xml
<action:ui_element>
  <action caption="Add the new offer!" uc_class_name="AddOffer" interaction_method_name="addOffer"/>
</action:ui_element>
```

And this XML may result, for example, in the following HTML, that employs a push button:

```html
<form ...> <!--As described before--->
  ... <input type="button" value="Add the new offer!"
    onclick="websi_use_case_class.value='AddOffer';
    websi_interaction_method.value='addOffer';
    document.websi_use_case_form.submit();">
  ...
</form>
```

### 6.2.5 Communication Between the Sub-Stylesheets

Since the various IO activity classes sub-stylesheets are integrated into the use-case display sub-stylesheet, the HTML elements they generate sometimes have to communicate with the HTML elements, generated by the use-case display sub-stylesheet. In the last example of the previous section, a push button element is generated by the sub-stylesheet of Action activity class. However, the JavaScript attached to the `onclick` attribute of the HTML `input` element uses the name of the form element (websi_use_case_form), that was specified by the use-case display sub-stylesheet that created the form. So as
we demonstrate, IO activity classes sub-stylesheets can make assumptions on the HTML content, generated by a use-case displayer sub-stylesheet (e.g., the names employed, the overall HTML structure etc.) If a particular assumption is wrong (e.g., the name of the form is assumed to be websi\_UCForm and not websi\_use\_case\_form), the resulting IS will obviously not function correctly. It is the IS developer’s responsibility to ensure that the resulting HTML elements work properly together.

6.2.6 Combining the Sub-Stylesheets into an Interaction Style

As we have already mentioned, the final XSL stylesheet is not defined as an atomic component, but rather is combined from several sub-stylesheets. The reason for this design is flexibility in specifying the IS.

For example, if we are not satisfied with a certain IS, we can exchange one or more of its sub-stylesheets with alternative sub-stylesheets. But we are not forced to exchange the whole IS component. Such a partial change, however, may result in a UI that does not have a uniform appearance. Another possibility is to develop several sets of sub-stylesheets, each set employing, e.g., a certain color and design scheme, and exchanging the sub-stylesheets within a specific set only.

Now consider the case, in which a new IO activity class is added to WebSI. In that case we must provide at least one sub-stylesheet that implements the methods of this new activity class. But we do not have to develop from scratch the whole IS component.

We recommend to make each sub-stylesheet as much independent from other sub-stylesheets as possible. Each sub-stylesheet should come with its own CSS file(s) and with its own JavaScript file(s). In order to avoid name clashes between CSS classes, JavaScript function names, indirect fetcher class names etc. that come from different sub-stylesheets, it is recommended that each sub-stylesheet is given a unique name, and this name is used as a prefix in the various identifiers. The unique name may be a combination of the IO activity class name and the style name, e.g., user\_composed\_default, output\_elegant and select\_sky. The displayer sub-stylesheets may be given names such as twocolumns\_disp, dynamic\_menu\_disp, listbox\_disp etc.

6.2.7 Handling Input Values Supplied by the User

As discussed in Sec. 4.3.2, an input request may be qualified by the programmer as mandatory or optional. Mandatory inputs must be supplied by the user. Optional input requests may be ignored, and the programmer, prior to fetching a value of an optional input request, must check whether the value has actually been supplied by the user. This check is performed by the inputSpecified method of the Fetch activity class. We will now define precisely, when WebSI considers that an input value for a certain fetch ID has been supplied, and when it considers otherwise.

Assume that a fetch ID we are interested in is foo. Then:

- If the HTTP request contains the key websi\_indirect\_fetch\_foo, WebSI invokes an appropriate fetcher method, as described in Sec. 6.2.3. Now, in the following cases WebSI considers that an input value for the foo fetch ID has not been supplied:
  - The fetcher method returned null.
  - The fetcher method returned a String that contains whitespaces only, i.e., trim() ming this String yields an empty String.
The fetcher method returned a String[], whose length is zero.

The fetcher method returned a String[], each element of which is either null or a whitespace-only String, trim()ming which yields an empty String.

Otherwise, WebSI considers that an input value for the foo fetch ID has been supplied.

• If the HTTP request does not contain the key websi.indirect_fetch.foo, then in the following cases WebSI considers that an input value for the foo fetch ID has not been supplied:
  
  – The HTTP request does not contain the key foo.
  
  – The HTTP request contains the key foo, but a value for this key is a whitespace-only String, trim()ming which yields an empty String.

Otherwise, WebSI considers that an input value for the foo fetch ID has been supplied.

The above definitions imply several rules on IS implementors:

• Strings, which arrive in an HTTP request and are empty or contain only whitespaces, are not treated by WebSI as legal input values.

• A fetcher method should not return a String[], whose length is not zero and some (but not all) of its Strings are nulls or contain only whitespaces. Such a String[] is not considered as “empty” by WebSI, but null or whitespace-only Strings may cause problems for the methods of the Fetch activity class, trying to convert such String to a desired type.

To conclude, WebSI takes for legal input values only non-empty strings, which contain at least one non-whitespace character and are trim()ed. If an input value comes directly from an HTTP request, WebSI takes care to trim() it. If an input value comes from a fetcher method, it must already be trim()med.

The various methods of the Fetch activity class deliver the input values to the programmer, such that the following holds:

• The methods that return an object (not a primitive datatype) never return null.

• The methods that return an array, never return a null or a zero-length array.

• If the returned array contains objects (not datatypes), none of its elements is null.

• All Strings returned by the methods are non-empty and trim()med.

From our experience, this simplifies the client programmer’s job, not having to check such subtle cases as null Strings, empty Strings, whitespace-only Strings, null or zero-length arrays, null elements in an array and so on.

The most important methods of the Fetch activity class are Fetch.string and Fetch.strings methods. (The reason is that WebSI receives HTTP parameter values as strings from the underlying J2EE layer.) When invoked with a fetch ID foo, these methods first check whether a key websi.indirect_fetch.foo is present in the request. If yes, the value for that key indicates the indirect fetcher method to be invoked. This value is parsed and the fetcher method is invoked via Java reflection API, using the services of
All the other methods of the Fetch activity class first invoke the Fetch.string or the Fetch.strings method, and then try to convert the resulting string value to the desired type. The conversion is performed using the static string conversion methods presented in Tab. 6.2. In case the conversion fails, a custom WebSI exception, named websi.exceptions.NonFatalUseCaseExecutionException is thrown. The exception object holds an error message, including the value(s) that failed to undergo the conversion. The NonFatalUseCaseExecutionException is specifically monitored by the execute method of the UseCase class, as discussed in Sec. 6.5.2.

### 6.3 The DBProxy Bean

As described in Sec. 4.4, an interaction method uses read and write methods of the DB activity class, in order to manipulate the database. The same methods may be also used by user-defined session objects, whose classes are derived from the websi.SessionObject class.

**WebSI** accesses the database using Enterprise Java Beans (EJB). An EJB is a portable J2EE server-side component, that encapsulates a part of the application’s business logic and provides a clean interface, through which the business logic is executed. An EJB is managed by an EJB container, that is a part of a J2EE server. The EJB container provides system-level services to the bean, so that the bean’s code can concentrate on the business logic.

**WebSI** defines a single kind of EJB that is called a “DBProxy” bean. This bean hides the complexity of opening and closing database connections, managing database transactions and other database resources. The DBProxy bean provides a clean interface, through which the database can be conveniently manipulated. The actual location of the database is specified during the deployment of a **WebSI** application on a J2EE server. The DBProxy bean obtains a database connection using the deployment settings.

DBProxy is a *stateful session bean*. Other kinds of EJBs are stateless session beans, entity beans and message-driven beans. A session bean is intended to provide services to a single client of a J2EE application. A session bean is not persistent, i.e., its internal state is not saved to a database. When a client that uses the bean terminates, the session bean becomes not associated with the client anymore, the bean’s state is lost, and the bean
can be reused by some other client or garbage-collected. A stateful session bean holds its state between the calls to the bean’s methods, while a stateless session bean loses its state between method calls.

Each user of a WebSEI application is associated with a separate DBProxy bean that is stored in an HttpSession object. Each DBProxy opens and manages a separate connection to the database. In this way, we achieve what is called a “true database concurrency”:

- A J2EE server handles several HTTP requests concurrently (although, at most one request from a certain user is handled at a time, as described in Sec. 6.1.8).
- Each request is associated with a certain user, i.e., with a specific session.
- Each request is handled in a separate thread.
- When a use-case object (or a session object) that belongs to a certain session accesses the database, it does this through its separate DBProxy bean, i.e., through a separate database connection.

As a result, the database is accessed concurrently by several connections, each connection being opened on behalf of a different user. Figure 6.11 illustrates the above.

![Figure 6.11: Accessing the database via DBProxy beans](image)

### 6.3.1 The Remote and the Local Interface

A client may access the services of a session bean only through the methods defined in the bean’s interfaces. These interfaces define the client’s view of the bean. All other aspects
of the bean, e.g., method implementations, are hidden from the client. A session bean may define a remote interface, a local interface, or both. When using a remote interface, the bean’s client may run on a different Java Virtual Machine (JVM), than the bean it accesses. In that case, the bean’s location is transparent to the client. When using a local interface, the bean’s client must run on the same JVM, as the bean it accesses, and the bean’s location is not transparent to the client [25].

The DBProxy bean defines both a local and a remote interface. During the deployment, it is specified, whether the bean should be accessed through the local or the remote interface (see Sec. A.6.1). The methods defined in both interfaces are very similar, except a slight difference in the signatures. The following is the definition of DBProxy’s remote interface:

```java
public interface DBProxy extends EJBObject {
    public Table read(String sql_query) throws RemoteException;
    public int write(String sql_query) throws RemoteException, DBConstraintViolationException;
    public void commitTransaction() throws RemoteException, TransactionRolledBackException;
    public void rollbackTransaction() throws RemoteException;
}
```

Here is the definition of DBProxy’s local interface:

```java
public interface DBProxyLocal extends EJBLocalObject {
    public Table read(String sql_query);
    public int write(String sql_query) throws DBConstraintViolationException;
    public void commitTransaction() throws TransactionRolledBackException;
    public void rollbackTransaction();
}
```

The only difference is that the methods of the remote interface throw a RemoteException. This is required, since these methods may be invoked remotely, i.e., over a network by a remote client. We will now briefly describe the methods of the two interfaces:

- The read and the write methods are the same as of the DB activity class, and were discussed in Sec. 4.4.
- The commitTransaction method is used to commit the ongoing transaction, when an interaction method ends successfully (see Sec. 6.5.2). A new transaction is opened on-demand by DBProxy, when a read or a write method is invoked, and the DBProxy is not associated with a transaction at that time.
- The rollbackTransaction method is used to roll back the ongoing transaction (if any), in case the client programmer asks to do so, or in exceptional situations.

### 6.3.2 The Home and the Local Home Interface

The home and the local home interfaces of a bean are used by a client in order to obtain a reference to a bean. The home interface is used for beans that are accessed remotely, while the local home interface is used for beans that are accessed locally. The following is the definition of DBProxy’s home interface:

```java
public interface DBProxyHome extends EJBHome {
    DBProxy create(DBVendorSpecificSupport support)
}
```
Here is the definition of DBProxy’s local home interface:

```java
public interface DBProxyLocalHome extends EJBLocalHome {
    DBProxyLocal create(DBVendorSpecificSupport support)
        throws CreateException;

    public static final String LOOKUP_NAME =
        "java:comp/env/ejb/DBProxyLocal";
}
```

The `create` method obtains a reference to the DBProxy bean. The type of a reference is `websi.db.DBProxy` in case of a remote access, or `websi.db.DBProxyLocal` in case of a local access. The argument to the `create` method will be discussed in Sec. 6.3.5. The `LOOKUP_NAME` field stores a name that is used in order to obtain a reference to a home interface or to a local home interface. This name is mapped during deployment to a JNDI name of the DBProxy bean in case of a remote access, or to an “Enterprise Bean Name” in case of a local access (see Sec. A.8.2 and [25]).

### 6.3.3 The `DBProxyBean` Class

The `websi.db.DBProxyBean` class implements the services of the DBProxy bean. It implements the methods defined in the `DBProxy` and `DBProxyLocal` interfaces. It implements the `ejbCreate` method, which is invoked by an EJB container, when a client invokes the `create` method of the `DBProxyHome` or `DBProxyLocalHome` Interface. In addition, the `DBProxyBean` class implements the methods, required by EJB technology, such as `ejbActivate` and `ejbRemove`. For more details, the reader is advised to browse the code of the `DBProxyBean` class.

### 6.3.4 The DB Activity Class

The DB activity class (Fig. 6.12) provides a set of methods that manage the DBProxy bean’s life-cycle.

The three public methods (read, write and rollbackTransaction) are accessed by the client programmer, during an interaction method execution, as described in Sec. 4.4. The package access methods (existsDBProxy, isDBProxyRemote, removeDBProxy and commitTransaction) are used by the DB class as well as by other WebSI components, such as the UseCase class. The private methods (getDBProxyLocal and getDBProxy) are used by the DB class only. We will briefly describe the package access and the private methods of DB activity class.

- The `existsDBProxy` method checks, whether the HttpSession, on behalf of which the interaction method is executing, currently stores a reference to the DBProxy bean remote or local interface.

- The `isDBProxyRemote` method checks, whether the DBProxy bean in the current WebSI application uses the remote interface or the local interface (according to the deployment settings).

- The `getDBProxy` and `getDBProxyLocal` methods obtain a reference to the DBProxy bean, i.e., to its remote or local interface accordingly. (If, for example, the `getDBProxy`
method is invoked, but the current application uses a local access DBProxy, a runtime exception is thrown.) Both methods instantiate the bean on-demand and store it in the HttpSession.

- The removeDBProxy method destroys the DBProxy bean, bound to the HttpSession, on behalf of which the interaction method is executing. In general, the same DBProxy can serve the user as long as the session is valid, but there may occur various exceptional situations, in which it is needed to destroy the bean. In these situations this method is used. A new bean will be created on-demand by the getDBProxy or getDBProxyLocal method.

- The commitTransaction method accesses the DBProxy bean, bound to the session, on behalf of which the interaction method is executing, and commits the bean’s ongoing transaction if any. This method is used by WebSI to commit the transaction, after the interaction method has finished executing.

### 6.3.5 The DBVendorSpecificSupport Class

A J2EE technology is designed in such a way, that its components can access external resources, such as databases, in a platform-independent manner. In other words, the code of a J2EE component, such as an EJB, need not be altered or recompiled if, for example, the developers choose to employ a database engine of a different vendor. The DBProxy bean is also a platform-independent J2EE component, it can work with any database that provides a J2EE connector and supports such features as concurrent transactions and scrollable java.sql.ResultSets [26].

In some cases, however, a vendor-specific information is required. Consider the WebSI feature of monitoring various database constraints, described in Sec. 4.4.2. In case a database constraint, such as PRIMARY KEY, is violated, the database engine raises...
an error. This error is conveyed to the code of the `write` method of the `DBProxyBean` class in the form of a `java.sql.SQLException`. However, an `SQLException` can originate from several causes, such as incorrect SQL syntax, referring to a table that does not exist etc. The actual cause of a certain `SQLException` can be inspected via the `getSQLState` method of the `SQLException` class. This method returns a `String`, that holds the code of the occurred error. The problem is, that error codes differ from vendor to vendor.

`WebSI` provides a general mechanism for the `DBProxy` bean to handle situations, in which a vendor-specific information is required. `WebSI` defines an interface, named `websi.db.DBVendorSpecificSupport`. This interface defines methods that provide vendor-specific information to the `DBProxyBean` class. For each database product to be used with `WebSI`, an implementation of this interface should be supplied to `WebSI`. This is performed during the deployment, by providing a fully-qualified name of a class that implements the `DBVendorSpecificSupport` interface (see Sec. A.6.1). When an instance of a `DBProxy` bean is created, via the `create` method of a `DBProxyHome` or a `DBProxyLocalHome` interface, `WebSI` passes the instance of the vendor-specific support class as a parameter to the `create` method. In such a way, the `DBProxyBean` class can obtain vendor-specific information, when required.

If no implementation of the `DBVendorSpecificSupport` interface is provided, `WebSI` instantiates the `DBProxy` bean with a `null` argument. In that case, the `DBProxyBean` class will not provide vendor-specific services. For example, database constraints violations will not be delivered to the client programmer as constraint-specific exceptions; instead they will be treated as general database errors, which will result in aborting the interaction method execution, followed by the ongoing transaction rollback.

Currently, the only vendor-specific feature in `WebSI` is the monitoring of database constraint violation exceptions. Therefore, currently, there is a single method in the `DBVendorSpecificSupport` interface:

```java
public DBConstraintViolationException constraintViolated(SQLException exc);
```

This method should return a `DBConstraintViolationException` object or an object of one of its constraint-specific subclasses (see Sec. 4.4), in case the `SQLException` passed to it originates from a database constraint violation. Otherwise, this method should return `null`.

`WebSI` provides a single implementation of the `DBVendorSpecificSupport` interface, for the IBM's Cloudscape 4.0.6 database. The implementation class is named `websi.db.Cloudscape406`.

### 6.4 The Table Interface and its Implementations

In Sec. 4.4.1 we have discussed the `websi.db.Table` interface that represents a tabular object, having columns and rows.

Since a table is one of the main `WebSI` abstractions, the `Table` interface may potentially have many implementations. Therefore, we strive to provide a convenient way of adding new `Table` implementations to `WebSI`. For that purpose, we employ the Adapter design pattern [16]. We will give now some background on this pattern.

The Adapter design pattern (see Fig. 6.13) can be applied, when a class `C` is required to conform to an interface `I`. In that case, an adapter class `C2I` should be implemented, that, in particular, conforms to the `I` interface. The constructor of the `C2I` class usually receives a `C` object as an argument. The `C2I` class implements the methods of the `I`
interface according to the functionality of the $C$ class. Adapting an object of the $C$ class to the $I$ interface may look like this:

\[
\ldots \\
C\toAdapt = \ldots ;\\nI\ adaptedC = new\ C2I(\toAdapt) ;\\n\ldots
\]

Implementing all the methods of the $I$ interface in the code of the $C2I$ class can sometimes be labor-intensive, especially if the $I$ interface contains many methods. To alleviate this problem, the Adapter design pattern can be slightly modified, by defining an abstract adapter class $\text{AbstractC2I}$. This is an abstract class, that implements some or all methods of the $I$ interface, and leaves a relatively small amount of methods that should be implemented by concrete adapter classes, derived from it (see Fig. 6.14). Adapting an

Figure 6.13: The Adapter design pattern

Figure 6.14: The Adapter design pattern with an abstract adapter class
object of the C class to the I interface, in that case may look like this:

```java
public class C2I extends AbstractC2I {
    public void toBeImplemented1() {
        // Implementation
        ...
    }

    public void toBeImplemented2() {
        // Implementation
        ...
    }
    ...
}

C toAdapt = ... ;
I adaptedC = new C2I(toAdapt);
...
```

For the Table interface WebSI defines such an abstract adapter class, which is named websi.AbstractTableAdapter. This class implements the methods of the Table interface, except for the following five methods:

```java
// How many columns the Table has? (should be >= 1)
public int getColumnsNr();

// How many rows the Table has? (should be >= 0)
public int getRowsNr();

// What is the name of a column, whose number is 'column'?  
// Columns are numbered starting from 1
public String getColumnName(int column);

// Column label may potentially differ from the column name
// Labels are used in printouts
public String getColumnLabel(int column);

// What is the value of the Table in the ('row', 'column') cell?  
// Rows and columns are numbered starting from 1
public Object getObject(int row, int column);
```

These five methods are left abstract and they should be implemented by derived classes that wish to conform to the Table interface. The remaining methods of the Table interface are the various getXXX methods, such as getString and getTime. These methods are implemented in AbstractTableAdapter by delegating to the getObject method, and trying to convert the resulting Object to the desired class. The conversion employs appropriate static methods of target classes, for example:

```java
public int getInt(int row, int column) {
    // Obtain the value by indirectly delegating to getObject
    Object value = ... ;
    // NULL values treated as 0
    if (value == null)
        return 0;

    if (value instanceof Integer)
        return ((Integer)value).intValue();

    // Try to convert the string representation
    // If this fails, a runtime exception is thrown
```
\begin{verbatim}
  return Integer.parseInt(value.toString());
}
\end{verbatim}

In addition, the `AbstractTableAdapter` class implements the equality testing method:

\begin{verbatim}
  public boolean equals(Object otherObj);
\end{verbatim}

This method considers two `AbstractTableAdapter`s to be equal as described in Sec. 4.4.1:

- they have the same number of rows and columns and
  - the objects in their corresponding cells are equal, when applying the `equals` method on them, or
  - the string representations of these objects (obtained by applying `toString`) are identical

Currently, all `WebSI` `Table`s are derived from `AbstractTableAdapter`. Therefore, any two `Table`s can be tested for equality in a consistent way. For the future, we also recommend to derive the various `Table` implementations from `AbstractTableAdapter`, in order to preserve this property.

### 6.4.1 The StaticCopyResultSet2TableAdapter Adapter

One of the frequent `WebSI` operations is to present the results of an SQL SELECT query as a `Table`. SQL SELECT queries are performed in the code of the `read` method of the `DBProxyBean` class. The implementation of this method employs JDBC [26] facilities, in which the result of a SELECT query is returned as a `java.sql.ResultSet`. In order to present the `ResultSet` as a `Table`, `WebSI` defines a concrete adapter class, named `websi.db.StaticCopyResultSet2TableAdapter` (see Fig. 6.15). This class is derived from the `AbstractTableAdapter` class and implements the five methods mentioned above. The constructor of this class receives a `ResultSet` as an argument. The contents of the passed `ResultSet` are copied and statically stored inside the `StaticCopyResultSet2TableAdapter` class.

Since the `DBProxy` bean may be accessed remotely, the objects returned by its methods (and those received) must be serializable. (A serializable object can be converted to a stream, which can be passed over a network.) In particular, `Table`s returned by the `read` method of the `DBProxy` bean must hold only serializable data.

Neither `Table` nor `AbstractTableAdapter` are defined as `java.io.Serializable`, since some of their implementations may potentially be non-serializable. The `StaticCopyResultSet2TableAdapter` class is, however, defined as `Serializable`, because when copying data from a `ResultSet`, each `Object` stored in the `ResultSet` is assumed to be serializable. This assumption is correct for Java classes that are equivalents of basic SQL datatypes: `Integer`, `String`, `java.sql.Time`, `java.sql.Date` etc. So in the constructor of `StaticCopyResultSet2TableAdapter`, each `Object` stored in a `ResultSet` is casted to `Serializable` and stored in the `StaticCopyResultSet2TableAdapter`.

To conclude, the `StaticCopyResultSet2TableAdapter` class is an immutable serializable implementation of the `Table` interface. The contents of a `StaticCopyResultSet2TableAdapter` are static and independent of the database.
6.4.2 The TableImpl Class

We have already discussed the services of the TableImpl class in Sec. 4.9. This class is also derived from the AbstractTableAdapter and represents a mutable Table that typically is used in user-defined session objects. The TableImpl class is not defined as Serializable, since the programmer may store any Object in it, using the following method:

```
public void setTableValue(int row, int col, Object value);
```

6.5 The UseCase Class

The UseCase class resides in the websi package. This is an abstract class that serves as a base class for all use-case classes. The UseCase class encapsulates the following functionalities:

- Defining an abstract start interaction method, which must be overridden by a use-case class.
- Providing the methods that support inclusion, returning from inclusion, obtaining inclusion parameters and results.
- Providing the functionality to execute an interaction method.

6.5.1 Handling Inclusions and Returns from Inclusions

The UseCase class provides methods and data members that implement the "<<include>>" relationship between use cases, as described in Sec. 4.6. Figure 6.16 shows these methods.
Table 6.3: Dividing the inclusion-related data members into four categories

<table>
<thead>
<tr>
<th>Category Number</th>
<th>Category Description</th>
<th>Data members</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Used by the base use case, when inclusion starts</td>
<td>_inclusion_use_case, _inclusion_param_to_pass, _return_to_method_name_to_pass</td>
</tr>
<tr>
<td>2</td>
<td>Used by the inclusion use case, when the inclusion starts</td>
<td>_base_use_case, _inclusion_param_to_get, _return_to_method_name_to_get</td>
</tr>
<tr>
<td>3</td>
<td>Used by the inclusion use case, when the inclusion ends</td>
<td>_inclusion_result_to_pass, _end_inclusion</td>
</tr>
<tr>
<td>4</td>
<td>Used by the base use case, when it receives control back from the inclusion use case</td>
<td>_inclusion_result_to_get</td>
</tr>
</tbody>
</table>

and data members.

```java
UseCase
(from websi)

_inclusion_use_case:UseCase
_inclusion_param_to_pass:Object
_return_to_method_name_to_pass:String
_base_use_case:UseCase
_inclusion_param_to_get:Object
_return_to_method_name_to_get:String
_inclusion_result_to_pass:String
_end_inclusion:boolean
_inclusion_result_to_get:Object
```

Figure 6.16: The members of the UseCase class, related to the inclusion mechanism

The data members, presented on Fig. 6.16, can be divided into four categories, as shown on Tab. 6.3. We will now describe how these data members are used during an inclusion, followed by returning from inclusion.

When a base use case, during an interaction method execution, invokes the includeUseCase method, this method sets the three data members of the first category of the base use case (which is the current use-case object):

- _inclusion_use_case—the use-case object to be included
- _inclusion_param_to_pass—the inclusion parameter to pass to the inclusion use case (may be null)
Prior to setting the three data members, the includeUseCase method accesses the ApplicationUseCasesSetup object stored in the ServletContext, and obtains the names of the security roles, whose users are authorized to execute the inclusion use case. Then the method checks, whether the user is logged-in for an appropriate security role. If yes, the method proceeds to setting the values of the three data members. Otherwise, the method throws a NonFatalUseCaseExecutionException. This exception is specifically monitored by the execute method of the UseCase class, during the execution of an interaction method. The exception object holds an error message, which will be presented to the user.

The described security check is essential, since this is the only case, in which a use case execution is initiated not by the use-case displayer.

After the includeUseCase method successfully finishes, the inclusion use case member is non-null. When the interaction method execution ends, WebSI checks this data member, and, since it is non-null, infers that an inclusion must happen. Then WebSI sets the data members of the second category of the inclusion use-case object:

- _base_use_case—the base use case, which included the inclusion use case
- _inclusion_param_to_get—the inclusion parameter passed by the base use case to the inclusion use case (may be null)
- _return_to_method_name_to_get—the name of the interaction method of the base use case, which will be executed when the inclusion ends

Now, the start interaction method of the inclusion use case is executed.

The inclusion use case, at some point in time, may invoke the getInclusionParam method, which will return the value of its _inclusion_param_to_get data member, i.e., the parameter that the base use case has passed. Then, at some later point in time, the inclusion use case may invoke the endInclusion method. This invocation sets the data members of the third category of the inclusion use case:

- _inclusion_result_to_pass—the inclusion result, which the inclusion use case wishes to pass to the base use case (may be null)
- _end_inclusion—is set to true

Setting the _end_inclusion data member to true tells WebSI that the inclusion use case wishes to end the inclusion. Then WebSI uses the data members of the second category of the inclusion use case in order to determine, which use case is the base use case, and which its interaction method has to be executed. WebSI also sets the data member of the fourth category of the base use case:

- _inclusion_result_to_get—the inclusion result passed to the base use case by the inclusion use case (may be null)

Now WebSI invokes the required interaction method of the base use case, thus returning control back to it. The base use case, at some point in time, may invoke the getInclusionResult method, which will return the value of its _inclusion_result_to_get data member.

By properly nullifying the appropriate inclusion-related data members, WebSI supports complex inclusion chains, without circles.
6.5.2 Executing an Interaction Method

The `execute` method of the `UseCase` class is, probably, the heart of the `WebSI` system. This method is invoked by the use-case displayer component on the required use-case object in order to execute an interaction method of this object. In addition, this method is invoked by a base use case on an inclusion use-case object, when the inclusion begins. This method is also invoked by the inclusion use case on the base use-case object, when the inclusion ends.

The use-case object is an object of a class, derived from the `UseCase` class. Therefore, the static type of this object is `UseCase`. The dynamic type of this object is not known to `WebSI` at compile time, because the use-classes are implemented by the client programmer, and their fully-qualified class names are supplied to `WebSI` at deployment time (see Sec. A.6.2).

The signature of the `execute` method is:

```java
void execute(Document doc, Node root, HttpServletRequest request, String interactionMethodToInvoke);
```

The first three parameters are required for activity classes, as discussed in Sec. 6.1.7:

- `Document` represents the XML response content. This object provides factory methods, which enable the IO activity classes to create XML content, such as XML elements and XML text nodes.
- `Node` represents an XML element, under which the IO activity classes put their generated XML content.
- `HttpServletRequest` encapsulates the various HTTP request data.

The fourth parameter indicates the name of the interaction method, which must be executed. This method belongs to the current use-case class, i.e., the dynamic class of the object, on which the `execute` method has been invoked. It is sufficient to supply only the name of the interaction method, since all interaction methods share the same signature:

```java
public void methodName() throws Exception;
```

We will now describe the flow of events of the `execute` method.

1. Prior to all, the first three arguments, i.e., `Document`, `Node` and `HttpServletRequest`, are stored in the data members of the current use-case object.

2. The current use-case object is associated with a thread, which is running the `execute` method, as described in Sec. 6.1.7.

3. The interaction method, whose name has been passed, is invoked on the current use-case object. This is performed via Java reflection API, using the services of `WebSI ReflectionTools` class.

4. The interaction method execution may throw exceptions, originating from several causes, such as:

   - An interaction method with the specified name does not exist.
   - The database connection could not be established.
   - An SQL query syntax is incorrect.
   - An SQL query violated a database constraint, and the violation exception has not been caught by the programmer (see Sec. 4.4.4).
• The programmer tried to include a use case, but the user was not logged in for an appropriate security role, as discussed earlier in Sec. 6.5.1.

• The programmer tried to fetch an input value, for a non-existing fetch ID.

• The programmer tried to fetch an input value of an optional input request, for which the value has not been supplied.

• The programmer tried to fetch an input value, and the conversion to the desired type failed. For example, the programmer has issued:

\[ \text{int } x = \text{Fetch.Int}(\ldots); \]

but the value supplied by the user cannot be converted to \text{int}.

• The programmer’s code caused some general Java error, such as dereferencing a \text{null} reference, or accessing an array with an incorrect index.

\textit{WebSI} divides all the errors, which may occur during an interaction method execution, into \textit{fatal} and \textit{non-fatal}. A non-fatal error is recoverable. After a non-fatal error occurs, \textit{WebSI} presents an error message to the user, and the user may continue to operate the system. Fatal error originates from a bug in the programmer’s code, or from a software or hardware failure outside \textit{WebSI} responsibilities. It is not clear whether a fatal error is recoverable.

At each point within the \texttt{execute} method, \textit{WebSI} remembers whether any fatal and any non-fatal errors have occurred. At this point of the \texttt{execute} method, a non-fatal error may originate from two causes:

• security check failure in the \texttt{includeUseCase} method

• failure to fetch an input value, converted to a datatype requested by the programmer

Any non-fatal error at this point is indicated by a \texttt{NonFatalUseCaseExecutionException}, which is specifically monitored. This exception objects contains the system error message describing what happened. Throwing a \texttt{NonFatalUseCaseExecutionException} has been described in Sec. 6.2.7 and 6.5.1.

In case a \texttt{NonFatalUseCaseExecutionException} is caught, \textit{WebSI} cleans all the XML content generated by the interaction method execution, and generates a new content with an error message, which suggests what the user should do, e.g., try to submit the request again. In case of a fatal error, \textit{WebSI} does the same, except that the error message contains the stack trace. Such an error message is intended to be used for debugging.

5. Now \textit{WebSI} handles the ongoing database transaction (if any):

• If any fatal or non-fatal errors occurred, \textit{WebSI} rolls the ongoing transaction back. In case a rollback fails, \textit{WebSI} considers it as a fatal error, and generates XML error message content, including the stack trace.

• Otherwise, no errors have occurred until now, and \textit{WebSI} tries to commit the ongoing transaction, using the \texttt{commitTransaction} method of the \texttt{DB} activity class. Committing a transaction may fail with an exception:
A `websi.exceptions.TransactionRolledBackException` indicates that the transaction failed to commit and has been rolled back. WebSI considers it as a non-fatal error. WebSI clears all the XML content generated so far and presents a message asking the user to execute the use case again.

Any other exception is considered a fatal error. WebSI generates an error message, including the stack trace.

6. If any fatal errors have occurred until now, WebSI destroys the DBProxy bean of the current session, using the `removeDBProxy` method of the DB activity class. Then WebSI removes the association between the current use-case object and the current thread, and returns from the `execute` method.

   We remove the DBProxy bean here, since many of the fatal errors are database errors, such as incorrect SQL syntax. When such a severe database error occurs, the DBProxy bean is destroyed automatically by the J2EE server. The DBProxy or the DBProxyLocal reference, stored in the HttpSession, however, still exists (although any attempt to call its methods results in exception). Therefore, we find it safer in case of a severe error (even if not a database error), to destroy the existing DBProxy bean (if not destroyed yet by the J2EE server) and to remove the DBProxy or the DBProxyLocal reference from the session. New DBProxy for the current session will be created on-demand.

7. If any non-fatal errors have occurred until now, WebSI removes the use case-thread association and returns from the `execute` method. The DBProxy bean is not removed in this case.

8. If no errors occurred, WebSI removes the use case-thread association and moves on to check whether an inclusion use case should be included, or the current use case wants to return control back to a base use case. Handling inclusions and returns from inclusions employs the data members of the `UseCase` class, discussed in Sec. 6.5.1.

9. First of all, WebSI performs a sanity check: the case, in which the programmer indicated both an inclusion to start and to end, like this:

   ```java
   public void someInteractionMethod() throws Exception {
       ...
       includeUseCase(...);
       ...
       endInclusion(...);
       ...
   }
   ``

   The sanity check is:
   ```java
   if (_inclusion_use_case != null && _end_inclusion)
       throw new WebSIException("...");
   ```

10. Now WebSI handles a case, in which the current use case starts an inclusion of another use case:

    ```java
    if (_inclusion_use_case != null) {
    ```
// Set the inclusion use case category 2 data members,
// using this use case category 1 data members
_inclusion_use_case._base_use_case = this;
_inclusion_use_case._inclusion_param_to_get =
  _inclusion_param_to_pass;
_inclusion_use_case._return_to_method_name_to_get =
  _return_to_method_name_to_pass;

// Nullify this use case category 1 data members,
// but first remember the inclusion use case and its class name
UseCase inclusion_use_case = _inclusion_use_case;
String inclusion_uc_class_name =
  inclusion_use_case.getClass().getName();
_inclusion_use_case = null;
_inclusion_param_to_pass = null;
_return_to_method_name_to_pass = null;

Debug.print(
  "This is use case " + this +
  ", starting inclusion of use case " + inclusion_use_case);

// Finally, call ‘execute’ on an inclusion use case
inclusion_use_case.execute(_document, _root, _request,
  USE_CASE_STARTING_METHOD_NAME);

return;
}

Managing the inclusion-related data members has been discussed in Sec. 6.5.1. Note that the call to the execute method of the inclusion use case passes the request-related parameters of the current use case, stored in its data members. The data members _document, _root and _request are the Document, Node and HttpServletRequest objects respectively, passed to the current use-case object’s execute method. Now if, for example, the current use case has generated some UI content (which is stored under the _root Node), and the inclusion use case’s_start method also generates some UI content (which will also be stored under _root), the user will see both contents in the response. In addition, the inclusion use case can access the HTTP request parameters, passed to the current use case.

Note also, that WebST has committed the ongoing transaction prior to including the inclusion use case. Therefore, the inclusion use case’s_start method will open its separate transaction.
11. Finally, WebSI manages a return from inclusion:

```java
if (_end_inclusion) {
...
    return;
}
```

- First WebSI checks whether a control should be passed to a base use case, i.e., whether a base use case specified that its interaction method should be executed, when the inclusion ends. The case, in which the control should not be returned:

```java
// Do we have a return-to method? If no - just do a cleanup and return
if (_return_to_method_name_to_get == null) {
    // Nullify my category 2 and 3 data members
    _base_use_case = null;
    _inclusion_param_to_get = null;
    _return_to_method_name_to_get = null; // This is already null

    _inclusion_result_to_pass = null;
    _end_inclusion = false;

    Debug.print("This is use case "+ this + ", returning from inclusion, no return-to method.");

    return;
}
```

- If the control has to be returned to the base use case, WebSI performs this similarly to starting an inclusion:

```java
// Set the base use case category 4 data members
_base_use_case._inclusion_result_to_get = _inclusion_result_to_pass;

// Nullify this use case category 2 and 3 data members,
// but first remember the base use case and its return-to method
UseCase base_use_case = _base_use_case;
String return_to_method_name = _return_to_method_name_to_get;
_base_use_case = null;
_inclusion_param_to_get = null;
_return_to_method_name_to_get = null;

_inclusion_result_to_pass = null;
_end_inclusion = false;

Debug.print("This is use case "+ this + ", returning from inclusion, return-to method: "+
    return_to_method_name);
```
/ Finally call 'execute' on the base use case
base.use_case.execute(document, root, request,
    return_to_method_name);
Chapter 7

Evaluation

7.1 Theoretical Evaluation

In this section we perform a theoretical evaluation of WebSI, according to the goals we set in Chap. 1.

The WebSI tool is based on a combination of three techniques that have proven to be useful in manufacturing of complex software systems:

**High-level declarative notations** The WebSI API is declarative, rather than operative. Such methods as Select.one, UserComposed.timeOptional and DB.read, enable the programmer to specify what is the desired system’s functionality, but not which specific technicalities should be used to implement this functionality. In addition, WebSI employs the declarative SQL notation to manipulate database data.

**Frameworks** WebSI is an application framework. It comes with a set of ready-made, reusable software components, and a specific methodology of combining these components together.

Some of the ready-made, reusable WebSI components are: the UseCase class, the various activity classes and the available interaction styles (IS). WebSI methodology of combining the components together includes defining a separate class for each use case, derived from the UseCase class. Then the programmer must code the start interaction method, which implements the first input-output (IO) sequence of the use case. The other IO sequences of the use case will be coded in additional interaction methods. Throughout the code of an interaction method, the programmer may employ the services of WebSI activity classes, such as UserComposed, Output, Select, DB and so on.

**Software development processes** WebSI is intended to be used for the construction stage of a software process. WebSI assumes the availability of a use-case specification and of the database schema.

In Sec. 2.2 we introduced the concept of use case-oriented framework and set several design goals of such a framework. Let us investigate to which extent WebSI supports the set design goals.

**The programmer’s code is around IO and data processing** To find out which kinds of code a WebSI programmer writes, let us check, which WebSI services are accessible to programmer and what is the nature of these services.
First of all, there are the IO activity classes: Output, UserComposed, Select and Action. The programmer employs the services of these classes in order to request inputs from the user and to present outputs to the user. These are clear IO activities.

The read and write methods of the DB activity class are employed for database data manipulation. The DB.rollbackTransaction method is employed to undo all the SQL queries in a current transaction. In principle, in some cases it may be possible to manually perform the “undo”s, by applying appropriate INSERT, UPDATE and DELETE queries via the DB.write method. Obviously, the DB.rollbackTransaction method makes it much simpler to undo the queries. Therefore, this method may also be considered as a data manipulation method.

The methods of the Fetch activity class are clearly around IO.

There are two activity classes left, which are the UserID and the Session classes. They both address the multi-user nature of WebSI. The UserID class enables the programmer to obtain identification details of the user that is operating the use case. The Session class allows to manage temporary data for a certain user, so that this data is accessible from all the use-case classes. Therefore, the services of these classes are not around IO or data manipulation.

**Construction and manipulation of the user interface are hidden** The WebSI programmer communicates with the user through a declarative API of the IO activity classes and of the Fetch activity class. The API does not mention specific user interface (UI) controls, sizes, colors and layout. Therefore, the construction and manipulation of the UI is hidden from the programmer. As a result, switching to an alternative IS, does not require the WebSI-based code to be changed or even recompiled.

**Database access and transactions details are hidden** The details of database access in WebSI are hidden from the programmer. Opening and closing database connections, and managing database resources are performed automatically by WebSI. The programmer just employs the DB.read and DB.write methods to manipulate the database.

As for transactions, technical details of opening, committing and rolling a transaction back are all taken care of by WebSI. Transaction demarcation, however, is not completely hidden:

- The programmer must be familiar with the WebSI transactions model. That is, she must know that a first SQL query of an interaction method implicitly opens a new transaction, and that after the interaction method ends, the ongoing transaction will be committed.

- In some cases, the programmer uses the DB.rollbackTransaction method in order to explicitly roll the ongoing transaction back. In that case, a new transaction will be opened, when an additional SQL query is issued.

**Managing the multiple system users is hidden** The WebSI programmer designs her use-case classes mostly as if a single user is going to operate the system. That is, the programmer does not consider such issues as threads management, synchronization, allocating system resources for each new user, freeing resources when a certain user terminates and so on. All these issues are performed automatically by WebSI.

At any time, the programmer can obtain identification details (the username and the security role) of a user that is operating a certain use case. These details, however, are
not obtained for such technical purposes as threads management or synchronization. The username is mostly used for manipulating database data that is related to a certain user. The security role is used for security purposes (as described below).

The only activity, in which the programmer must be aware of a multi-user nature of WebST, is manipulating the database. Between two separate transactions, i.e., two different IO sequences, the programmer cannot assume that the database state does not change.

**Security-related details are hidden** In many cases, WebST-based code does not address security at all. When coding a use case, the programmer may assume that this use case will be accessed only by authorized users. If, in order to implement a certain use case, user’s identification details (i.e., the username) are required, the programmer may assume that WebST will enforce the user to identify herself correctly prior to executing the use case. The programmer just employs the UserID.getUsername method to obtain the user’s identification.

The security settings of a WebST-based application are defined at deployment time. All the use cases that require identification of the user must be defined as non-public, i.e., associated with one or more security roles. Use cases that do not require user identification, but still should not be executed by unauthorized users, must also be defined as non-public. For each security role, it is defined where in the database its usernames and passwords are stored.

In some cases, however, the programmer’s code still explicitly addresses security-related issues. Consider, for example, a use case, which is executed differently by users related to different security roles. For such use case, the programmer must in her code obtain the security role of the operating user and act accordingly.

**Environment-specific details are hidden** WebST-based code is free from environment-specific details. The code does not mention, for example, which operating system is employed for the application server, on which hardware it runs and which communication protocol is used between the multiple users. The result is that when there is a change in the environment, the WebST-based code does not have to be changed.

The DBVendorSpecificSupport interface (and its implementations) provide information to WebST, which is specific for a certain database vendor. The programmer, however, is not aware of this interface, and no vendor-specific database details appear in the programmer’s code. At deployment time, it is required to provide an implementation of DBVendorSpecificSupport interface, according to the employed database product. This implementation is not application-specific, it is coded once, and can be reused for many applications that employ the same database product.

**Application-specific code vs. common code** It seems that a WebST programmer writes only code, which is specific to the information system she is developing. We are not aware of a common activity, which must be re-implemented or re-designed by the programmer for different applications. Moreover, the reason some of the details discussed above are not completely hidden by WebST, is that these details are application-specific. For example, the DB.rollbackTransaction method is used by the programmer according to the needs of a specific application. After the programmer issues several SQL queries, she may decide that something is not right, and these queries must be undone. WebST, of course, cannot detect such a situation automatically.
The Web$SI$-based code resembles the natural language use-case specifications  This, of course, is a matter of judgement. We have tried to design our API in such way. We believe that method names, such as Select.one and UserComposed.yesNo, as well as the use of SQL contribute to this purpose.

In addition, we have seen in the example in Chap. 5 that each application logic (AL)-related natural language statement is translated to few Java code statements. We also saw that for each use-case specification statement, it is required to write few Java statements (the Java code also handles situations, which are hard to foresee at the specification stage). We believe that this kind of ratio can be maintained, when data processing is performed mostly by SQL, and it is not required to code an algorithm beyond SQL abilities. Anyway, when an AL-related natural language statement is translated to a small number of Java statements, this contributes to the issue of code resembling the specifications.

In many cases, not only method names contribute to code-specification resemblance, but also the code structure. Each use case of a specification is implemented in a separate use-case class. In addition, consider, for example, a case, in which an exceptional situation in a use case is coded with an \texttt{if} statement. The condition of the \texttt{if} statement is then mapped to the cause of the exceptional situation, as formulated in the use case. The body of the \texttt{if} statement corresponds to the use-case statements, which should be performed in this situation.

Direct system verification by code inspection  We have shown in Chap. 5, how Java code statements can be mapped to use-case specification statements. It was possible to obtain such a mapping because the Web$SI$-based code was short, high-level and declarative. After performing the mapping we could perform several activities, related to verification:

- Ensure that certain Java statements implement correctly the use-case specification statements mapped to them.
- Check that all statements in the use-case specification are indeed implemented, i.e., not forgotten.
- For Java statements that were left not mapped to any specification statements, check whether this points to a defect in the specification. That is, whether the specification does not consider a certain situation.

To conclude, we believe that inspecting a Web$SI$-based code can contribute more to system verification, than inspecting code, which employs traditional programming tools.

Extension and modification of the system  In a Web$SI$-based information system, a use-case object, i.e., an instance of a use-case class, usually interfaces only with the user and the database, but not with other use cases. The developer of a use case needs, therefore, not know anything about the internals of other use cases. She is required to be familiar with the use-case specification, the database schema and with the Web$SI$ API. As a result, the effort required for extending the information system with a new use case, or modifying a use case is thus independent of the number of already existing use cases, i.e., the extension complexity [33] of the architecture is usually $O(1)$.

The achievement of this $O(1)$ extension complexity is based on two assumptions. The first assumption is that the database schema is not changed. This assumption is probably often met, as observed by [21], that the schema of a database changes much
slower than database applications. The second assumption is that it is possible to design
the use cases such that they need not interface (except for the &lt;&lt;include&gt;&gt; relationship)
with other use cases. This assumption was met in all systems we have built with the
help of WebSI. There were, however, use cases that did not interface directly, but were
dependent on each other through common database data, by participating in the same
workflow. Consider, for example, a workflow, in which a worker in a company issues a
purchase request (one use case). The request must be approved by two financial clerks,
who may work in parallel (another use case, executed twice by different users). The
approved request is executed by a purchasing clerk (an additional use case). From our
experience, such use cases are best to be designed and developed together.

We have shown in this section that the design goals set in Chap. 1 are to a substantial
extent met by WebSI.

7.2 WebSI Compared to Its Predecessors

In Sec. 2.4 and 2.5 we have discussed two predecessors of WebSI—SI and SI+—
and their principle drawbacks. In addition, there were several early releases of WebSI,
which we do not consider here. It seems that WebSI corrects all SI and SI+ design
decisions, which we considered as unsuccessful. In addition, WebSI introduces features
beyond SI and SI+.

Let us now list the various drawbacks and limitations of SI and SI+, and see how
WebSI deals with them.

• In SI, the programmer’s code was sometimes IS-dependent. Switching to another
IS, therefore, might require code changes. In WebSI, the programmer’s code em-

ploying the same API for all IS’s. Therefore, the WebSI-based code does not have
to changed, when it is required to switch to an alternative IS.

• SI had no clear separation between the IO activities. WebSI groups the various
IO activities into activity classes. A certain activity class provides services for a
certain kind of activities. For example, the Output activity class provides services
for presenting outputs to the user. The Select activity class has services that enable
the user to select items out of predefined sets and so on.

• SI had no clean model for fetching input values supplied by the user. In WebSI,
fetching all user input values is performed through the Fetch activity class methods,
employing fetch ID’s, specified in the appropriate input requests.

• In SI, the programmer had to manually implement the initiation of a use case and,
in some cases, to define, in which order the use cases are executed. In WebSI,
the ready-available use-case displayer component takes care of use-case initiation
issues.

• In SI+, the programmer’s code mentioned specific GUI controls, and even used
Swing to manipulate them. The API of WebSI is declarative, it mentions the
desired UI functionalities, but not specific GUI controls or other UI properties.

• In SI+, the programmer had to manually manage the consistency between the
GUI’s of the various use cases. This was difficult, error-prone and led to situations,
in which use cases were directly dependent on each other. In WebSI, the use-case classes do not communicate, except for the \(<\texttt{include}>\) relationship.

- The selection of the IS to be employed, both in SI and SI+, was performed in the programmer’s code, in several places. In WebSI, the IS to be employed is specified at deployment time, and not in the programmer’s code. As a result, the programmer’s code does not have to be changed.
- We believe that WebSI produces more quality GUI than SI and SI+.
- SI produced stand-alone information systems and used single-query transactions to access the database. SI+ employed multi-query database transactions, but still produced stand-alone systems. Systems produced by WebSI are multi-user, employing database transactions for data consistency.

### 7.3 WebSI User Interface Construction Approach Compared to Other Methods

The primary goal of our project is to support the software engineering issues of the specification-oriented (SO) software construction methodology. Specifically, we wish to investigate the feasibility of a use case-oriented application framework for Web-based information systems. Therefore, WebSI should not be viewed as a UI design and construction tool. The SO construction of the UI is just one of WebSI challenges, involved in deriving the system implementation directly from its use-case specification. Other challenges of WebSI include database transactions management, handling multiple users, security. The UI construction is, however, one of the most important WebSI challenges.

In this section we first evaluate the WebSI UI construction methodology. Then we compare it to some of the traditional UI construction approaches: employing an object-based UI construction toolkit and the Model-View-Controller design (Sec. 7.3.2), and employing a visual UI builder tool (Sec. 7.3.3). As a representative of industrial efforts, in Sec. 7.3.4, we compare WebSI to the interesting JavaServer Faces [28] technology of Sun. Then, in Sec. 7.3.5, 7.3.6, 7.3.7 and 7.3.8 we describe several innovative UI development processes suggested by various researchers, and we compare them to WebSI.

It must be noted, that there has been and still is much academic and industrial activity in the field of UI design, development, operation and evaluation. For example, about twenty XML-based UI-related projects are listed in [64]. In addition, lately especially much activity is noted in the field of UI design and development methodologies for multiple devices and platforms. Many of the suggested approaches and methodologies are supported by appropriate tools. Therefore, for this section we have chosen to discuss only several approaches, which seem both interesting and related, and important enough based on their publications.

For the comparisons we have chosen the following criteria:

- The quality of the developed UI
- The effort required to develop the UI
- The effort required to perform changes in the UI. Here we also consider the effort required to switch to an alternative UI platform, if possible
- To what extent the UI can be reused for many applications
To what extent the approach separates the AL from the presentational issues

7.3.1 WebSI Specification-Driven User Interface

From the WebSI perspective, the essence of the UI is the data that the user inputs to the system, and the data that the system outputs to the user. The graphics of the GUI, for example, are only considered as a means to facilitate the job of the user: viewing the outputs and supplying the required inputs.

The user’s input and the system’s output comprise the AL of the information system and are defined in use-case specifications. Therefore, we believe that the UI should be driven by specifications, moreover, it should be the (automatic) consequence of implementing the specifications. Specifications implementation should provide enough information to WebSI in order to automatically generate the UI. In some way we reverse the order of activities of traditional software development processes (see Fig. 7.1), which recommend to design at least the sketches of the UI at the specification stage. Given a

![Diagram](image)

Figure 7.1: Traditional software development approach vs. WebSI approach

specification, we proceed directly to implementing it, i.e., to implementing the AL. Since the use-case specifications semantically define the data flow between the user and the system, we use them also to generate the UI. By translating the IO specifications into WebSI IO activity classes method calls, the programmer stays focused on application semantics rather than going into the low-level details of specific UI controls. Given the AL implementation, WebSI automatically generates the UI.

Let us now evaluate the UI construction facilities of WebSI according to the criteria we have suggested above.

The quality of the developed UI At present, the visual quality of the automatically generated UI’s is quite modest. A further research is required in order to improve the IS components, so that the produced UI’s could successfully compete with hand-made UI’s. In particular, current IS’s employ only basic layout techniques. We believe that these techniques can be improved with a modest effort.
In Sec. 7.4 we discuss our observations for systems developed with WebSI. It is a matter of taste whether the quite simple GUI’s produced by WebSI have a pleasing appearance. It was our impression, however, that the usability of the produced GUI’s was satisfactory. We felt that these GUI’s were easy-to-learn and enabled the user to accomplish their tasks quite easily and with few mistakes.

**UI development effort** The WebSI methodology of software construction does not include a stage, at which a specific UI is developed. Assuming that ready-made IS components exist, the programmer may concentrate on implementing the AL, she is not required to develop the UI at all. The UI is generated automatically, as a result of coding the use-case specifications.

Developing a new IS may, of course, require a substantial effort. But this activity is not related to any specific information system, it can be performed completely independently. Furthermore, the resulting IS can be reused in many different applications, significantly decreasing their development effort, and producing a uniform predictable UI for all these applications. We believe, that these properties make the investment into a new IS development well-justified.

**Ease of changing the UI** As we have already discussed, exchanging the IS component with an alternative one does not require changes or recompilations to the AL code. In WebSI this activity is performed manually at deployment time, by editing the deployment settings of a WebSI application.

In Sec. 7.3.2–7.3.8 we compare our method with some other known tools and methods for UI manufacturing. We note that our method is the only one, in which changing the UI appearance does not require any changes in the AL code, as we have only to exchange the IS. This is an important advantage, as it facilitates experimentations with different UI appearances in an attempt to find the most appropriate solution. Furthermore, our method is the only one, in which changes in the application code do not require any changes in the UI code. This is an advantage over methods, in which AL code and UI code are interwoven in each other.

The WebSI-based code that implements the use cases is completely environment-independent. This code does not mention any environment-specific details such as UI controls, database management and Web communication techniques. The AL code communicates with an environment through a declarative interface, which specifies only what is the required IO activity, but not how this activity is performed on a certain environment. Therefore, this code needs not be altered, if, for example, the IS component is exchanged, or another database software is employed.

Current WebSI environment includes a J2EE application server, a JDBC-compatible database engine, an HTTP communication protocol and a GUI of a standard Web browser. Consequently, all the existing IS’s are designed for a standard Web browser. Other environments have not been tried. Although WebSI does not directly attack the problem of multi-target UI construction, it is possible, in principle, to develop IS’s suitable for UI platforms, such as a Web-enabled mobile phone, a PDA and a command-line UI. However, some of the WebSI IO semantics may appear too sophisticated for a certain platform to handle. For example, the Select.one method may be hard to implement on a mobile phone platform. Further research is required on the suitability of the WebSI UI construction approach for different environments.
Reusing the UI  An important difference between WebSI UI construction approach and many UI construction methodologies and tools we know, is that WebSI IS component is not application-specific. An IS for a certain UI platform is developed once, and its development is completely independent of a certain information system. Being not application-specific, the same IS can be reused in many applications. For example, a software manufacturer may develop an IS, which incorporate this manufacturer’s UI appearance standards, such as colors, fonts and a company logo. This IS may be employed for a line of software products of this manufacturer.

Separation between AL and UI  When coding a certain AL, the WebSI programmer communicates with the environment through a declarative interface. The programmer specifies which IO activities should be performed, but not which concrete UI facilities will enable these activities. Therefore, the programmer is not required to have knowledge in any specific UI construction technology. On the other hand, the WebSI IS component developer is not required to have any knowledge about the AL of some specific information system. She can develop the IS completely independently. In WebSI, therefore, the separation between the AL and the IS seems to be quite complete.

7.3.2 Object-Based User Interface Construction Toolkits and MVC

Many popular toolkits for UI construction, such as [42, 17, 62, 27] employ the object-based approach for modelling the domain. In other words, a class (or a module, in non-OO languages) is defined for each one of the important objects of the domain. For example, the Java’s Swing toolkit defines classes for such domain objects as a push button, a text field, a list box, a radio button, a panel and a frame.

In such an environment, in order to code a UI, the programmer must first instantiate the classes of the UI controls. Then the programmer must configure the resulting objects properly, by invoking their various methods. Then it is required to arrange the controls in UI containers, such as frames, windows, dialog boxes and panels, i.e., to specify the layout of the UI. Finally, the programmer must code the event handlers of the various controls and attach the handlers to the controls. If a more custom UI is desired, the programmer must also specify additional properties of the UI elements, such as colors, fonts and graphics. In some cases, it is even required to implement custom UI controls, according to the extension rules of the employed toolkit. Programming at this level of abstraction can be quite labor-intensive, and the amount of the UI code can become significantly large. If at some stage of the development the UI has to be altered, it is required to rewrite and to recompile the UI code, and, in many cases, some of the AL code, as we will now demonstrate.

Let us take a look at a typical UI code, written in Java, employing the services of Swing. Let us assume, that a single-selection list box control is required for the UI. Let us also assume, that the programmer wants to be notified each time the user selects a different item in the list box. Here is the code, that the programmer has written:

```java
String[] opts = {"Option1", "Option2", "Option3", "Option4", "Option5"};

JList optionsList = new JList(opts);

optionsList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
optionsList.setSelectedIndex(3);

optionsList.addListSelectionListener(new ListSelectionListener(){
```

Technion - Computer Science Department - M.Sc. Thesis  MSC-2006-09 - 2006
```java
public void valueChanged(ListSelectionEvent listSelectionEvent)
{
    if (listSelectionEvent.getValueIsAdjusting())
        return;
    // Application logic goes here...
    ...
}
}
```

// Make the list box scrollable
JScrollPane scrollableList = new JScrollPane(optionsList);

This code has most of the properties discussed above: first the programmer instantiates the control, then configures it, and, finally, attaches an event handler. The code of the event handler should obviously implement the AL, which has to be executed when the user selects a different item in the list box.

Now consider a situation, in which the UI of the application has to be altered. Let us assume, that the list box from the previous example has to be changed to a combo box. The resulting code may look like follows:

```java
String[] opts = {"Option1", "Option2", "Option3", "Option4", "Option5"};
JComboBox optionsCombo = new JComboBox(opts);
optionsCombo.setSelectedIndex(3);
optionsCombo.addItemListener(new ItemListener() {
    public void itemStateChanged(ItemEvent itemEvent) {
        // Application logic goes here
        ...
    }
});
```

It can be noted, that switching from a list box to a combo box requires some code changes, because the employed classes—JList and JComboBox—have different API’s, i.e., different sets of methods and events. Code changes may even be required for the AL code, residing inside the event handler, because ListSelectionEvent and ItemEvent classes also have a different API. It seems that API differences in the UI element classes may impose quite a burden on the programmer. Altering the UI appearance requires rewriting much code, potentially introducing bugs.

The question is, what is the real reason for the API differences? To answer this question we suggest to ask the following question for our example: what UI functionalities the combo box and the list box control support? Table 7.1 lists two basic functionalities for each one of these controls. We see that indeed the two controls share one functionality,

<table>
<thead>
<tr>
<th>Combo box</th>
<th>List box</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select a single value out of many</td>
<td>Select a single value out of many</td>
</tr>
<tr>
<td>Specify a user-composed value</td>
<td>Select zero or more values out of many</td>
</tr>
</tbody>
</table>

which is selecting a single value out of several values. But we also see, that a combo box supports something a list box does not, and vice versa. We believe, that this is the real reason for API differences in UI control classes. The API of a UI control class must support all the possible UI functionalities of the control. Since different controls have different UI functionalities, the API’s of their classes are different.
Let us now return to the code of the example. For the `JList` class, the default selection mode is multiple selection, therefore, the `JList` object must be explicitly configured for a single-selection mode. `JComboBox`, on the other hand, does not support the multiple-selection mode, but it can be editable or non-editable, i.e., it may include an editable field, in which the user can type a value. In our example, the programmer has to be aware that by default `JComboBox` is non-editable.

To finish the example, let us point at a specific difference between the two classes. The `JList` class has a method for selecting multiple items:

```java
public void setSelectedIndices(int[] indices);
```

This method cannot appear in the API of `JComboBox`. The `JComboBox` class, however, has a method for controlling its editability:

```java
public void setEditable(boolean isEditable);
```

This method cannot appear in `JList`.

It seems, that many popular UI construction toolkits follow the approach of Swing, i.e., they define a class for each UI control, and design the class API to support all the control’s functionalities. Such a design results in API difference problems, as we have illustrated. Figure 7.2 demonstrates for various UI controls some of their common functionalities.

<table>
<thead>
<tr>
<th>Control</th>
<th>Initiate an action in the system</th>
<th>Select one item out of many</th>
<th>Select zero or more items out of many</th>
<th>Specify a user-composed value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push button</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hyperlink</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combo-box</td>
<td>±</td>
<td>✓</td>
<td></td>
<td>±</td>
</tr>
<tr>
<td>Drop-down list</td>
<td>±</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Listbox</td>
<td>±</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Set of checkboxes</td>
<td></td>
<td></td>
<td></td>
<td>±</td>
</tr>
<tr>
<td>Set of toggle buttons</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Set of radio buttons</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Menu</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Text field</td>
<td>±</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Slider</td>
<td>±</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spinner</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

Figure 7.2: Common user interface controls and their functionalities

When a programmer picks and employs a certain UI control, e.g., a list box, what she really wants is to employ one of its functionalities, e.g., selecting a single value out of a values set. Therefore, in `WebSI`, we decide to take the approach of semantic UI specifications [53]. A `WebSI` programmer specifies which IO functionality she needs, rather than dealing with a particular implementation of that functionality. The programmer’s decision on employing a certain IO functionality originates in the specification, and,
therefore, has a low potential for changing—unless the specification changes. We design the WebST API to support various IO functionalities, which are useful for our problem domain.

Let us now discuss another aspect of UI development with object-based UI toolkits, which is the communication between the constructed UI and the AL. Obviously, the simplest way to connect the AL to the UI is to fully implement the required AL in an event handler’s code. In that case, the event handler’s code may include the following activities:

- Inspecting the properties of the event
- Inspecting the values of other UI controls
- Performing the required calculations, possibly with the help of external software components, such as the database
- Presenting the output to the user by setting the values of the relevant UI controls

The result of the described simple approach is that the AL and the presentational aspects become interwoven, and this leads to interdependencies between the application components, the coupling between them becomes high and they are difficult to alter and maintain.

Actually, the challenge of separating the AL from the presentational issues is one of the fundamental software engineering challenges. Ideally, those two components should be completely independent, changes in one of them should not trigger any changes in the other. It would be desirable just to be able to plug-out the unsuccessful component and to plug-in a different one.

The Model-View-Controller (MVC) [16] design pattern (see Fig. 7.3) is intended to decouple the presentational issues from the AL, and both these from the application’s data model. This design pattern proved important enough to be later adopted by UML [58] class stereotypes—“entity class” (Model), “boundary class” (View) and “control class” (Controller).

The presentational issues (View) are decoupled from the data (Model) by establishing a pull or a push consistency protocol (see Fig. 7.3). When employing the pull protocol,
the View is responsible to update itself when needed, by calling the methods of the Model. When using the push protocol, the View registers itself as a listener for Model events. The Model fires an event when its state changes. For both protocols, several different Views may be employed for the same Model.

The Controller is responsible for handling events that arrive from the View; these events originate from the user’s actions. The Controller updates the state of the Model according to the event type and to the AL.

When the MVC pattern is not employed, the event handler’s code of a certain UI control contains the full implementation of the action, required by the AL, as we have already discussed. This simple approach usually results in re-implementing, or copying and pasting the AL and the data access code over many handlers, which further requires maintenance in several places. With the help of the MVC pattern, the handlers’ code is short and merely delegates to the high-level components of the Controller or the Model.

We now summarize the evaluation of this section according to our criteria.

**The quality of the developed UI** Depends on the richness of a UI toolkit and the expertise of the developer.

**UI development effort** As we have shown, constructing a UI using the objects of a UI toolkit requires a substantial development effort. It may be especially labor-intensive to define the layout of the UI, i.e., the sizes of the UI controls and their arrangement on a screen.

In addition, some aspects of the MVC pattern implementation may require a considerable development effort. For example, when employing the push protocol, the model developers must remember to send notifications to the relevant views each time the state of the model changes. The implementation may become even more difficult and introduce performance issues for a multi-user system, in which a certain view may receive concurrent update notifications originating from other users’ actions, while being operated, in addition to that, by its own user.

**Ease of changing the UI** As we have shown, altering the UI appearance involves rewriting and recompiling the UI code and some of the AL code. In addition, many UI toolkits are platform-dependent, i.e., they have versions for a single platform only.

**Reusing the UI** If the toolkit is employed as-is, i.e., if no special design effort is invested (such as in WebSI), the constructed UI is application-specific, and cannot be reused for other applications.

**Separation between AL and UI** When not employing the MVC pattern for communication between the AL and the UI, there may be no separation at all. The AL code is interwoven in the UI code, such as in event handlers. The UI code is interwoven in the AL code, such as in “push” updates.

MVC decreases the coupling between the AL and the presentational issues of the application, but not eliminates it. Coding a handler still requires to be familiar with classes and methods of the Controller or the Model components. Establishing the push protocol, on the other hand, requires firing appropriate events, so that the UI can update itself. This firing is not a part of AL.
7.3.3 Visual User Interface Builders

To this end, many rather sophisticated integrated development environments (IDE) exist, which include visual UI building tools [48, 43, 4].

When employing these IDE’s, the UI is constructed visually, rather than by programming. The developer drags the various UI elements and places them on UI containers. Then the developer adjusts the UI elements, e.g., by resizing or by altering their default properties through visual editors. Finally, the developer specifies to which events of a certain UI control she would like to react, and the tool generates a skeleton of the required event handler, which is further manually edited by the developer.

An interesting issue is in which format the tool saves the produced UI. One option is to generate code that employs a certain UI construction toolkit, preferably in the same programming language as for the whole application. In that case, the generated code can be understood by the developer and edited manually, when the IDE is not available or fine-grain UI adjustments are required. Another option, is to employ some proprietary format that describes the UI. In that case, of course, the programmer may not be able to maintain the UI manually.

The popularity of visual UI building tools stems from the fact that they automate the tedious process of UI construction, i.e., controls instantiation, configuration and layout. The latter is especially significant, since correctly laying out the UI is considered quite a complex process.

We believe that there are several potential disadvantages when employing visual UI builders:

- It is still the programmer’s job to design the UI. That is, the programmer must still decide which specific UI controls will be employed and what will be their layout on the screen.

- Even in case a UI builder employs a known programming language to save the UI, and not a proprietary format, the code generated by it is often messy and hard-to-maintain.

Another issue is whether the tool employs a certain UI toolkit correctly. For example, a certain tool based on Java’s Swing toolkit may employ an absolute positioning for a certain complex UI layout. This means that the tool will hard code the coordinates and sizes of the various UI controls. Absolute positioning is not a preferred UI layout technique, since an absolutely positioned UI is extremely hard to maintain and modify. The preferred technique is to employ layout managers, but the tool may decide not to employ them.

- UI builders are usually platform-dependent. An application created by a desktop UI builder may not always be easily ported to a mobile device platform, for example.

- UI builders enforce the programmer to make decisions on the UI appearance at relatively early stages of the development process.

We will now summarize our visual UI tools evaluation according to the suggested criteria.

The quality of the developed UI To produce a quality UI using a certain UI toolkit an expert knowledge of this toolkit may be required, e.g., for producing complex UI layouts. When employing a visual tool, the UI is designed visually by direct manipulation,
therefore no expert UI programming knowledge is required. As a result, the quality of the produced UI’s is quite high.

**UI development effort** Since the UI is constructed visually, the development effort is quite small.

**Ease of changing the UI** It may be performed by employing the visual tool or by manually editing the UI code. When employing a visual tool, the new UI code is generated automatically, replacing the old one. However, there are IDE’s that do not handle this process correctly, they fail to ensure the overall consistency across the changes. In addition, it is, of course, required to update the event handlers’ code, so that they correctly operate the new UI controls.

When editing the UI code manually, a significant effort may be required, e.g., because of a messy UI code, incorrect usage of the UI toolkit or the complex proprietary format.

**Reusing the UI** The constructed UI is application-specific and cannot be reused in other applications.

**Separation between AL and UI** When employing visual UI builders, it is possible to achieve the same high level of separation, as with a UI toolkit and MVC. However, the automatically generated code may not always be optimal for such a high-level separation.

### 7.3.4 The JavaServer Faces Technology

An interesting technology for UI construction, called JavaServer Faces (JSF) [28] has been introduced by Sun. This technology is an extension of the popular JavaServer Pages (JSP) [29] technology with a set of tags, allowing a higher-level UI description than with basic HTML [20] tags.

In order to insert a UI component into a JSP page, a custom tag is used, that is a combination of a component functionality and rendering attributes. For example, in order to insert a command invocation component, one of two tags should be used: `commandButton`—resulting in a push button or `commandLink`—resulting in a hyperlink. In order to allow the user to select a single item out of the set of items, one of the three tags can be used: `selectOneMenu`, `selectOneListbox` or `selectOneRadio`, each one resulting in the appropriate control.

We evaluate the JSF technology, according to our criteria:

**The quality of the developed UI** The JSF developer has freedom in specifying the appearance of a particular JSP/JSF page: the employed UI controls, their layout, colors, fonts, images and so on. In addition, the JSF technology brings several rather sophisticated UI controls, beyond the basic HTML controls. As a result, the produced UI may be of a very high quality.

**UI development effort** The development effort is lower than with basic HTML tags, however, it still is quite significant. The developer must be familiar with both JSP and JSF technologies.
Ease of changing the UI  JSF tags, as demonstrated, specify not only the desired UI functionality, but also the desired implementation. This may lead to difficulties in exchanging the UI appearance. When a JSF tag of a certain UI component is exchanged with another tag, the new tag may introduce a different set of attributes, requiring further changes.

A better way of altering the UI appearance of a JSF page is with multiple renderers for the same UI component. Each such renderer defines a different way to render the component. Several renderers may be used for the same platform, and there may be renderers that render the same component on different platforms.

It is not completely clear to us why JSF chooses to specify both the desired functionality and the rendering attributes in their tags. If only the functionality were specified, the JSF tags would then be quite similar to WebST IO semantics. In that case, for example, the same command tag could be used to produce buttons and hyperlinks, by exchanging its renderer.

Reusing the UI  Being an extension of JSP, JSF shares an intrinsic JSP property: the constructed UI is application-specific. In other words, for each application its own set of JSF/JSP pages has to be developed. These pages cannot be readily employed in other applications.

Separation between AL and UI  When using JSF, “the page author and the application developer . . . need to work in tandem to make sure that the component tags refer to the proper UI component properties, to ensure that the properties have the acceptable types, and to take care of other such details” states the J2EE tutorial [25]. The JSF separation between AL and UI is better than of plain JSP’s (by introducing some higher-level constructs), but is far from being complete.

7.3.5 The CTTE and TERESA Tools, and the Related Method

In [53, 54, 44, 45] a method for designing and implementing multi-device, highly usable, context-sensitive interactive software systems has been suggested. The design process starts with a high-level task modelling, which specifies:

- the various possible contexts of use
- the various users’ roles
- the domain objects to be manipulated and relationships between them
- the platforms suitable to support each task

At the next stage, the task model is filtered and refined for each target platform by:

- removing tasks that cannot be supported by the platform
- adding navigational tasks necessary to interact with the considered platform
- adding details on how a task is decomposed for the target platform

The next step is to build an abstract UI model for each target platform. This model contains a set of abstract presentations and a set of transitions between the presentations. A presentation is described as a set of abstract interaction objects (interactors), composed
through a number of different operators—grouping, ordering, relation and hierarchy. The final step is to generate the platform-specific UI, according to the abstract UI model for this platform.

The suggested design process is supported by CTTE [44] and TERESA [45] tools. The CTTE tool employs the ConcurTaskTrees [52] notation in order to perform the task modelling. In addition, it allows for each task and for each domain object to specify the platforms, on which this task or domain object can be supported. Then it is possible to derive a task model filtered for a specific platform, and to save it for further usage by the TERESA tool. The TERESA tool allows to select a specific communication technique for each one of the interactors and composition operators. Some global UI parameters can also be specified. Finally, TERESA generates the UI for the desired platform.

The described approach is based on a single integrated task model, developed by the authors. The same single task model is used to derive all the platform-specific UIs. This helps to maintain a high level of consistency across the multiple UIs developed.

The suggested approach considers several possibilities when moving between different platforms, many of which \( \text{WebSI} \) cannot support:

- The same task can be performed on multiple platforms in the same manner, i.e., its presentation remains mostly unchanged on different platforms. If this holds for a certain task, \( \text{WebSI} \) can support it as well.

- The same task can be performed on multiple platforms, but with different UI objects. \( \text{WebSI} \) can support this possibility, by using an IS that employs UI controls, suitable for the presentational facilities of a specific platform.

- The same task can be performed on multiple platforms, but with different set of domain objects. For example, on a desktop PC all the available details about a certain artwork can be presented: title, image, description, author, material and date. On a mobile phone platform, on the other hand, only a low-resolution image and a title are presented. \( \text{WebSI} \) cannot support such a level of granularity.

- The same task can be performed on multiple platforms, but with different task decomposition (structure). This means that the task is sub-divided differently, with different sets of sub-tasks, depending on the platform. For example, desktop users may read long reviews about artworks, while to mobile phone users this option is inaccessible. \( \text{WebSI} \) cannot support this.

- The same task can be performed on multiple platforms, but with different time constraints, i.e., there is a difference in the temporal relationships among the sub-tasks. For example, on a PC the user may be presented a set of fields to fill. She may fill them in any order and then approve. On a mobile phone, due to its screen limitations, only one field can be presented at a time. Therefore, the user is forced to fill the fields in a predefined order, and to separately approve each field. \( \text{WebSI} \), in general, cannot support this.

- There exist dependencies among tasks performed on different platforms, i.e., a task on platform A affects the performance of other task on platform B. For example, consider the user visiting a museum and annotating the most interesting works of art on a PDA. When she arrives home and accesses the museum website through a desktop system, she would appreciate being able to receive information regarding these works first. \( \text{WebSI} \) cannot support this.

We now evaluate the described approach according to our criteria:
The quality of the developed UI  The approach especially focuses on producing highly usable interactive systems. On each platform the most suitable set of tasks is accessible, according to the platform’s presentational facilities and to the anticipated context of use.

UI development effort  The design process, despite being partly automated, requires a substantial effort.

Ease of changing the UI  The process especially focuses on producing a set of alternative UI’s from a single model, each one suitable for a certain platform.

Reusing the UI  In general, the approach is application-specific, because it starts with a task model, developed for a certain application. The various models, obtained during the process, cannot be readily reused for other applications.

Separation between AL and UI  The described approach does not consider implementing the AL for the multiple platforms. It is, therefore, not clear whether after applying the proposed approach, the AL code running on a certain platform will be free of presentational and other platform-specific details. It is also not clear whether the AL will be easy to verify, modify and extend.

7.3.6 UIML—User Interface Markup Language

UIML [1, 12] is an XML-based language for describing UI’s. The objective of UIML is to serve as a language with a single syntax, which allows to describe a UI for any device, any target language (e.g., Java, C, HTML), and any operating system on the device. When employing UIML, the developer first designs separate UI’s for every target device. Then, all these designs can be expressed in UIML, possibly in a single UIML document. UIML thus “does not magically creates multiple UI’s from a single description; instead it is a language, in which those multiple UI’s can be recorded”. Each one of the multiple UI’s is potentially targeted at a different device.

UIML does not contain tags, which describe UI elements specific for a certain device. Instead, UIML uses a set of generic tags, such as part, which describes an element of the UI, and property, which describes a property of a certain UI element. When writing UIML, the developer specifies instance and class names of her own choice for UI elements, properties and events. Then the developer maps those names to “abstract widget names”, which are further mapped to specific UI elements of a certain UI construction toolkit. The second mapping, i.e., the mapping from “abstract widget names” to specific UI elements is provided via an externally-defined vocabulary. There exist vocabularies for many target devices and operating systems, such as Java AWT and Swing, WML (a markup language for mobile phones), HTML and a couple of Unix windowing toolkits.

A skeleton of a UIML document looks like follows:

```xml
<peers> ... </peers>
<interface>
  <structure>
    <part> ... </part>
  </structure>
  <style>
    <property> ... </property>
  </style>
</interface>
```
The peers element specifies the employed vocabulary. The structure element enumerates a set of UI parts and their organization for various platforms and devices. Here user-defined names for UI elements, properties and events are used. The style element maps the user-defined names to the “abstract widget names” in the employed vocabulary. The content element associates words, sounds, and images with UI parts. The behavior element enumerates a set of rules that describe how the UI should react on different events, from the user, from a device, or from the application.

When using platform-specific vocabularies, it is necessary to create separate UIML files for the same UI functionality on the different devices. The actual number of lines that are common in the separate UIML files is very small, due to differences in platform-specific UI elements and their properties. To cope with this and other problems of platform-specific vocabularies, the authors suggest employing generic vocabularies.

There exist two kinds of generic vocabularies. The first kind is a “truly” generic one, which can accommodate all possible families of devices and their UI’s. Most of the elements of this vocabulary can be assumed to exist on any device. The second kind is a vocabulary specific to a certain family of devices. For example, there might be a vocabulary for GUI’s and another vocabulary for voice UI’s. A UI can be developed using elements from both the first-kind and the second-kind vocabularies.

An element in a generic vocabulary can be mapped to several possible UI elements of a specific platform, or to a combination of platform-specific elements. A generic element might also not have any corresponding elements on a certain platform.

We now evaluate UIML according to our criteria:

**The quality of the developed UI** UIML does not impose any constraints on the quality of the UI.

**UI development effort** The development effort can be significant. For example, when employing non-generic vocabularies, the UIML user has to design an application-specific UI for each target device.

**Ease of changing the UI** Similar to WebST, UIML allows to alter the UI appearance and the UI platform without changing the UI description. This is done by changing the mappings in the style element, i.e., by changing a vocabulary.

Unlike WebST, UIML vocabularies, even the generic ones, contain specific UI components, such as “GList” and “GSLTextRegion”. Potentially, this may make altering the UI appearance harder. It can be suggested to construct a vocabulary which describes various UI functionalities (such as “SelectSingle”, “SelectMultiple” etc.) rather than specific UI controls. This approach, however, has not been investigated by the authors.

**Reusing the UI** Some of UIML elements such as peers and style may be reused across many applications. Other elements, such as structure, content and behavior seem clearly to be application-specific.
Separation between AL and UI  This issue is not directly considered.

7.3.7 XIML—Extensible Interface Markup Language

XIML [55] has been inspired by recent industry and academic efforts to standardize the representation of data, e.g., by XML, and protocols for applications interoperability, e.g., by SOAP [60]. Similarly, XIML strives to provide a “standard mechanism for applications and tools to interchange interaction data and to interoperate within integrated UI engineering processes, from design, to operation, to evaluation”. Interaction data is defined as data that “defines and relates all the relevant elements of a UI”.

XIML is targeted to attack several challenges, which include:

- “A comprehensive, structured storage mechanism for interaction data”. XIML should enable to store and manage repositories of all data relevant to a single UI or to a collection of UI’s.
- Support the complete life cycle of a UI, i.e., design, development, operation and evaluation.
- Represent not only the concrete aspects of a UI, such as specific UI controls, but also the abstract aspects, such as interaction context.
- Effectively relate its various elements among themselves, especially relating abstract and concrete interaction data elements.

XIML categorizes its elements into major UI components, the basic of which are:

Task Captures the user tasks carried through a UI. Defines “a hierarchical decomposition of tasks and subtasks that also defines the expected flow among those tasks and the attributes of those tasks”.

Domain An organized collection of classes and objects, which are viewed or manipulated by the user.

User A hierarchy of users, which are individual users or user groups.

Presentation An hierarchy of concrete interaction elements.

Dialog A structured collection of elements “that determine the interaction actions that are available to the users of an interface”. Dialog also specifies the flow of interaction actions, thus defining a navigation of the UI.

An important feature of XIML is explicitly capturing the relations among the various elements. An example of a relation is “Data type A is displayed with Presentation Element B or Presentation Element C”. The set of XIML relations “creates a body of knowledge that can support design, operation, and evaluation functions for UI’s”.

XIML seems to especially effectively attack the challenge of multi-platform UI development. This is achieved by defining a “Presentation” component for each target device, while the rest of the UI specification remains unchanged. A “Presentation” component specifies which specific UI controls will be employed to display each data item on a certain device. The rendering of the UI is completely left to a target device.

Since creating a “Presentation” component for each target platform can be quite labor-intensive, the authors suggest an automation framework. A single “intermediate” presentation component is created, and XIML relations are used to determine how elements of
this component map to specific UI controls on a target device. XIML also allows these relations to be created and updated at runtime by intelligent tools, taking into account various constraints such as screen size, other UI elements on the screen, user preferences and contextual issues.

An additional goal of XIML is to enable a UI management at runtime. An example of this is dynamic presentation reorganization. Here a “Presentation” component of a specification is exchanged by another one, according to the current screen size, for example. A “Presentation” component can also be exchanged according to user preferences, captured in the “User” component of XIML specification.

According to [55], XIML has quite a broad applicability and scope.

We now evaluate XIML according to our criteria:

The quality of the developed UI XIML seems to not impose any constraints on the quality of the UI.

UI development effort Specifying a UI in XIML may involve a substantial effort, as several models may have to be developed.

Ease of changing the UI It seems that the “Presentation” component of XIML is quite similar to WebSI IS component. Both completely determine how the final UI on a specific device will look like. Exchanging the “Presentation” component, similar to exchanging the IS component, does not require changes in other parts of UI specification.

Reusing the UI It is not clear from [55] whether the “Presentation” component is application-specific, we believe that at least it can be application-independent. Other XIML UI specification components, such as “Domain”, “Task” and “Dialog”, seem clearly to be application-specific.

Separation between AL and UI This is not directly considered.

7.3.8 Plastic User Interfaces

A UI is plastic [8, 6, 7] if it is able to adapt to different contexts of use, while preserving usability. The term “plasticity” is inspired by the property of certain materials to change their shape under external constrains without breaking. A context of use for a plastic UI is defined by two classes of physical entities:

- The hardware and software platform, which is used for interacting with the system. Here the important attributes include the employed UI construction toolkit, screen size and resolution, memory size, network connection speed and so on.
- The physical environment where the interaction takes place. The environment includes “objects, persons and events that are peripheral to the current task(s) but that may have an impact on the system and/or the user’s behavior, either now or in the future”. Some important attributes of the environment are noise and light conditions.

In [8, 6, 7] a reference framework is proposed that structures the design process of plastic UI’s. In addition, a tool called ARTStudio [7], that partly supports the framework
is described. The suggested approach is model-based and its input are *initial models*. These initial models include known models, improved by the authors, and some new models, introduced by the authors. The existing models are the domain model and the task model. The new models are:

- The *platform model* and the *environment model* define the various contexts of use intended by the designers. These two models create a 2-dimensional space, whose axes rank the various platforms and environments. For example, a PC platform is ranked lower than the PDA platform, since the PC imposes fewer constraints on the UI. Similarly, an office environment is ranked lower than the street environment, if speech is planned as an interaction technique. Each couple \((x, y)\), such that \(x\) belongs to the “Platform” axis and \(y\) belongs to the “Environment” axis, form a context of use. The *plasticity domain* (see Fig. 7.4, taken from [7]) of a certain UI is the surface formed by all such couples that this UI is able to accommodate.

![Diagram showing plasticity domain and plasticity threshold of a user interface](image)

**Figure 7.4:** Plasticity domain and plasticity threshold of a user interface. Picture taken from [7].

- The *evolution model* specifies what should be done when a context changes. This model includes:
  - *Triggering conditions*, such as leaving a certain context and entering another context
  - Computation of the *reaction*, which includes identifying the candidate reactions and selecting one of them, according to acceptable *migration cost*, i.e., the effort the system and the user must put into a certain reaction. Possible reactions include switching to another platform, employing a different UI and adapting the current UI to the new context. In addition, a reaction may include a *prologue* and an *epilogue*, which are performed respectively before and after applying the reaction.

- The *interactors model* describes which concrete UI widgets are available

During the process, *transient and final models* are created. The first transient model is a task-oriented specification of the UI. This specification is obtained from the domain model, the task model and the platform model. From this specification and from the platform model, an *abstract UI model* is derived, which in ARTStudio is a set of *workspaces* and a *navigation scheme* between them. The *concrete UI model* is derived from the abstract UI model, with the help of the environment and the interactors model. The final
running system is produced from the concrete UI model. The process of deriving a model from another model is called reification. The design process described so far is presented on Fig. 7.5, taken from [6].

![Figure 7.5: Designing a user interface for a single context. Picture taken from [6]](image)

In order to target multiple contexts, the developers apply the described process for each context of use, while employing translation between models that are on the same level of reification (see Fig. 7.6, taken from [7]). There are many possible combinations for using translations and reifications in a particular design process. The ideal situation is, however, when reification is used until the very last step, at which the translation is used, as demonstrated on Fig. 7.7, taken from [7].

One of the factors that influence the quality of the proposed design process is the level of sophistication of the supporting tool. Concerning the plasticity of the UI, the quality of the process depends on the complexity of the various models the tool can support: the environment model, the platform model, the interactors model and the evolution model. The described ARTStudio, for example, does not support the environment and the evolution models, and its platform model allows to model only the screen depth and size.

We will now evaluate the proposed approach according to our criteria:

**The quality of the developed UI**  The proposed approach can produce highly qualitative UI, as it involves detailed modelling of the UI, with human intervention.

**UI development effort**  Even in the ideal design process, depicted at Fig. 7.7, the development of a plastic UI requires a considerable effort. Several detailed models are required to build.
Ease of changing the UI The UI resulting from the proposed process can adapt itself automatically, in real-time to the changes in the context of use. The UI generated by WebSI currently can be altered only by manually exchanging the IS. In principle, the WebSI approach can be extended to exchange the IS component automatically, according to the client software requests, but at present this is not supported.

The contexts of use addressed by WebSI currently include only the UI platform, e.g., HTML vs. command-line UI. WebSI does not address such issues of the user’s physical environment as noise and light conditions. In addition, as mentioned before, WebSI has not been tested for UI platforms, other than the standard Web browser.

Reusing the UI Some of the models in the proposed process may be application-specific. Although some of the initial models, such as the platform and the environment model, may be application-independent, all transient models are, of course, application-specific. This means that for each new interactive system, its appropriate initial models must be developed, and then the process is applied, producing transient application-specific models. The application-specific models cannot, of course, be reused for other applications.

Separation between AL and UI It is not clear what are the software engineering qualities of complete systems (not just the UI’s) constructed according to the proposed approach. Specifically, it is not clear whether it is possible to code the AL, so that it does not address any UI details, and vice-versa.
7.3.9 Comparisons Summary

In this section, we summarize the UI construction approach comparisons we made. We are doing this, so that the reader can get an immediate impression of strengths and weaknesses of the different approaches. For the summary, we employ the following scale:

<table>
<thead>
<tr>
<th>Scale</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>very good</td>
</tr>
<tr>
<td>+</td>
<td>good</td>
</tr>
<tr>
<td>0</td>
<td>acceptable</td>
</tr>
<tr>
<td>−</td>
<td>poor</td>
</tr>
<tr>
<td>−−</td>
<td>very poor</td>
</tr>
</tbody>
</table>

Table 7.2 presents the summary.

Table 7.2: User interface construction approach comparisons summary

<table>
<thead>
<tr>
<th>Approach</th>
<th>UI Quality</th>
<th>UI Development Effort</th>
<th>Ease of changing the UI</th>
<th>Reusing the UI</th>
<th>Separation between AL and UI</th>
</tr>
</thead>
<tbody>
<tr>
<td>WebST</td>
<td>0</td>
<td>++</td>
<td>+</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>UI toolkits</td>
<td>can be any</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>+ with MVC</td>
</tr>
<tr>
<td>Visual UI builders</td>
<td>++</td>
<td>−</td>
<td>0</td>
<td>0</td>
<td>0 or +, depends on a tool</td>
</tr>
<tr>
<td>JSF</td>
<td>++</td>
<td>−</td>
<td>0</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>CTTE, TERESA</td>
<td>++</td>
<td>−</td>
<td>++</td>
<td>−</td>
<td>not clear</td>
</tr>
<tr>
<td>UIML</td>
<td>can be any</td>
<td>−</td>
<td>+</td>
<td>−</td>
<td>not clear</td>
</tr>
<tr>
<td>XIML</td>
<td>can be any</td>
<td>−</td>
<td>+</td>
<td>−</td>
<td>not clear</td>
</tr>
<tr>
<td>Plastic UI’s</td>
<td>++</td>
<td>−</td>
<td>++</td>
<td>−</td>
<td>not clear</td>
</tr>
</tbody>
</table>

It can be seen from the table that WebST, as expected, attacks especially well the software engineering aspects of UI construction, specifically, the development effort, reuse and the AL-UI separation, while producing UI’s of modest quality. The traditional UI construction approaches involve a considerable effort and do not allow easily changing and reusing the UI. JSF is targeted at producing quality UI’s with acceptable software
engineering qualities, except for reuse. The various model-based approaches are successful in facilitating changes in the UI, but do not attack directly the software engineering challenges.

7.4 Testing \textit{WebSI} on Student Projects

We have held two student laboratories, in which the students employed \textit{WebSI} to construct information systems for various purposes. The laboratories were held during Winter 2003–2004 and Spring 2004 semesters. During this time \textit{WebSI}, of course, evolved. In fact, four releases of \textit{WebSI} were developed during this project, one of them being a maintenance release. During Winter 2003–2004 semester, the first \textit{WebSI} release was tested. During Spring 2004 semester (and the following summer), the second, the third and the fourth releases were tested.

At the beginning of a laboratory, the students received a description of a desired system. The descriptions were quite detailed, although they, of course, did not completely elicit the requirements. The students were requested to analyze the suggested requirements, and to correct and complement them if necessary.

One of the systems in Spring 2004 semester was suggested by the students.

In addition, in Spring 2004 semester, two teams were requested to further develop the projects produced during the previous semester. This included updating the code to match the new \textit{WebSI} release, correcting the defects of the previous version and implementing new functionalities. We performed this experiment in order to investigate how our proposed software development methodology supports extension and modification of existing systems.

The first assignment of the lab was to produce a use-case specification of a system. The specification included a use-case diagram and the detailed descriptions of the use cases. This assignment also included the students’ corrections and complements of the requirements. In addition, in this assignment the students were requested to write some test cases for their future systems. Writing test cases, at such early stage, without having a single line of code written, was intended to help discovering problems in the use-case specifications.

The second assignment was devoted to database design. (During the second lab, the first and the second assignment were submitted together.) The students were required to design a schema of their database and to provide some sample data. Special attention was payed to using database constraints, such as PRIMARY KEY, FOREIGN KEY, UNIQUE, CHECK and NOT NULL.

The rest of the semester was devoted to implementation. During the last week of the semester, the final presentations were held. Each final presentation lasted about two hours. Students explained the requirements, demonstrated the systems and briefly went over the code. The instructors gave their feedback, which included desired corrections and guidelines for writing the documentation.

After the presentations, additional time was given for preparing projects documentation, which included:

- A summary of the requirements, including those corrected and complemented by students.
- The use-case analysis, including use-case diagram and use-case descriptions.

\footnote{Our pre-thesis project, \textit{SI+}, was tested during Spring 2003 semester.}
• Implementation details. For each use case, some notes on its implementation were included.

• Database design, including the description of the database schema and sample data.

• Test cases for the system verification.

At the end of the lab, the students were asked to fill an evaluation questionnaire. Table 7.3 summarizes the timetables of both labs. It, of course, should be noted, that our students were also taking other courses during both semesters. So the time periods in Table 7.3 were, of course, not solely devoted to our lab.

We will now describe the systems, that were implemented by the students during both semesters. Then we will summarize the hours devoted solely to our lab, as reported by students.

**Winter 2003–2004 Semester**

**AU** The Air Union project. This system allows customers of a group of cooperating airline companies, called “The Air Union”, to obtain information on their scheduled flights. Customers operate the system by searching for routes between a source city and a destination city on particular date. A route contains one or more direct flights, with stops between them. When searching, the customers may specify various constraints, such as the desired arrival days, minimal and maximal waiting times and a specific airline to fly with. The Air Union employees can perform updates to the system’s data, such as adding, removing and modifying the flights of the different airline companies.

Difficulty: low–medium
Number of students: two

**CA** The Course Assignments project. At a certain faculty care is taken to try to avoid having a faculty member teach a course, with which he or she is not familiar. Teaching a course, that a faculty member has never taught before, is known as doing a “new prep” (“prep” being short for “preparation”), and is an extremely time-consuming process. The job of assigning classes in a way that minimizes the number of “new

<table>
<thead>
<tr>
<th>Period</th>
<th>Winter 2003–2004</th>
<th>Spring 2004</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time for the first assignment</td>
<td>5 weeks (including lab meetings, on which WebSI was taught)</td>
<td>10 weeks (including lab meetings and the Passover holiday)</td>
</tr>
<tr>
<td>Time for the second assignment</td>
<td>2 weeks</td>
<td></td>
</tr>
<tr>
<td>Time left until final presentations (as planned)</td>
<td>6 weeks</td>
<td>6.5 weeks</td>
</tr>
<tr>
<td>Additional time after the presentations</td>
<td>2–6 weeks, different for each team. This time also included student exams</td>
<td>8 weeks for teams, which met the final presentation deadline; up to 14 weeks for the other teams. This time also included student exams, and some students worked full-time during the summer</td>
</tr>
</tbody>
</table>
"preps" is a difficult one. The primary functionality of the system is to perform automatic course assignments, such that the number of "new preps" is minimized.

Each lecturer at this faculty is a member of a certain academic rank, such as "professor", "associate professor", "assistant professor", "visiting instructor" etc. The rank can change over time. Each different academic rank has a maximum number of "new preps", that should be given to a lecturer of that rank.

For each lecturer, the system stores which courses this lecturer has taught at the faculty. In addition, a lecturer may have taught a course at another institution, which is roughly equivalent to the course at the faculty; this information is also stored. In case of teaching an equivalent course at some other institution, teaching this course at the faculty is not considered a "new prep".

The system performs automatic course assignments, taking a set of constraints as an input. Each constraint is of the form: "Mr. X will teach course Y". In addition, the system allows performing various updates, such as adding a lecturer, modifying an academic history of a certain lecturer, adding a course and so on. Finally, the system generates various reports, such as all courses taught by a certain lecturer.

Difficulty: low–medium
Number of students: three

**EM** The theater Events Management project. The system manages information on theater events that take place in different halls. Each hall has a certain number of seats, which are divided into "Cheap", "Regular" and "Expensive" zones. Each event is related to a certain event type, such as "stand-up comedy", "theater show", "concert" etc. When a certain event takes place in a certain hall, a price is specified for each seat zone for that event. The system checks that no two events take place in the same hall on the same day.

Customers can browse the scheduled events and order tickets for events. In addition, customers can buy two kind of subscriptions: a subscription to a certain series of events and a subscription to all events that take place in a certain hall over a certain period of time. For each order, the system tries to assign adjacent seats in the specified price zone. In case this is not possible for the requested zone, the system suggests an alternative in other price zone.

For the operator, the system allows to add halls, create new events and assign them to event series. It also allows to view orders statistics for a certain date, for a hall or for a specific event. Finally, the system allows to move events from one hall to another, by preserving the subscriptions of both kinds, and trying to assign adjacent seats for each order in the preferred price zone.

Difficulty: medium–high
Number of students: four

**FNET** The Football Net project. This system manages football league competitions during one season.

At the beginning of the season, the operator specifies the names of participating teams and their home-fields. A team’s home-filed can change during the season.
Then a list of referees is supplied; they will judge this season’s games. The list of referees can also be altered during the season.

For each week, the system automatically generates games schedule, according to the following rules:

- Each team will play two games with every other team, once on its own home-field and once on the other team’s home-field.
- Each referee will judge at most once a game between two certain teams.

Then the operator can manually update the schedule and, finally, approve it. The operator can also feed results for all games that took place during a certain week. For each won game, a team receives three points. For each game that ends with a draw, both teams receive one point each.

The system generates several kinds of reports. Public reports include games schedule for a given week, games results for a given week, and the teams ratings up to the current point in time. Additional system reports are for the operators, and these reports include: the number of games played on each field and the number of games judged by each referee.

Difficulty: medium
Number of students: two

WF  The Web Forums project. This was an attempt to build a system, which manages a set of Web discussion groups with basic functions.

Registered users can discuss various topics, each topic being discussed in a separate discussion thread. A registered user can start a new topic, by opening a new thread and leaving a first message for that thread, or she can view existing topics, read their existing messages and add messages to their threads. A registered user can also search for messages, by specifying keywords from message content or a partial author name.

The system administrator can create new forums, search for messages and remove messages and threads; she can also prevent a registered user from posting additional messages.

Difficulty: low–medium
Number of students: two

Spring 2004 Semester

AU2  The Air Union project extended. In this project, a team received the code and the documentation of the project from the previous semester. We added new requirements beyond those implemented in the previous semester:

- There are three types of flights: daily (departing at the same time every day), weekly (departing at the same day and time every week) and special (one-time flights).
- Each flight has several kinds of seat classes: business class, first class, economy class. Not all classes are present on every flight. For each class on a certain flight, the number of seats in this class is specified.
• Richer set of constraints when searching for routes, e.g., latest arrival time, latest departure time, flying only within a particular class, specifying the maximal number of stops between flights and more.

• The flight agent may order seats on flights of the chosen routes. She specifies the number of seats to order and the desired class.

This feature is especially interesting. Imagine that multiple agents are ordering seats on the same flight. If the number of requested seats exceeds the number of available seats, the system must handle such a situation consistently. Specifically, the number of ordered seats for a certain flight must not exceed the number of available seats on that flight.

• The flight agent may see a report of all her orders.

• A customer (for whom the flight agent orders the flights) may see a report of all orders made for her.

Difficulty: medium–high
Number of students: two

AU3 The same extended Air Union project. This team did not receive the previous semester’s work, i.e., the team implemented the project from scratch.

Difficulty: medium–high
Number of students: three

FITNET The FitNet project. It is required to implement a system that would support a network of fitness clubs.

Each fitness club within the network has two kinds of trainers: aerobics and bodybuilding. The aerobics trainers include treadmills, bicycles and X-trainers. The body-building trainers include chest, tummy, back, legs and arms trainers. Each trainer within a certain club has a unique ID. In addition, a certain trainer can be available or broken.

A trainee registered within the system can build a training plan, which includes the trainers she wish to use and the number of minutes she wish to use each trainer for. The trainee can change its training plan once in a while. All the training plans of a trainee are stored within the system.

In order to perform a training session, a trainee has to connect to the system and to reserve trainers, according to her current training plan. The trainee specifies the desired day she wants to train on and the hour, at which she wishes to start her training session. In addition, the trainee can specify one or more fitness clubs (all belonging to the FitNet network) she wishes to train in. The FitNet system will then present to the trainee several options of performing her training session. The trainer will select a single option, for which the trainers will then be reserved by the system.

The FitNet system tries to reserve trainers for a training session, so that a trainee will not have to wait less than 5 minutes or more than 15 minutes for a trainer. A trainer will be reserved to a trainee for a time, specified in the trainee’s training plan.

Reservations created by FitNet are stored in the system, and a reservation can also be cancelled by a trainee.
FitNet operators can view the trainees’ training plans, reservations and the statuses of trainers at different clubs. The operators can also update the statuses of the trainers, e.g., from broken to available.

**Difficulty: medium**  
**Number of students:** 3

**FNET2** The Football Net project extended. In this project, a team received the code and the documentation of the project from the previous semester. We added new requirements beyond those implemented in the previous semester:

- The competitions in a certain season are held in a league style or in a cup style. When competing in a cup style, a game cannot end with a draw, and the loosing team leaves the competitions. In addition, a game in a cup style cannot be played on a field that is a home-field of one of the participating teams. Each week, the remaining teams are randomly assigned to play one with another by the system. The operator, as before, can manually update the assignments. Rating reports for a cup season specify the remaining teams and those that left the competition.

- Each week, registered users can enter their guesses for the results of this week’s games. When the results of a certain week are published, the registered users can compare their guesses with actual results.

- The system saves the history of played seasons. For each season, the system saves the details of all games played in this season, the final team ratings for this season, the number of games played on each field and the number of games judged by each referee. For each game, all its details are saved: field, teams, referees and the result.

**Difficulty: medium–high**  
**Number of students:** two

**NABOA** The North Alabama Basketball Officials Association (NABOA) project. NABOA is a local organization for providing officials (referees) for scholastic basketball games. The association’s job is to assign two or three officials (depending on the home school’s preference) to officiate every basketball game played at schools, which are in its responsibility.

The required system will store information on all the available officials, including their “out” days, and the time of day, when the official is available at NABOA office. The system will also store the details of all schools, which are in its responsibility, including an approximate driving time from NABOA office to the school.

Games are played at several levels, such as “Middle School”, “Freshman”, “Junior Varsity” and more. For some of the levels the use of three officials is mandated by NABOA rules. Also, for each official the system will store a level of game, that NABOA is comfortable allowing this official to officiate.

The primary functionality of the system is to produce a master officiating schedule for a given period of time. This includes assigning officials for each game in the period, so that the assigned officials can make it to the game. The schedule must be legal, e.g., the game level constrains must be met. Additional system functionalities include adding a new request to officiate a game, adding a new school, adding
a new official, submitting a “voucher” showing the contests, that a certain official has officiated and more. System reports include an officiating schedule for a certain official, a payment report for a certain school (for all the games this school has requested to officiate), a summary of payments for a certain official and more.

Difficulty: medium–high
Number of students: three

MAGIC The MAGIC project. This project has been suggested by the students. The idea was to build a system, that would support a community of Magic card game players. Magic is a trading card game. Imagine the regular chess game, except that instead of six different kinds of pieces, there exist thousands of kinds of cards. Each player has her own card stock. When the game begins, the player selects sixty cards out of her stock to play with.

The cards for the Magic game are sold in fifteen-card packs. Each pack contains a random set of cards. Therefore, each player has a different stock of cards. If a player is interested in a certain card, she either has to buy a lot of packs and hopefully she will get the wanted card, or she has to find another player that has the desired card and buy it or trade it. For that purpose, the MAGIC system has been suggested. It allows each player to keep records of the cards she has (and potentially willing to sell) and to post requests for the cards she wants to obtain. Furthermore, the system allows to search for players that have a certain card and to contact them in order to trade or buy the card. For each card the player has, she can specify the price she is willing to sell it for. Another option is to announce an auction, in which the card will be sold to the highest bidder. Additional system features include a “merchant ranking” and a recommendation system.

Currently, the activities described above are performed through Web forums. For example, the forum of the Israeli Magic community is located at http://www.mtgil.com. Using a forum for such activities is messy, error-prone and leads to several kinds of problems, such as:

- Spelling errors in the names of cards do not allow quality searching.
- Sometimes the users argue about who was the higher bidder at a certain auction.
- When two users agree on a certain transaction, e.g., selling or trading cards, but they do not trust each other, it is required to carefully record all the details of the transaction in order to prevent arguments.

Difficulty: medium
Number of students: three

This project is special, because it has been tested with real users, those of the Israeli Magic community.

Table 7.4 summarizes the hours invested by students into the different projects, during Winter 2003–2004 semester. Table 7.5 presents the same information for Spring 2004 semester.
Table 7.4: Hours invested into projects, as reported by the students (Winter 2003–2004 semester)

<table>
<thead>
<tr>
<th>Project</th>
<th>Requirements and Specification</th>
<th>Implementation Database Design and Verification</th>
<th>Req. and Spec. % of entire effort</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>AU</td>
<td>22</td>
<td>49</td>
<td>31 %</td>
<td>The quality of the project was low-medium. A few bugs where discovered during the checking. The implementation at some places was quite messy.</td>
</tr>
<tr>
<td>CA</td>
<td>86</td>
<td>74</td>
<td>53.8 %</td>
<td>The quality of the project was quite good. Only a few very minor defects were discovered during the checking. The code was clear and easy to read.</td>
</tr>
<tr>
<td>EM</td>
<td>115</td>
<td>597 (including documentation and preparing the final presentation)</td>
<td>16.2 %</td>
<td>All in all, the quality of the project was OK. Some defects were detected during the checking. It seems that this project was considerably bigger than the other projects this semester.</td>
</tr>
<tr>
<td>FNET</td>
<td>Not reported</td>
<td>Not reported</td>
<td></td>
<td>The quality of the project was good. The AL was implemented correctly. The main problem was that the quality of messages presented to the user was quite low. In some cases, no feedback was given at all.</td>
</tr>
<tr>
<td>WF</td>
<td>67</td>
<td>161</td>
<td>29.4 %</td>
<td>The quality of the project was low-medium. A few bugs were discovered during the checking.</td>
</tr>
</tbody>
</table>
Table 7.5: Hours invested into projects, as reported by the students (Spring 2004 semester)

<table>
<thead>
<tr>
<th>Project</th>
<th>Requirements and Specification</th>
<th>Implementation Database Design and Verification</th>
<th>Req. and Spec. % of entire effort</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>AU</td>
<td>48</td>
<td>161, including 40 hours invested into understanding and updating the legacy code</td>
<td>29.8%</td>
<td>The quality of the project was quite good. Only few minor defects were discovered during the checking. The usability of the system could still be improved.</td>
</tr>
<tr>
<td>AU2</td>
<td>40</td>
<td>118</td>
<td>25.3%</td>
<td>The developed system was small and its quality was medium. Several AL-related bugs were discovered during the checking. The main functionality of the system, i.e., searching for flight routes and ordering seats, had quite a good usability.</td>
</tr>
<tr>
<td>FITNET</td>
<td>26</td>
<td>85</td>
<td>30.6%</td>
<td>The quality of the project was medium. Several bugs were discovered during the checking, including some AL-related bugs. The usability of the produced UI’s was, however, quite good.</td>
</tr>
<tr>
<td>FNET</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>This project should be considered a failure. On the author’s opinion, the students were quite unorganized; in many cases they ignored the guidelines given by the author. The implemented system had many untested scenarios, on which it crashed, the code quality was low and the project was submitted very late.</td>
</tr>
<tr>
<td>MAGIC</td>
<td>12 This includes only the preliminary requirements and specification, see Sec. 7.4.3</td>
<td>249</td>
<td>4.6%</td>
<td>See Sec. 7.4.3</td>
</tr>
<tr>
<td>NABOA</td>
<td>39</td>
<td>175</td>
<td>22.2%</td>
<td>The quality of the project was medium. Several bugs were discovered during the checking, including some AL-related bugs. The usability of the produced UI’s was, however, quite good.</td>
</tr>
</tbody>
</table>
7.4.1 Our Impressions from the Student Labs

We had seven hours of lab meetings, in which \textit{WebSI} was introduced to the students. Afterwards, it took most of the students from five up to ten hours to become well-familiar with \textit{WebSI} and start using it. In any case, this is considerably less than it would have taken the students to learn all the relevant Web programming technologies.

Not all the students were able to successfully meet the deadlines suggested by us. For example, several teams submitted the use-case analysis assignment on time, however, their analyses were not quite usable for further project development. It was difficult to some of the students to accept the idea that use-case descriptions should only specify IO of data. Their use-case descriptions in many cases involved UI details as well, such as “this screen opens”, “the user fills that text field”, “the user selects an item out of a combo box”, “I would like a set of buttons here, and the user will make up her mind by clicking one of the buttons” and so on.

The concept of a use-case displayer was also somewhat hard to adopt. Many students kept asking: “What does the user see when our system starts? We must implement some sort of a “main application screen”, mustn’t we? Or perhaps we should design a super-use case, that will present to the user a set of available use cases, and will dispatch her to the desired use case.” It was quite difficult for them to accept the idea that they have only to design and code the use cases, but they do not have to implement an environment, that enables the user to execute their use cases. A ready-made use-case displayer is such an environment.

On the other hand, there were teams that understood our software construction vision quite well. Their use-case analyses were to the point, specifying only the IO of data. Only minor corrections were required to the use-case descriptions, mainly because of \textit{WebSI} limitations in its set of IO methods. For example, the students wrote something like: “Optionally, the user specifies a date.” But that release of \textit{WebSI} did not support optional inputs, therefore, a work-around had to be found. Except such issues, it was possible to proceed right away and code the use cases. Limiting themselves to the AL-related issues in specification and implementations, those teams were mostly able to meet all our deadlines.

Most of the teams reported that writing test cases at the early stages of use-case specification helped them to validate their use cases.

The final presentations were, of course, not all perfect. There were some teams that could not meet the presentation deadline. They received some additional time, as in Tab. 7.3. In some cases, bugs were discovered during the demonstrations.

On the other hand, there were some very successful presentations. The systems were, in our opinion, quite usable. The students demonstrated test cases, which convinced us that the system implements the requirements correctly. Our feedback for these presentations included only minor corrections and modifications. Most of the after-presentation time for these teams was devoted to writing the documentation for their projects.

7.4.2 Students Impressions from the Labs

In this section we bring some of the students’ feedbacks from both labs we held.

\textbf{The Air Union project (AU), Winter 2003–2004 semester}

- The suggested software development process has a potential of being less error-prone than traditional processes. The reason is that many technicalities are hidden
from the programmer.

- It would be nice if interaction methods could receive parameters. Currently, in order to pass a value to an interaction method, it is required to store the value in a data member of a use-case class, and to read this value from the code of the relevant interaction method.

- The \texttt{<<include>>} relationship support is useful. It allows to capture some common behavior. If changes are required to this common behavior, they are performed in a single place.

- The connection between the interaction methods is by supplying textual method names. (For example, the \texttt{Action.action} method receives the name of an interaction method in its first parameter.) This is error-prone, since if, for example, a name of an interaction method changes, all the places, in which this name is mentioned, must be updated.

- The interaction methods reflect well the IO sequences in the use-case description.

\textbf{The Course Assignments project (CA), Winter 2003–2004 semester}

- The software development process supported by \texttt{WebSI} is quite impressive. It seems to be applicable to bigger systems as well.

- The process does not seem to decrease the amount of bugs. There will always be bugs, that can be discovered only at the code level.

- The suggested process decreases the amount of time devoted to coding. This is because most of the thinking is done at the specification stage and not at the implementation stage.

- The process is not appropriate for very small systems. For such systems, it may be desirable to skip the specification stage and to dive right into coding. For systems that are bigger, the specification stage is crucial.

- For our project it was indeed possible to implement each use case in a separate component. For bigger systems, this may not work. Especially, this may not work in reactive systems, in which the tasks do not have predefined start and end points.

- \texttt{WebSI} allows a fast feedback. In minutes it is possible to set up a basic IO application working over the Web.

- The fact that a \texttt{WebSI}-based application is independent of a presentational platform is impressive.

- For systems that contain a large number of use cases, the use-case displayer component should allow hierarchical presentation of the use cases. That is, it should be possible to group use cases in packages and to arrange a hierarchy between the packages.

- The supported database manipulation routines are sometimes insufficient. It would be desirable to have advanced database services, such as updatable SELECT query results, similar to those supported by JDBC.
• At present, a transaction spans a single IO sequence. “Long” transactions, spanning several IO sequences, are sometimes desirable.

• The <<include>> support was useful. We used the <<include>> relationship during the specification stage and were able to code it at the implementation stage.

The theater Events Management project (EM), Winter 2003–2004 semester

• WebSI at present is not mature enough to be used for big systems. For example, it is difficult for several people to work in parallel on the implementation of the same use case.

• The suggested process saves development effort and is less error-prone than traditional processes, because of the ready-made components that can be assumed to be correct.

• The <<include>> support can, in principle, be dropped.

The Web Forums (WF) project, Winter 2003–2004 semester

• Mapping each use case to a separate component allows to implement these components in a quite independent manner.

• When several use-case components require to share a certain functionality, it is difficult to share the code of this functionality. Therefore, there is a danger of code duplication among the use cases.

• The <<include>> relationship is very useful, probably the most useful WebSI feature.

• It seems there is still a long way to go until WebSI can become practical for real-life systems.

The FitNet project, Spring 2004 semester

The team compared their project to a yearly project they performed with another two students, e.g., five students total. In the yearly project, a system for supporting multiple-choice exams has been developed. The software process in a yearly project was the traditional waterfall process, supported by UML.

The students estimated the other project to be about 100% larger than the FitNet project. Considering that the other project was performed by five students during two semesters, while the FitNet project was performed by three students during one semester, we can estimate that WebSI saved them about 40% development effort. Although this estimation can be very inaccurate, it gives some confidence that WebSI can substantially save development effort.

The WebSI advantages pointed by students included the absence of UI development effort, which was about 25% of the yearly project’s effort. In addition, the students pointed that WebSI enabled them to focus on the AL, that it was quite reasonable to check that the AL code corresponds to the use-case specifications. They also thought that the UI automatically generated by WebSI can be learned quite easily by all the users. As for WebSI disadvantages, the students pointed that with current WebSI UI facilities, it would not be possible to implement the UI of the yearly project.

Additional points mentioned by students were:
• The approach of implementing each use case in a separate component contributes to verification, i.e., its easier to ensure the correctness of the separate components and their relationships.

• The support of the <<include>> relationship is successful, it is easy to use and contributes substantially. The ability to pass values between a base use case and an inclusion use case is especially useful.

• WebSI may not support well the construction of big systems, in which there are many complex relationships between use cases and actors. In such systems the design stage cannot be skipped.

• Employing special exceptions to monitor database constraints violations is useful.

• WebSI was easy to learn, even without the lab meetings.

• The introductory lab meetings should include guidelines for database design.

The Air Union project extended (AU2), Spring 2004 semester
The team compared their project to a project in a natural languages processing lab. The students pointed that UI development was quite laborious in the other project, and that they were happy to be able to avoid it in WebSI. As we have mentioned, this team had to extend a project performed during Winter 2003–2004 semester. The required extensions were very comprehensive, and still it was easy for the students to understand the legacy code, because its structure obeyed the restrictive WebSI rules. The students also expected their UI to be reasonably fast learned by new users.

The NABOA project, Spring 2004 semester

• The direct translation of a use-case specification to code helps to avoid AL-related bugs.

• When a bug is detected during testing, locating it in the code is easy: it is clear in which use-case class and in which interaction method to look.

• The <<include>> support is useful, it helps to avoid code duplication. There are, however, cases, in which it is still difficult to share code between the different use-case components, even with the supported <<include>> relationship. In these cases code duplication can be significant.

• The WebSI-based programming is very structured and standardized. After coding your first use-case class, it is easy to proceed to other use-case classes.

• Extensions and modifications of a WebSI-based system are relatively easy, because each use case is coded separately. Furthermore, the standard structure of a use-case class makes all such classes look similar in some way and contributes to understanding their code.

• The available IS’s are not satisfactory for systems that have many use cases.

• A semantics that enables the user to supply multiple user-composed values in a single input request would be useful.
• It is nice that the programmer does not have to validate the type of input values supplied by the user.

• It is nice that the system handles the multi-user-related issues automatically.

7.4.3 The MAGIC Project

As already mentioned, this project was different from all other projects:

• It was developed to in order to satisfy real needs of an existing community.

• It was tested with real users.

The requirements for the project were suggested and elicited by the project team. Based on the requirements, a preliminary use-case specification was developed and coded. Once the students decided the project was in good shape, they deployed it on an application server and told some of their friends from the Israeli Magic community forum to try using it. Although the first users found some bugs and suggested many improvements, they still were able to successfully utilize all the system’s services. The students improved the system according to the users’ feedbacks and deployed it again. Several such iterations were performed. At late iterations, the students told their friends to “abuse” the system as much as they could. This resulted in discovering some additional defects.

During the testing, the students tried several different IS’s and finally picked the one that suited their system the best, on their opinion. This was the “sky” theme IS (see Appendix B).

All in all, the MAGIC project turned into a modest-size but quite realistic Web application, performing several non-trivial activities and serving the real needs of real users. The most important result for us was that WebSI provided all the required functionality for constructing such an application. A special attention was devoted to the fact that the students’ code had no tricks or work-arounds, and all the required services were provided by WebSI and not implemented by the students. Although the MAGIC team suggested many improvements to WebSI, based on their experience and on their users’ feedbacks, they said that all these improvements are by and large “decorative” and all the required core functionality is already present in WebSI.

Additional points mentioned by the students were:

• There does not seem to be any special limitation on the size of systems constructed with the help of WebSI.

• It is very important to learn thoroughly all the WebSI functionalities. But once you have learned them, it definitely saves development effort, about 30–40% on our estimation. In addition, the suggested process decreases the amount of bugs, because of the standard appearance of the code. It is also easy to add new use cases to the system.

• The drawbacks of the methodology are the lack of flexibility, not being able to perform some non-standard IO activities and not being able to have a full control over the UI.

• It is desirable to have input semantics, which would allow the user to supply tabular data, rather than individual values.
The produced UI made it possible for the users to perform all the required tasks. However, there were also many complaints from the users about the lack of UI flexibility.

The most remarkable WebSI advantages are: producing the functionally correct UI’s automatically, the WebSI security model and the automatic validation of user input values. These three features saved us much of the development effort.

We employed the <<include>> relationship for functional decomposition of some of the use cases. Such practice is in general discouraged by UML, since a use case should be an atomic service of a system, but for our application this seemed appropriate. It enabled us to establish many connections between the use-case components, which resulted in our system being similar to realistic Web applications.

Fetching the input values supplied by the user is only possible in an interaction method, which is invoked right after the values are supplied. If it is required to use a certain value later, it must be saved in a data member.
Chapter 8

*WebSI* Limitations and Desired Improvements

The *WebSI* tool at present cannot and should not be considered a mature commercial product. It is merely a research prototype, developed in order to demonstrate the feasibility of the approach. As a result, applications constructed by *WebSI* have a number of limitations. Some of these limitations are technical by their nature and can be overcome with a reasonable development effort. Others require additional research and investigation.

In this chapter we discuss the various *WebSI* limitations and suggest concrete improvements to *WebSI*, based on our observations and student projects. Some of these suggestions can be implemented right away with a modest effort, while others may require additional planning and experimentation.

- When the user of a *WebSI*-based system uses her Web browser’s ‘Back’ and ‘Forward’ controls, there are cases, in which the system can reach an inconsistent state or a run-time error can occur. The reason is that these controls allow the user to operate the system from outdated Web pages, cached by the browser. These outdated pages do not reflect correctly the state of the database and the state of the appropriate use-case object, as defined by the current values of its data members.

  This problem is by its nature a pure Web-related issue and it occurs also in commercial Web applications, in which it is usually solved by invalidating some of the outdated cached pages. A possible solution is to employ an IS, which is, e.g., based on a *WebSI*-dedicated desktop client application. At present, however, all our available IS’s are designed for a standard Web browser. It must be noted, that in some cases, a properly written *WebSI*-based code can prevent inconsistencies, but we cannot suggest at present a general methodology, which would avoid such errors. In our student projects we instructed the students to assume that ‘Back’ and ‘Forward’ browser controls are not used by the users.

- *WebSI*-based systems in many cases assume that the user navigates the system only as guided by the generated UI. Consider, for example a use case, which has two interaction methods—`start()` and `foo()`. The use case execution starts in the `start()` method, and this method enables the user to initiate an action, which would execute the `foo()` interaction method. So the user cannot execute the `foo()` interaction method, prior to starting the use-case execution. However, a malicious user may, in principle, attack the system by manually constructing an HTTP request, which executes any interaction method of any use case, sending any desired input.
parameters. Such an HTTP request would have to be constructed as described in Chap. 6. In this way, the attacker may exploit the system for her personal benefit, or lead the system to an inconsistent state.

At present we ignore this problem, but a commercial product must, of course, deal with it.

- When the user inputs a string, which contains single quotation characters (‘’), and this string is to be used in an SQL query, a database error occurs. The reason is that SQL syntax employs single quotation characters for literals, such as string literals, time literals and date literals. A quotation character that is not a part of SQL syntax should be escaped with another single quotation character, like this: ‘’.

However, WebSI handles SQL queries after the programmer constructs them, i.e., an SQL query is accepted by WebSI’s DB.read and DB.write methods as a single complete string. In such a complete string, WebSI cannot determine which single quotation characters are part of SQL syntax and which should be escaped. Automatically escaping single quotation characters in all user-supplied strings is not a solution either, since some of them may not be used in SQL queries and should remain as-is.

At present we choose to ignore this problem.

- The mechanisms of user input validation in current IS’s are slightly uncomfortable. At present, each input field is designed to hold only legal values at any time. If the user supplies an illegal value, the system forces the user to correct it (or automatically replaces it with some default legal value), even if the user at the moment does not want to continue operating the use case, e.g., the user decides to initiate some other use case or to switch to another window on his desktop. The preferred solution would be to check that input values are legal, when the user initiates some action, which continues the execution of the current use case. This solution can be implemented by reworking the existing IS’s.

- Some of the WebSI services rely on identity of textual strings to work correctly. For example, an input request is associated with a fetch ID, which is a textual string. An identical string must then be used in order to fetch the value, supplied for this input request. Another example is the action initiation request, Action.action, which receives the name of an interaction method to be invoked. Relying on text identity may be error-prone: the programmer may misspell a fetch ID, or the name of an interaction method may change. In that case, the error can detected only at run-time. Other association mechanisms, which do not rely on text identity, and allow detecting errors at compile-time, may be very useful. It should, however, be noted that any alternative solution must primarily strive for programming simplicity.\(^1\)

- When WebSI fails to commit a transaction, it simply sends an error message to the end-user. In that case, the user is asked to try to execute the use case again. It may, however, be desirable to be able to react to such a situation in the code.

- Current IS’s employ a very simple layout technique: they arrange the various UI elements (those generated by an interaction method execution) one under another.

\(^1\)In the latest WebSI edition, fetch variables, which do not rely on text identity, were introduced instead of fetch ID’s.
We believe that developing more sophisticated layout algorithms may be interesting and useful. For example, an algorithm may group and place together subsequent elements of the same kind (e.g., put all subsequent buttons in one row and align their sizes), allocate more horizontal space for tables that have many columns and so on.

- The current set of supported input and output (IO) semantics can be improved. For example, the following semantics may be useful:
  
  - Enable the user to specify multiple user-composed values in a single input request. The values may be un-typed or of a certain type.
  - Enable the user to specify one or more values for each row of a table. The table may be a result of an SQL SELECT query. The values may be user-composed or selected from predefined sets.

  To support such semantics, the Fetch activity class has also to be modified. It has to include a method that returns a Table.

- Currently a database transaction spans single IO sequence. It may be interesting to introduce “long” transactions that span several IO sequences. For example, a whole use case may be associated with a single, long database transaction. The transaction is committed only when the use-case execution ends. In that case, between two IO sequences of a certain use case on behalf of the same user, the programmer can assume that the database stage does not change. Such approach may eliminate some of the exceptional situations, which originate from the fact that multiple users operate the system concurrently. As a result, the programmer’s code may become shorter.

  On the other hand, long transactions may introduce performance issues.

- For systems having many use cases, it may be desirable to organize the use cases into an hierarchy, rather than in a flat list as done now. Possible solutions include, as suggested in Sec. 3.8, grouping use cases into UML packages hierarchy, employing multiple use-case displayers and so on.

  At present, all the use-case components of a WebST application employ the same IS. It may be suggested to separately associate each use case with its own IS (at deployment time). An alternative suggestion is to associate a package of use cases with a certain interaction style, while individual use cases inside this package (or other packages inside this package recursively) override and redefine the parent IS, with the one more suitable. As a result, more flexible and more usable UI’s may be produced. For example, domain experts may reason that a certain use case is expected to output tables that are “wide”, i.e., have many columns. For such use case, the experts then may suggest to employ an IS that is especially designed to handle wide tabular data. Such a special IS may, however, not be successful for “narrow” tables. Therefore, other use-case components will be associated with other IS’s.

  On the other hand, such approach may lead to developing IS’s, which are tailored to a specific use case and, therefore, are difficult or even impossible to reuse in other use cases or in other applications.²

---

²The latest WebST edition fully implements the features discussed in this bullet. However, since we do not have enough experience with these new features, we cannot yet report on their usefulness.
Chapter 9

Suggested Future Research

9.1 Use Case-Oriented Testing

A well-known popular way of performing system verification is by testing. During testing, predefined test scenarios are executed, and the actual outputs are compared to the expected ones. In software development processes, which employ use-case specifications, these specifications also serve as a starting point for deriving test scenarios.

Further research may investigate the following challenge: given a use-case specification only, manually produce test scenarios, which may further be automatically executed, after the system or a part of it is constructed. We suggest to attack this challenge using the IS approach. That is, it is suggested to construct an IS, which, instead of communicating with a human user, will communicate with a file that contains test data in a predefined format. The same AL code is employed both for testing and (after it passes the tests) for production. No changes or recompilations to the AL code are required; the AL has only to be combined with an appropriate IS. The IS for testing can be especially useful for automatic regression tests, when changes are made to the AL.

We believe that the suggested IS for testing can be designed, such that it may interface with the current WebSI. For example, the testing IS may be a client application, which reads the test scenario data from a file, connects to a WebSI application deployed on an application server via HTTP, operates the application by sending HTTP requests based on the test data file, and compares the responses it receives to the desired outcome data.

9.2 Supporting Workflows with UML Activity Diagrams

UML activity diagrams [58] can be employed to describe workflows, which have such features as branches and merges (for conditional computations), forks and joins (for parallel independent computations) and others. Now consider a workflow, in which a faculty member issues a purchase request for a color printer. The request has to be authorized by two financial clerks, which may work in parallel. Finally, the faculty maintenance department performs the purchase and notifies the person who ordered the printer. Such workflow can be specified by a UML activity diagram, whose activity states [58] represent the execution of use cases.

In our opinion, it may be interesting for the future research to develop a tool, based on WebSI, which will support the execution of the UML activity diagrams having use-case activity states. The input for such a tool will be a set of implemented use cases and a set of activity diagram specifications. The tool will then enable the users to execute the submit-
ted activity diagrams, while preserving their semantics. For the example above, e.g., the maintenance department person will not be able to execute the “Perform a purchase” use case, until both financial clerks successfully finish the “Authorize a purchase” use case, each one working on its own instantiation of this use case.

At present, to implement such workflows, WebSTI programmers must manually code the constraints regarding the order and conditions, in which the use cases should be executed. Being able to specify such constraints in a standardized way via UML activity diagrams, and having a tool, which would automatically enforce the constraints given the activity diagrams, is, therefore, expected to significantly reduce the effort required to implement use-case workflows.
Chapter 10

Conclusions

In this project we have developed a tool to support a specification-oriented (SO) software construction methodology, which is based on direct implementation of system specifications. The developed tool is a WebSI application framework, for construction of multi-user, interactive information systems, operating in the Web environment. The WebSI framework enables a direct manual translation of natural language, e.g., English, use-case specifications to Java code. The results of our work have been published in [34].

The software development process, supported by the SO methodology and the developed WebSI framework, starts with requirements elicitation, followed by development and validation of a use-case specification. We also assume the availability of a database schema; in case the schema is not available, it has to be designed. When the use-case specification and the database schema are available, the specification can be directly coded with the help of WebSI. The system’s user interface (UI) is then generated automatically by WebSI. Such common aspects of Web-based information systems as database access, security, communication over the Web and more, are also taken care of by WebSI.

The principal idea of WebSI is to divide a software system into two parts: the application logic (AL) part and the interaction style (IS) part. The code of the AL implements all application-specific data manipulations. The AL communicates with the environment via the IS component, through a declarative, semantic interface between the AL and the IS. A certain IS implements the AL-IS interface for a certain environment, e.g., the Web/HTML environment. The code of the AL part may be considered to be the code of the application. This code does not have to be changed when the current environment changes, e.g., when the application is ported to run on a PDA or a cellular phone. Instead, a new IS, suitable for the new environment has to be employed. There may exist several IS’s for the same environment; they may differ, for example, in their UI properties.

The interface, between the AL and the IS, enables the programmer to specify what is the desired activity, but not how this activity should be implemented in a certain environment. The how is determined by the code in the employed IS. Examples of activities, supported through the interface are:

- Enable the user to select a single item out of a set of items.
- Enable the user to specify a mandatory user-composed value of any type.
- Enable the user to optionally answer a yes-no question.
- Enable the user to initiate an action in the system.
- Perform an SQL SELECT query on the database.
Present a table to the user, which may be a result of an SQL SELECT query.

Perform an update to the database, i.e., an UPDATE, INSERT or DELETE query.

Fetch the identification details of the user.

The set of activities, supported by WebST’s AL-IS interface, has been driven by typical abstractions seen in use-case specifications of interactive information software. This set has a major influence on the capabilities of WebST. We do not claim that our suggested interface is the ideal one. This interface has been constantly improved, during our evaluation of WebST through student projects. We believe that the current interface is quite useful for the domain of Web-based interactive information systems. Other domains may require a different interface. For that purpose, the design of WebST’s activity classes, discussed in Sec. 6.1.7, enables easy modification of the AL-IS interface.

Having a semantic, rather than a syntactic interface between the AL part and the IS part, is, in our opinion, an advantage. Had the interface been a syntactic one, e.g., such as the interface between the Java code and the implementation of the Java Virtual Machine (JVM) for a specific environment, the UI appearances of an application in the different environments would be very much alike. There are, however, cases, in which a different UI is required, because of the different UI facilities of a specific environment (e.g., a small screen size). Having a semantic interface allows us to provide different syntactic implementations, while the preserving the application semantics. For example, the single selection semantics can be implemented by a set of radio buttons on a large PC screen and by a space-saving pull-down menu on a small screen of a PDA.

A special property of a WebST-based code, is its closeness to the natural language use-case specification. When it is possible to employ only SQL for data manipulation, the code is short and resembles the use-case specification literally and by its structure. This enables easier tracing of the specification backwards and forwards, which facilitates verifying that a certain use-case specification statement is implemented correctly. The closeness to the natural language use-case specification is also expected to facilitate checking the correctness of future code modifications.

A WebST-based system is essentially a set of components, each one implementing a single use case. We have taken this approach in order to make the code resemble the use-case specification, but also to investigate the possibility of reducing system extension and modification costs. Adding a new functionality to the system can be expressed in adding a use case to the use-case specification. The effort of implementing this new use case with the help of WebST is, therefore, expected to be independent of the number of already existing use cases. The same is true when a requirement changes and a certain use case needs to be modified. However, as we already mentioned in Sec. 7.1, such ease of extension and modification is difficult to achieve when there are explicit or implicit dependencies between the relevant use cases.

WebST has undergone quite a heavy evaluation in two student labs. The students employed WebST to construct information systems for air travel planning and ordering, football games scheduling, theater events management and more. Some of the systems constructed by students were quite realistic and usable. The observations made with the student projects suggest the usefulness of the SO approach for software engineering training. Students that invested in developing quality use-case specification could deliver a system having good usability with a modest programming effort. Rough estimations point that employing WebST for construction of Web-based information systems saves about 30% of programming effort, compared to constructing such systems with standard tools.
Students also estimated that the SO approach reduces development costs, compared to the traditional object-oriented (OO) approach, which is based on modelling the domain objects by classes.

Additional significant result of the student labs was that understanding a WebSI-based legacy code had been easier than understanding a legacy code, written using traditional programming paradigms. After becoming familiar with system requirements and specification, it was possible to directly track the specification in the code and understand the legacy implementation. It has to be mentioned, although, that understanding WebSI-based legacy code still remained a non-trivial task.

The UI’s automatically generated by WebSI had, on our opinion, acceptable usability. Despite being quite modest in their visual qualities, the UI’s enable the user to perform the required tasks. An interesting result was that in many cases the UI’s were quite easy to learn and use. Since the common wisdom is that the UI of an entire application can not be produced automatically, this result is quite surprising. We believe the reason for this is that WebSI considerably decreases the gap between natural language use-case specifications and the code. Assuming that the specifications are validated for their usability, it may be expected that a WebSI-based system will retain much of this usability through a smaller specifications-to-code gap. Automatic UI manufacturing, as done by WebSI, may save considerable effort, compared to traditional manual UI construction, which is known to be labor-intensive. Further research is, however, required to understand in which cases a satisfactory UI can be produced by our approach.

We believe that the learning curve of WebSI, as reported in Sec. 7.4.1, is acceptable. It must be noted here, however, that a certain change of thinking and programming approach is required. The WebSI programmer should not think of a system from the UI point of view, i.e., she should not try to imagine how certain windows, frames or dialogs will look like. Instead, she must concentrate on the AL of the system, i.e., which inputs should be requested from the user, and how to compute the required outputs from these inputs. Student teams that were able to adopt their thinking in a short time, produced usable, correct systems with a modest implementation effort. It seems, therefore, that a tool like WebSI can be successfully employed for teaching student labs that focus on requirements engineering, specification engineering, application correctness and usability.

Although the student projects cannot be considered as very small, they are significantly smaller than large, real-life systems. Still, we expect that our suggested methodology, supported by tools like WebSI, can contribute substantially to the industry. One such contribution is UI standardization. A certain software manufacturer may develop an IS according to its UI standards. This IS can then be reused in many applications, developed by this software manufacturer. Another possible contribution is the organization of human effort. A software manufacturer may employ UI and usability experts, which will develop IS’s for different environments. These people will not be required to work in tandem with a team that develops a certain application; they may work completely independently. The application team, on the other hand, is not distracted by such issues as low-level UI technicalities, instead it may concentrate on implementing the AL correctly.
Appendix A

Deploying \textit{WebSI} Applications

A \textit{WebSI}-based application is deployed on a J2EE server that supports J2EE 1.4 technology. Currently, we employ Sun Java System Application Server Platform Edition 8, which is free for development and production. We employ this software bundled with the popular NetBeans 3.6 IDE for Java [48]. Besides the server and the development environment, the bundle also includes a plug-in, which allows an easy operation of the J2EE server from within the NetBeans IDE. For the database server, we employ IBM’s Cloudscape 4.0.6 database engine. For the operating system we employ Microsoft Windows XP Professional.

In this chapter we will describe the deployment process, which is specific for the environment we use and for the ready-assembled software components we provide. However, any other environment, supporting J2EE 1.4 technology may be employed. In an alternative environment the deployment process will, of course, be different. Therefore, in Sec. A.8 we will give some guidelines for assembling the \textit{WebSI} system in any J2EE 1.4-compatible environment.

A.1 Prerequisites

First of all, it is required to download and install the standard Java platform. We recommend to use J2SE 1.4.2.04 platform, freely downloadable from http://java.sun.com/j2se/1.4.2/download.html.

Next, it is required to download and install the Sun Java System Application Server Platform Edition 8, bundled with NetBeans 3.6 IDE. The download location is:

http://java.sun.com/j2ee/1.4/download.html

From our experience, the Sun Application Server requires at least 512 MB of physical memory.

Now define the following environment variables:

- \texttt{JAVA\_HOME}—contains the path to the root directory of J2SE installation

- \texttt{J2EE\_HOME}—contains the path to the root directory of the Sun Application Server. This directory is located inside the NetBeans IDE installation directory and is called \texttt{SunAppServer8}

- \texttt{PATH}—make sure it contains the \texttt{bin} directory inside the J2SE root directory and the \texttt{bin} directory inside the Sun Application Server root directory
A.2 Setting Up the Database Server

Sun Application Server comes with a Pointbase database engine. For various reasons, we prefer to use the IBM’s Cloudscape 4.0.6 database engine. We provide a “Cloudscape Kit” ZIP, which contains the database engine software, a database named CloudscapeDB, convenience scripts, a command-line database manager tool and a GUI-based database manager tool, named “Cloudview”.

Unzip the kit in any directory with write permissions. A directory named cloudscape is created, in which the CloudscapeDB database and all the other files reside.

To start the database server, execute the startserver.bat file in the cloudscape directory.

To run the command-line database manager, execute the text_client.bat file in the cloudscape directory. Now you can issue SQL queries and tool-specific commands (refer to the provided Cloudscape documentation for details on these commands).

To run the GUI-based manager, execute the GUI_client.bat file in the cloudscape directory. Use the tool’s online help for its various features.

To stop the database server, execute the stopserver.bat file in the cloudscape directory.

The CloudscapeDB directory (located inside the cloudscape directory) contains the contents of the CloudscapeDB database. We will (later) configure the J2EE server to employ this database for WebSE applications. After stopping the database server, it is possible to backup the contents of the CloudscapeDB directory, which will backup the current database state.

Make sure the database server is running prior to moving on.

A.3 Configuring the J2EE Server

Start the J2EE server. You can either perform from a command line:

asadmin start-domain --verbose domain1

or use NetBeans as shown on Fig. A.1.

The J2EE server can be administrated through a Web-based console. Point your browser to the following URL:

http://localhost:4848/asadmin/

Now you must specify a username and a password. Default values (unless you specified other values during J2EE installation) are:

<table>
<thead>
<tr>
<th>Username</th>
<th>admin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Password</td>
<td>adminadmin</td>
</tr>
</tbody>
</table>

We will now add the required Cloudscape libraries to the J2EE server’s CLASSPATH. Go to “Application Server” → “JVM Settings” → “Path Settings”. Locate the “Classpath Suffix” text area and add a full path to four JAR files located in the cloudscape directory: cloudbclient.jar, cloudscape.jar, cloudutil.jar and RmiJdbc.jar, as demonstrated on Fig. A.2. If a message appears to restart the server, restart it and reenter the administration console.

The next step is to create a pool of Cloudscape database connections. Go to “JDBC” → “Connection Pools” and click “New...”. Now enter the values as on Fig. A.3 and click “Next”. Enter the value as on Fig. A.4 and click “Next”. Scroll the page down. Leave only two properties by checking two check boxes, as demonstrated on Fig. A.5:
Now click “Save”.

Click “Ping” in order to test the connection pool (make sure the database server is running). You should see a success message.

We will now create a JDBC resource, that will use the connection pool we have just created. Go to “JDBC” → “JDBC Resources” and click “New…”. Enter the values as on Fig. A.6. Click “OK”.

## A.4 Deploying the DBProxy Component

A *WebSI*-based application consists of two J2EE components. One of them is a J2EE Enterprise JavaBean component and it contains the code of the DBProxy bean. This component is packaged in a single JAR file, called `DBProxy.jar`. The component should be deployed only once, i.e., its code will not change and no redeployments will be necessary.

We will now deploy the DBProxy component. Go to “Applications” → “EJB Modules” and click “Deploy…”. Click the “Browse” button, navigate to the directory with the `DBProxy.jar` file and select this file (see Fig. A.7). Click “Next”. Make sure the values are as on Fig. A.8 and click “Finish”.

Close the administration console.

## A.5 The *WebSI* Web Component

The second component of a *WebSI* application is a J2EE Web component. In order to be deployed, J2EE Web components are usually packaged as WAR (Web Archive) files, which are JAR files having a `war` extension. However, it is also possible to deploy a non-packaged Web component, in case its files are arranged in a correct directory structure.
We provide the code of the *WebSI* Web component in a single directory named *war*. This directory has the correct structure, so that the Web component can be deployed from within NetBeans IDE.

The contents of the *war* directory are presented on Fig. A.9 and they include:

- The various IS files, organized in directories, according to IO activity classes and to IS names
- The various *WebSI* XML Schema files
- The *WebSI* application setup files
- *index.jsp* file, that conveniently redirects a browser, opened by NetBeans when a *WebSI* application is started, to the Engine Servlet
- Deployment descriptors for the Web component
- *WebSI* sources and class files
- Client programmer’s sources and class files
Now, from within NetBeans IDE, “mount” the J2EE API JAR file, so that it will be possible to compile the WebSI sources: go to “Filesystems” tab, right-click the “Filesystems” node, select “Mount”, select “Archive Files”, as shown on Fig. A.10. Navigate to the %J2EE_HOME%/lib directory and select the j2ee.jar file.

Next, in the same way, mount the WebSI war directory, this time selecting “Local Directory” instead of “Archive Files”. NetBeans infers from the directory structure that this is a Web component and automatically mounts an additional directory—war/WEB-INF/classes.

Expand the war directory node, then the WEB-INF node and then the classes directory (see Fig. A.11). You will see a directory called websi, under the classes directory. The websi directory contains all WebSI sources. Client programmer’s Java sources should also be placed under the classes directory. If your files belong to the unnamed
Java package, they should be placed directly in the classes directory. If you create packages, the package directories should be placed under the classes directory.

The simplest way to compile all the sources, those that belong to WebSI and those of the client programmer, is to right-click the WEB-INF node and select “Build All”. Afterwards, when you update a Java source file, you can compile only this file, by pressing F9 or right-clicking this file’s node and selecting “Compile”.

### A.6 Setting Deployment Properties

Prior to executing the WebSI application, you should define various deployment properties, such as:

- Which use cases will be accessible to the user
- The desired setup of security roles
- The desired IS
- ...
This is performed by editing two XML files in the war directory: system_setup.xml and user_setup.xml. The formats of these files are constrained by XML Schema files, called respectively system_setup.xsd and user_setup.xsd. Those schema files define the XML format precisely. For the sake of simplicity, we will explain the formats of these files by examples. After editing one of these files, you can validate its format against the appropriate XML Schema, by right-clicking the file node in NetBeans and selecting “Validate XML Schema”.

### A.6.1 The system_setup.xml File

We will explain by example how this file should be organized. Here is a sample contents of this file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<setup:systemSetup
 xsi:schemaLocation=
 "http://www.cs.technion.ac.il/~lyakasal/websi/SystemSetup system_setup.xsd"
 xmlns:xsi=
 "http://www.w3.org/2001/XMLSchema-instance">  
```

Figure A.9: The contents of the WebSI Web component
Figure A.10: “Mounting” a JAR file in NetBeans IDE

```xml
<isDBProxyRemote>true</isDBProxyRemote>
<debug>true</debug>
<DBVendorSpecificSupportClassName>
  websi.db.Cloudscape406
</DBVendorSpecificSupportClassName>

<interactionStyle>
  <UseCaseDisplayXSL>
    twocolumns_elegant DISP/twocolumns_elegant DISP.xsl
  </UseCaseDisplayXSL>
  <activityClassXSL>
    output_elegant/output_elegant.xsl
  </activityClassXSL>
  <activityClassXSL>
    user_composed_elegant/user_composed_elegant.xsl
  </activityClassXSL>
  <activityClassXSL>
    action_elegant/action_elegant.xsl
  </activityClassXSL>
  <activityClassXSL>
    select_elegant3/select_elegant3.xsl
  </activityClassXSL>
</interactionStyle>

<responseXMLSchema>websi.xsd</responseXMLSchema>
</ssetup:systemSetup>
```

The first XML element under the root element (which is `ssetup:systemSetup`) is the `isDBProxyRemote` element. This element defines whether the DBProxy bean should use a remote or a local access (see Sec. 6.3.1). It’s value should be `true` for a remote access and `false` for a local access. In Sec. A.4 we have deployed the DBProxy bean component as a separate J2EE component, therefore, remote access must be used.
Then goes the `debug` element. It defines whether debug messages should be printed or suppressed. The value of this element should be `true` or `false` accordingly.

The `DBVendorSpecificSupportClassName` should contain the fully-qualified name of the class that implements the `websi.db.DBVendorSpecificSupport` interface (see Sec. 6.3.5 on vendor-specific database support). If no vendor-specific database support is required, the `DBVendorSpecificSupportClassName` element should be omitted.

The `interactionStyle` element defines the IS to be employed. It must contain a single `UseCaseDisplayerXSL` element and then zero or more `activityClassXSL` elements. The `UseCaseDisplayerXSL` element specifies the XSL stylesheet to be used for the use-case displayer `WebSI` component. The `activityClassXSL` elements specify the XSL stylesheets for the various IO activity classes. These stylesheets will be integrated into the use-case displayer stylesheet, as described in Sec. 6.2.1. Currently, `WebSI` has four IO activity classes, for which an XSL stylesheet should be specified: `websi.Output`, `websi.UserComposed`, `websi.Select` and `websi.Action`. The locations of the XSL stylesheets should be given relatively to the `war` directory.

Finally, the `responseXMLSchema` element specifies the name of the XML Schema file, that constrains the `WebSI` response XML files. If this element appears, `WebSI` will use the specified XML Schema file to validate all XML response files, which it generates. This validation takes an extra memory and run-time overhead. To disable response XML files validation, omit the `responseXMLSchema` element.

### A.6.2 The `user_setup.xml` File

We will again explain by example how this file should be organized. Here is a sample contents:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<userSetup:userSetup
```
The first XML element under the root \texttt{usetup:userSetup} element is the \texttt{applicationName} element. It defines the name of a \textit{WebSI} application.

Next goes the \texttt{securityRoles} element. This element is optional and it should be omitted if no security roles are required. Otherwise, it must contain one or more \texttt{securityRole} elements. Each \texttt{securityRole} element defines a security role, by specifying:

- the name of the security role—in the \texttt{name} attribute
- the name of the table in the database, where the usernames and the passwords for this security role are stored—in the \texttt{tableName} attribute
- the name of the column that stores usernames—in the \texttt{usernameColumnName} attribute
- the name of the column that stores passwords—in the \texttt{passwordColumnName} attribute

Finally, there goes the \texttt{UseCases} element. It defines all the application’s use cases. First go the use cases that can be directly initiated by the user. For each such use case, a \texttt{UseCase} element specifies:

- the name of the use case—in the \texttt{name} attribute
• the fully-qualified name of the use-case class—in the className attribute

In addition, if the use case is not public, its UseCase element contains one or more securityRole elements, that specify (in their name attribute) the names of security roles, whose users are authorized to access the use case.

Then go the use cases, that are only <<include>>d by other use cases and not initiated directly by the user. For each such use case, a NotDirectlyAccessibleUseCase element specifies the fully-qualified name of the use-case class, in the className attribute. If the use case is not public, its NotDirectlyAccessibleUseCase element contains one or more securityRole elements, that specify (in their name attribute) the names of security roles, whose users are authorized to access the use case.

### A.7 Launching and Debugging the WebSI Application

First, make sure you have compiled all the sources. Then select the “Runtime” tab and make sure the locally installed J2EE server is the default server, see Fig. A.12. Now right-

![Figure A.12: Setting the default J2EE server](image)

...click the WEB-INF node and select “Execute”. NetBeans will deploy the application on the J2EE server and open the browser window in the correct URL.

Note that the URL of the WebSI application (in case the J2EE server is installed locally) is:

```
http://localhost:8081/websiapp/main
```

NetBeans opens the following URL:

```
http://localhost:8081/websiapp
```
For that purpose, the index.jsp file is provided, located in the war directory. This file redirects the browser to the correct URL.

A typical activity of a WebST client programmer is to update some Java sources, compile them and redeploy the application. When using NetBeans, performing this activity is quite convenient. Just compile your updated sources and launch the application by right-clicking the WEB-INF node and selecting “Execute”.

When you update some non-Java files, e.g., the setup XML files, it is required, prior to launching the application, to rebuild it, otherwise the changed non-Java files are not deployed by NetBeans. Right-click the WEB-INF node and select “Build All”. Then right-click it again and select “Execute”.

In order to debug the application, set breakpoints in the desired places, right-click the WEB-INF node and select “Run in Debugger”.

A.8 Assembling the WebST Application in an Alternative J2EE Environment

In this section we will give some guidelines for assembling the WebST system in any environment that supports J2EE 1.4 technology. The reader is advised to browse the J2EE tutorial [25] for details on various J2EE concepts used throughout this section.

A.8.1 Assembling the DBProxy Component

The DBProxy component contains the code of the DBProxy Enterprise JavaBean. Here are the guidelines for assembling this component:

• A separate JAR file, named DBProxy.jar should be created. The “JAR Display Name” of the JAR file should be DBProxyJAR

• The following files should be added to the JAR file:
  – all files in websi.db package (compiled)
  – the following files (compiled) from the websi.exceptions package:
    CheckViolationException.class, DBConstraintViolationException.class,
    ForeignKeyViolationException.class, NotNULLViolationException.class,
    PrimaryKeyOrUniqueViolationException.class,
    TransactionRolledBackException.class,
    WebSIException.class
  – compiled Table.class and AbstractTableAdapter.class (located in the websi package)

• The “Enterprise Bean Type” should be a “stateful session” bean

• The “Enterprise Bean Class” should be websi.db.DBProxyBean

• The “Enterprise Bean Name” should be DBProxyEJB

• The remote interface should be websi.db.DBProxy

• The remote home interface should be websi.db.DBProxyHome

• The local interface should be websi.db.DBProxyLocal
• The local home interface should be `websi.db.DBProxyLocalHome`

• “Transaction Management” should be set to “Bean-Managed”

• The JNDI name of the bean should be `DBProxy`

The DBProxy bean opens a connection to a database, therefore a “resource reference” that points to the desired database should be defined for the DBProxy component. Here are the parameters of the resource reference:

<table>
<thead>
<tr>
<th>Coded Name</th>
<th>jdbc/DBProxyDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>javax.sql.DataSource</td>
</tr>
<tr>
<td>Authentication</td>
<td>Container</td>
</tr>
<tr>
<td>Sharable</td>
<td>yes</td>
</tr>
<tr>
<td>JNDI Name</td>
<td>The JNDI name of a JDBC resource that represents the desired database. This name is defined by a J2EE server. In Sec. A.3, we have defined a JDBC resource, named <code>jdbc/WebSIAppDB</code> that points to the database, named <code>CloudscapeDB</code>.</td>
</tr>
<tr>
<td>User Name</td>
<td>The username and the password required to access the desired database. These parameters are usually defined by the database. For the <code>CloudscapeDB</code> database, the username and the password are empty.</td>
</tr>
<tr>
<td>Password</td>
<td></td>
</tr>
</tbody>
</table>

All the other deployment settings, such as security settings, should be left at default values or not specified.

A.8.2 Assembling the `WebSI` Web Component

The `WebSI` Web component represents an HTTP servlet, implemented in the `websi.EngineServlet` class (see Sec. 6.1). Here are the guidelines for assembling this component:

• A separate WAR file should be created for the component

• The “WAR Display Name” should be `WebSIAppWAR`

• All files in the `WebSI` war directory (except the `WEB-INF/sun-web.xml` file\(^1\)) should be added to the WAR file, preserving the directory structure

• The type of the Web component should be “Servlet”

• “Servlet Class” should be `websi.EngineServlet`

• “Web Component Name” should be `EngineServlet`

• A “component alias” should be defined for the Web component. The alias should be `/main`

• Two “context parameters” should be specified for the component:

<table>
<thead>
<tr>
<th>Coded Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>systemSetupXML</td>
<td><code>system_setup.xml</code></td>
</tr>
<tr>
<td>userSetupXML</td>
<td><code>user_setup.xml</code></td>
</tr>
</tbody>
</table>

\(^1\)This file contains Sun-specific deployment settings and is not needed in another environment.
• Two event listener classes should be defined: `websi.EngineServletContextListener` and `websi.EngineServletRequestListener`.

• The “Web context” for accessing the `WebSIAppWAR` component should be `/websiapp`.

The `WebSIAppWAR` component uses the DBProxy bean in order to manipulate the database. Therefore, it is necessary to define an “EJB reference”, that will enable to lookup and instantiate the DBProxy bean when needed. The parameters for an “EJB reference” are:

<table>
<thead>
<tr>
<th>Coded Name</th>
<th>ejb/DBProxy</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJB Type</td>
<td>Session</td>
</tr>
<tr>
<td>Interfaces</td>
<td>Remote</td>
</tr>
<tr>
<td>Home Interface</td>
<td><code>websi.db.DBProxyHome</code></td>
</tr>
<tr>
<td>Local/Remote Interface</td>
<td><code>websi.db.DBProxy</code></td>
</tr>
<tr>
<td>Bean’s JNDI Name</td>
<td>DBProxy</td>
</tr>
</tbody>
</table>
Appendix B

The Interaction Styles Available in WebSI

As discussed in Sec. 6.2.1, the WebSI IS component is composed from several XSL stylesheets:

- A stylesheet that defines the appearance of the use-case displayer
- For each IO activity class, a stylesheet that provides a UI implementation for the IO methods of this activity class

WebSI currently suggests several stylesheets for the use-case displayer:

- twocolumns_disp.xsl
- twocolumns_elegant_disp.xsl
- twocolumns_sky Disp.xsl
- topscreen uc_links disp.xsl
- listbox disp.xsl
- dynamic menu disp.xsl
- imagebuttons disp.xsl

Let us discuss briefly the properties of each one of these stylesheets. The twocolumns_disp is the default WebSI use-case displayer style. It divides the screen into two columns. The left column contains hyperlinks that launch the accessible use cases, and hyperlinks for logging in to security roles and for logging out. Figure B.1 demonstrates this displayer.

The twocolumns_elegant_disp style follows the same layout as the twocolumns Disp, but introduces a different color theme, which we call “elegant”. As we will later demonstrate, WebSI offers a set of IO activity classes stylesheets, which all follow the “elegant” theme and are recommended to be used with this displayer. Figure B.2 demonstrates this displayer.

The twocolumns_sky disp style also follows the same layout, but employs a “sky” design and color theme, for which WebSI also has a set of IO activity classes stylesheets. Figure B.3 demonstrates this displayer.

The topscreen uc_links disp style follows a different layout. The hyperlinks that launch the use cases, and those that perform “Login” and “Logout” operations are located at the top of the screen. Figure B.4 demonstrates this displayer.
Figure B.1: The twocolumn\_disp displayer, before “Login” is performed (top) and after the “Login” (bottom)

The listbox\_disp style employs list boxes to select the use case to be launched, and to select the security role to log in for. Figure B.5 demonstrates this displayer.

The dynamic\_menu\_disp style employs two menus located at the top of the screen. One menu is for selecting a security role to log in for, another is for selecting a use case to launch. Figure B.6 demonstrates this displayer.

Finally, there is the imagebuttons\_disp style. It employs large legible buttons for launching the use cases and for logging in and out to and from the security roles. Figure B.7 demonstrates this displayer.

In the following sections, we will demonstrate all the currently available IO activity classes stylesheets. For each stylesheet, we will demonstrate its UI implementation of all the IO methods of the correspondent IO activity class. We will also mention whether this stylesheet belongs to a certain design and color theme.
Figure B.2: The twocolumns_elegant_disp display, before “Login” is performed (top) and after the “Login” (bottom)
Figure B.3: The `twocolumns_sky_disp` displayer, before “Login” is performed (top) and after the “Login” (bottom)
Figure B.4: The `topscreen_uc_links_disp` display, before “Login” is performed (top) and after the “Login” (bottom)
Figure B.5: The listbox_disp displayers, before “Login” is performed (top) and after the “Login” (bottom)

Figure B.6: The dynamic_menu_disp displayers, before “Login” is performed (top) and after the “Login” (bottom)
Figure B.7: The `imagebuttons_disp` displayer, before “Login” is performed (top) and after the “Login” (bottom)
B.1 The Output Activity Class Stylesheets

B.1.1 The output_default Style

This is the default style for the webi.Output activity class. It presents tables as tabular structures with columns and rows. It can be used with any use-case displayer style.

Figure B.8 demonstrates how Output.scalar and Output.table methods are implemented.

<table>
<thead>
<tr>
<th>AUTHOR</th>
<th>TITLE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Webster Masterson</td>
<td>Web Components for Web Developers</td>
<td>17.99</td>
</tr>
<tr>
<td>Kevin Novation</td>
<td>From Oak to Java: The Revolution of a Language</td>
<td>29.95</td>
</tr>
<tr>
<td>Gosling James</td>
<td>Java Intermediate Bytecodes</td>
<td>45.3</td>
</tr>
<tr>
<td>Itzai Tru</td>
<td>Duke: A Biography of the Java Evangelist</td>
<td>23.35</td>
</tr>
<tr>
<td>Lykas Alexander</td>
<td>My Trip to Bari</td>
<td>59.99</td>
</tr>
<tr>
<td>Tom Cruise</td>
<td>How I became a star</td>
<td>30.25</td>
</tr>
</tbody>
</table>

Figure B.8: The output_default style, implementing the Output.scalar and Output.table methods
B.1.2 The output_default2 Style

This style employs the “fat” design theme. It presents tables as sets of records. It can be used with any use-case displayer style.

Figure B.9 demonstrates how `Output.scalar` and `Output.table` methods are implemented.

![Table and Figure B.9](image)

Figure B.9: The output_default2 style, implementing the `Output.scalar` and `Output.table` methods
B.1.3 The output_elegant Style

This style employs the “elegant” design and color theme. It presents tables as tabular structures, which are sortable by any column. It is designed to be used with displayers that are compatible with the “elegant” theme, which are `twocolumns_elegant_disp`, `dynamic_menu_disp` and `imagebuttons_disp`.

Figure B.10 demonstrates how `Output.scalar` and `Output.table` methods are implemented. The table has been sorted by the “Price” column in descending order.

<table>
<thead>
<tr>
<th>AUTHOR</th>
<th>TITLE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lyakos Alexander</td>
<td>My Trip to Berli</td>
<td>69.99</td>
</tr>
<tr>
<td>Gosling James</td>
<td>Java Intermediate Bytecodes</td>
<td>45.3</td>
</tr>
<tr>
<td>Tom Cruise</td>
<td>How I become a star</td>
<td>30.25</td>
</tr>
<tr>
<td>Kevin Novation</td>
<td>From Oak to Java: The Revolution of a Language</td>
<td>29.95</td>
</tr>
<tr>
<td>Itzal Tru</td>
<td>Duke: A Biography of the Java Evangelist</td>
<td>23.35</td>
</tr>
<tr>
<td>Webster Masterson</td>
<td>Web Components for Web Developers</td>
<td>17.99</td>
</tr>
</tbody>
</table>
B.1.4 The output\_elegant\_wide\_tables Style

This style is designed to handle wide tables, i.e., having many columns. It presents them as sets of records, ordered from top to bottom. This style employs the “elegant” design and color theme. The style designed to be used with displayers that are compatible with the “elegant” theme, which are\ twocolumns\_elegant\_disp,\ dynamic\_menu\_disp and imagebuttons\_disp.

Figure B.11 demonstrates how Output.scalar and Output.table methods are implemented.

![Figure B.11: The output\_elegant\_wide\_tables style, implementing the Output.scalar and Output.table methods](image)

Figure B.11: The output\_elegant\_wide\_tables style, implementing the Output.scalar and Output.table methods
### B.1.5 The `output_sky` Style

This style employs the “sky” design and color theme. It presents tables as tabular structures, which are sortable by any column. It is designed to be used with the `twocolumn_sky_disp` display.

Figure B.12 demonstrates how `Output.scalar` and `Output.table` methods are implemented. The table has been sorted by the “Price” column in descending order.

![Table](image)

<table>
<thead>
<tr>
<th>AUTHOR</th>
<th>TITLE</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lyakas Alexander</td>
<td>My Trip to Barbi</td>
<td>60.99</td>
</tr>
<tr>
<td>Gosling James</td>
<td>Java Intermediate Bytecodes</td>
<td>45.3</td>
</tr>
<tr>
<td>Tom Cruise</td>
<td>How I became a star</td>
<td>30.25</td>
</tr>
<tr>
<td>Kevin Novation</td>
<td>From Oak to Java: The Revolution of a Language</td>
<td>29.95</td>
</tr>
<tr>
<td>Itzai Tru</td>
<td>Duke: A Biography of the Java Evangelist</td>
<td>23.35</td>
</tr>
<tr>
<td>Webster Masterson</td>
<td>Web Components for Web Developers</td>
<td>17.99</td>
</tr>
</tbody>
</table>

Figure B.12: The `output_sky` style, implementing the `Output.scalar` and `Output.table` methods
B.2 The Action Activity Class Stylesheets

The \texttt{websi.Action} activity class contains a single method, \texttt{action}. Figure B.13 demonstrates how the various \texttt{Action} activity class stylesheets implement this method.

![Diagram of Action activity class stylesheets](image)

Figure B.13: Implementing the \texttt{Action.action} method by various stylesheets. From top to bottom: \texttt{action_default}, \texttt{action_default2}, \texttt{action_elegant}, \texttt{action_sky}
B.3 The UserComposed Activity Class Stylesheets

B.3.1 The user_composed_default Style

This is the default style for the websi.UserComposed activity class. It can be used with any use-case displayer style.

For UserComposed.any[Optional], UserComposed.anyLong[Optional] and for the UserComposed.secret, this style employs respectively a text field, a text area and a text field that does not show its characters. For mandatory inputs, the style does not allow the text field and the text area to remain empty. In case this happens, a message box with an alert message is presented, and a default value (either specified by the programmer or automatic) is written in the text field or the text area. Figure B.14 illustrates the above.

![Image](image1)

Figure B.14: The user_composed_default style implementing the UserComposed.any[Optional], UserComposed.anyLong[Optional] and UserComposed.secret methods (from top to bottom)

For UserComposed.yesNo method this style employs a check box. For UserComposed.yesNoOptional method—a combo box with three options, ‘yes’, ‘no’ and ‘unspecified’, is employed. Figure B.15 illustrates this.

![Image](image2)

Figure B.15: The user_composed_default style implementing the UserComposed.yesNo and UserComposed.yesNoOptional methods (from left to right)

For UserComposed.time the style employs two combo boxes: one for hours, another for minutes. For UserComposed.timeOptional the combo boxes contain an additional ‘unspecified’ entry. Selecting this entry in one of the combo boxes, automatically selects this entry in the other combo box. The opposite is also true—when both combo boxes
Figure B.16: The `user_composed_default` style implementing the `UserComposed.time` and `UserComposed.timeOptional` methods (from top to bottom)

show ‘unspecified’, selecting a time value in one of them also selects the first time value in another. Figure B.16 demonstrates the above.

For `UserComposed.date` the style employs three combo boxes: for years, months and days. For `UserComposed.dateOptional` the style, in addition, employs a check box. When the check box is unchecked, the combo boxes are disabled. The values in the combo boxes are always correct. For example, if the selected month is February and the selected year is a leap year, the days combo box allows to select the values 1–29. Figure B.17 illustrates this.
Figure B.17: The `user_composed_default` style implementing the `UserComposed.date` and `UserComposed.dateOptional` methods (from left to right)
B.3.2 The user_composed_default2 Style

This style is quite similar to the user_composed_default style, except the following differences:

- the style uses the “fat” design theme
- the UserComposed.yesNoOptional method is implemented with three radio buttons, instead of a combo box with three options
- the UserComposed.time and UserComposed.timeOptional methods implementation employs a pop-up calendar window

Figure B.18 demonstrates the implementation of UserComposed.any, UserComposed.anyLong, UserComposed.secret, UserComposed.yesNo, UserComposed.yesNoOptional, UserComposed.time and UserComposed.timeOptional methods.

![Implementation of user_composed_default2 style](image)

Figure B.18: The user_composed_default2 style implementing the following methods: UserComposed.any, UserComposed.anyLong, UserComposed.secret, UserComposed.yesNo, UserComposed.yesNoOptional, UserComposed.time and UserComposed.timeOptional

Figure B.19 illustrates the implementation of the UserComposed.date and the UserComposed.dateOptional methods. Clicking on a small image, located to the right of the date value, brings a pop-up calendar, in which the user can specify another date value or cancel the operation. The optional input request is accompanied with a check box, as in the user_composed_default style.
Figure B.19: The `user_composed_default2` style, implementing the `UserComposed.date` and `UserComposed.dateOptional` methods.
B.3.3 The user_composed_elegant Style

This style employs exactly the same controls as user_composed_default, but it uses the “elegant” design and color theme. Therefore, this style is intended to be used together with twocolumns_elegant_disp, dynamic_menu_disp and imagebuttons_disp displayer styles.

Figure B.20 demonstrates some of the UserComposed activity class methods implementations by this style.

![UserComposed Activity Class Methods Implementations](image)

Figure B.20: The user_composed_elegant style
B.3.4 The `user_composed.sky` Style

This style differs from the `user_composed.default` style in two factors:

- It employs a pop-up calendar window for `UserComposed.date[Optional]` methods
- It employs the “sky” design and color theme

This style is intended to be used with `twocolumns.sky disp` display style. Figure B.21 demonstrates some of the `UserComposed` activity class methods implementations by this style.

![UserComposed sky style](image)

Figure B.21: The `user_composed.sky` style
B.4 The Select Activity Class Stylesheets

B.4.1 The select_default Style

This is the default style for the websi.Select activity class. It can be used with any use-case displayer style.

Table B.1 describes the controls used by the select_default style for implementing the methods of the Select activity class. Note that different controls are employed, depending on the number of columns in a table, out of which the selection is performed.

Table B.1: Select activity class methods implementations by select_default style

<table>
<thead>
<tr>
<th>Method</th>
<th>Selecting from a Single-Column Table</th>
<th>Selecting from a Multiple-Column Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select.one</td>
<td>A combo box</td>
<td>A tabular structure with a radio button on each row</td>
</tr>
<tr>
<td>Select.oneOptional</td>
<td>A combo box with additional 'unspecified' entry</td>
<td>A tabular structure with a radio button on each row, and an additional radio button, labelled as 'unspecified'</td>
</tr>
<tr>
<td>Select.oneOrMore</td>
<td>A multiple-selection list box, which does not allow unselecting all the entries</td>
<td>A tabular structure with a check box on each row. At least one check box must be checked</td>
</tr>
<tr>
<td>Select.oneOrMoreOptional</td>
<td>A multiple-selection list box</td>
<td>A tabular structure with a check box on each row</td>
</tr>
<tr>
<td>Select.action</td>
<td>A combo box. Selecting an entry in the combo box initiates the required action</td>
<td>A tabular structure with hyperlinks in active cells</td>
</tr>
</tbody>
</table>

Figure B.22 demonstrates the implementations of Select.one[Optional] methods for single-column and multiple-column tables. Figure B.23 demonstrates the implementations of Select.oneOrMore[Optional] methods for single-column and multiple-column tables. Figure B.24 demonstrates the implementation of Select.action method for single-column and multiple-column tables.
Figure B.22: The select_default style, implementing Select.one[Optional] methods, for single-column and multiple-column tables
Figure B.23: The select_default style, implementing Select.oneOrMore[Optional] methods, for single-column and multiple-column tables

Figure B.24: The select_default style, implementing Select.action method, for single-column and multiple-column tables
B.4.2 The select_default2 Style

This style employs the “fat” design theme. The style can be used with any displayer style.

Table B.2 describes the controls used by the select_default2 style for implementing the methods of the Select activity class.

Table B.2: Select activity class methods implementations by select_default2 style

<table>
<thead>
<tr>
<th>Method</th>
<th>Selecting from a Single-Column Table</th>
<th>Selecting from a Multiple-Column Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select.one</td>
<td>A single-selection list box</td>
<td>A set of records with a radio button in each record</td>
</tr>
<tr>
<td>Select.oneOptional</td>
<td>A single-selection list box with additional 'unspecified' entry</td>
<td>A set of records with a radio button in each record, and an additional record, labelled as 'unspecified'</td>
</tr>
<tr>
<td>Select.oneOrMore</td>
<td>A multiple-selection list box, which does not allow unselecting all the entries</td>
<td>A set of records with a check box in each record. At least one check box must be checked</td>
</tr>
<tr>
<td>Select.oneOrMoreOptional</td>
<td>A multiple-selection list box</td>
<td>A set of records with a check box in each record</td>
</tr>
<tr>
<td>Select.action</td>
<td>A list box. Clicking an entry in the list box initiates the required action</td>
<td>A set of records with push buttons for active entries</td>
</tr>
</tbody>
</table>

Figure B.25 demonstrates the implementations of Select.one[Optional] methods for single-column and multiple-column tables. Figure B.26 demonstrates the implementations of Select.oneOrMore[Optional] methods for single-column and multiple-column tables. Figure B.27 demonstrates the implementation of Select.action method for single-column and multiple-column tables.
Figure B.25: The `select_default2` style, implementing `Select.one[Optional]` methods, for single-column and multiple-column tables
Figure B.26: The select.default2 style, implementing Select.oneOrMore[Optional] methods, for single-column and multiple-column tables.
Figure B.27: The `select_default2` style, implementing `Select.action` method, for single-column and multiple-column tables
B.4.3 The select_default3 Style

This style strives to employ GUI controls that require little of screen space. The style can be used with any displayer style.

Table B.3 describes the controls used by the select_default3 style for implementing the methods of the Select activity class.

Table B.3: Select activity class methods implementations by select_default3 style

<table>
<thead>
<tr>
<th>Method</th>
<th>Selecting from a Single-Column Table</th>
<th>Selecting from a Multiple-Column Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select.one</td>
<td>A combo box</td>
<td></td>
</tr>
<tr>
<td>Select.oneOptional</td>
<td>A combo box with an additional ‘unspecified’ entry</td>
<td></td>
</tr>
<tr>
<td>Select.oneOrMore</td>
<td>A multiple-selection list box, which does not allow unselecting all the entries</td>
<td></td>
</tr>
<tr>
<td>Select.oneOrMoreOptional</td>
<td>A multiple-selection list box</td>
<td></td>
</tr>
<tr>
<td>Select.action</td>
<td>A combo box. Selecting an entry in the combo box initiates the required action</td>
<td>A table with hyperlinks in active cells</td>
</tr>
</tbody>
</table>

Figure B.28 demonstrates the implementations of Select.one and Select.oneOrMore methods for multiple-column tables.

Figure B.28: The select_default3 style, implementing Select.one and Select.oneOrMore methods, for multiple-column tables
B.4.4 The select\_elegant Style

This style is similar to the select\_default style, except:

- It employs the “elegant” design and color theme
- It generates tabular structures, which are sortable by any column, in ascending and descending order

Figure B.29 demonstrates the implementations of Select\_oneOptional, Select\_oneOrMoreOptional and Select\_action methods for multiple-column tables.

![Table Example](image)

Figure B.29: The select\_elegant style, implementing the following methods for multiple-column tables: Select\_oneOptional, Select\_oneOrMoreOptional and Select\_action
B.4.5 The select_elegant2 Style

This style is similar to select.elegant, except that it treats similarly both single-column tables and multiple-column tables: it presents them as sortable tabular structures. Figure B.30 demonstrates the implementation of Select.oneOptional, Select.oneOrMoreOptional and Select.action methods for single-column tables.

Figure B.30: The select_elegant2 style, implementing the following methods for single-column tables: Select.oneOptional, Select.oneOrMoreOptional and Select.action.
B.4.6 The select_elegant3 Style

This style also employs the “elegant” design and color theme.

Table B.4 describes the controls used by the select_default3 style for implementing the methods of the Select activity class.

Table B.4: Select activity class methods implementations by select_elegant3 style

<table>
<thead>
<tr>
<th>Method</th>
<th>Selecting from a Single-Column Table</th>
<th>Selecting from a Multiple-Column Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select.one</td>
<td>Two columns of entries with radio buttons. Clicking a radio button colors the appropriate entry</td>
<td>A sortable tabular structure with a radio button on each row. Clicking a radio button colors its row</td>
</tr>
<tr>
<td>Select.oneOptional</td>
<td>Two columns of entries with radio buttons. An additional ‘unspecified’ entry. Clicking a radio button colors the appropriate entry</td>
<td>A sortable tabular structure with a radio button on each row. An additional ‘unspecified’ entry. Clicking a radio button colors its row</td>
</tr>
<tr>
<td>Select.oneOrMore</td>
<td>Two columns of entries with check boxes. At least one check box must be checked. Checking/uncheking a check box colors/uncolors its entry</td>
<td>A sortable tabular structure with a check box on each row. At least one check box must be checked. Checking/uncheking a check box colors/uncolors its row</td>
</tr>
<tr>
<td>Select.oneOrMoreOptional</td>
<td>Two columns of entries with check boxes. Checking/uncheking a check box colors/uncolors its entry</td>
<td>A sortable tabular structure with a check box on each row. Checking/uncheking a check box colors/uncolors its row</td>
</tr>
</tbody>
</table>

Figure B.31 demonstrates the implementation of Select.one, Select.oneOptional, Select.oneOrMore and Select.oneOrMoreOptional methods for single-column tables.
Figure B.31: The `select_elegant3` style, implementing the following methods for single-column tables: `Select.one`, `Select.oneOptional`, `Select.oneOrMore` and `Select.oneOrMoreOptional`
B.4.7 The select_elegant_wide_tables Style

This style has been designed to handle tables having many columns. It employs the “elegant” design and color theme.

Table B.5 describes the controls used by the select_elegant_wide_tables style for implementing the methods of Select activity class.

Table B.5: Select activity class methods implementations by select_elegant_wide_tables style

<table>
<thead>
<tr>
<th>Method</th>
<th>Selecting from a Single-Column Table</th>
<th>Selecting from a Multiple-Column Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select.one</td>
<td>A combo box</td>
<td>A set of records, ordered from top to bottom, with a radio button in each record</td>
</tr>
<tr>
<td>Select.oneOptional</td>
<td>A combo box with an additional ‘unspecified’ entry</td>
<td>A set of records, ordered from top to bottom, with a radio button in each record. An additional ‘unspecified’ record</td>
</tr>
<tr>
<td>Select.oneOrMore</td>
<td>A multiple-selection list box, which does not allow unselecting all the entries</td>
<td>A set of records, ordered from top to bottom, with a check box in each record. At least one check box must be checked</td>
</tr>
<tr>
<td>Select.oneOrMoreOptional</td>
<td>A multiple-selection list box</td>
<td>A set of records, ordered from top to bottom, with a check box in each record.</td>
</tr>
<tr>
<td>Select.action</td>
<td>A combo box. Selecting an entry in the combo box initiates the required action</td>
<td>A set of records, ordered from top to bottom, with hyperlinks for active entries</td>
</tr>
</tbody>
</table>

Figure B.32 demonstrates the implementation of Select.oneOptional method for multiple-column tables. Figure B.33 demonstrates the implementation of Select.oneOrMoreOptional method for multiple-column tables. Figure B.34 demonstrates the implementation of Select.action method for multiple-column tables.
Figure B.32: The *select_elegant_width_tables* style, implementing `Select.oneOptional` method, for multiple-column tables

Figure B.33: The *select_elegant_width_tables* style, implementing `Select.oneOrMoreOptional` method, for multiple-column tables
Caption

Table: Web Components for Web Developers

<table>
<thead>
<tr>
<th>Title</th>
<th>Description</th>
<th>Author</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>A bit heavy to read! Use at your own risk.</td>
<td>Wesley M. Anderson</td>
<td>$17.99</td>
<td></td>
</tr>
</tbody>
</table>

Table: From Oak to Java: The Revolution of a Language

<table>
<thead>
<tr>
<th>Title</th>
<th>Description</th>
<th>Author</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Going further in the exploration of programming language evolution.</td>
<td>Wesley M. Anderson</td>
<td>$29.95</td>
<td></td>
</tr>
</tbody>
</table>

Table: Java Intermediate Bytecodes

<table>
<thead>
<tr>
<th>Title</th>
<th>Description</th>
<th>Author</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definitely not for amateurs!</td>
<td>Wesley M. Anderson</td>
<td>$45.25</td>
<td></td>
</tr>
</tbody>
</table>

Table: The Green Project: Programming for Consumer Devices

<table>
<thead>
<tr>
<th>Title</th>
<th>Description</th>
<th>Author</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Become a consumer device programmer in 24 hours!</td>
<td>Wesley M. Anderson</td>
<td>$9.99</td>
<td></td>
</tr>
</tbody>
</table>

Table: Duke: A Biography of the Java Evangelist

<table>
<thead>
<tr>
<th>Title</th>
<th>Description</th>
<th>Author</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fascinating and thrilling, yet not without a touch of tasteful humor and get one!</td>
<td>Wesley M. Anderson</td>
<td>$12.95</td>
<td></td>
</tr>
</tbody>
</table>

Figure B.34: The `select_elegant, wide, tables` style, implementing `Select.action` method, for multiple-column tables
B.4.8 The `select_sky` Style

This style is similar to `select_default`, except that its tabular structures are sortable and it uses the “sky” design and color theme. Figure B.35 demonstrates some of the `Select` activity class method implementations.

![Select activity class method implementations](image)

Figure B.35: The `select_sky` style, implementing the methods of `Select` activity class
Bibliography


http://www.borland.com/jbuilder/


[10] Cascading Style Sheets
http://www.w3c.org/Style/CSS/

http://java.sun.com/products/ejb/


[17] GTK+—The GIMP Toolkit
http://www.gtk.org/


[20] HyperText Markup Language (HTML and XHTML)
http://www.w3.org/MarkUp/


http://java.sun.com/j2se/

http://java.sun.com/j2ee/

[25] The J2EE 1.4 Tutorial
http://java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html

[26] JDBC Tutorial
http://java.sun.com/docs/books/tutorial/jdbc

[27] Java Foundation Classes (JFC/Swing)
http://java.sun.com/products/jfc/

http://java.sun.com/j2ee/javaserverfaces/

[29] JavaServer Pages technology.
http://java.sun.com/products/jsp/

[30] Java Servlet Technology
http://java.sun.com/products/servlet
[31] JavaScript Central  
http://devedge.netscape.com/central/javascript/

[32] Java Transaction API (JTA)  
http://java.sun.com/products/jta/


[43] Microsoft Visual Studio Home  
http://msdn.microsoft.com/vstudio/


[46] The Message Passing Interface (MPI) standard  
http://www-unix.mcs.anl.gov/mpi/

[47] MPICH-A Portable Implementation of MPI  
http://www-unix.mcs.anl.gov/mpi/mpich/

[48] The NetBeans IDE  
http://www.netbeans.org


[60] SOAP Specifications  
http://www.w3.org/TR/soap/


[62] Tcl Developer Xchange  
http://www.tcl.tk/

[64] XML Markup Languages for User Interface Definition
http://xml.coverpages.org/userInterfaceXML.html

[65] XML Schema
http://www.w3.org/XML/Schema

[66] The Extensible Stylesheet Language Family (XSL)
http://www.w3.org/Style/XSL/
ייזור מונחה-אפטי של מערכות מידת לאינטגרט

חיבור על פרוייקט

לשם مليוי חלקי של הדרישות ולkoneksiות הת嬪ור

מגייסר لم던ים במחסן המחשב

אולכסנדר ליאקופ

הוגש לסנט הקסניא – מכון טכנולוגי לישראל

חיפה

נובמבר 2005

חישון תשמ"ז
הפרוייקט עשוי להיתמר פורפ' על ידי קר darmי bjk
בקולטה למודיע המחשב

אני מודד ל السوري על התמיכת במטביס הנדיב בershitalmoti
תקציר

1. מבוא
   1.1. שפות מחקר רitatיביות וחברה חוגגת
   1.2. תהליך פיתוח התכנה
   1.3. מסגרות
   1.4. תכנית קציבור
   1.5. מתאר החיבור

2. אפיון, תכנית וחברה מחמונחת
   2.1. המ narginת המקורית
   2.2. אפיון, מסגרות מחמונחת
   2.3. אפיון, מסגרות מחמונחת
   2.4. עבודה מחודשת
   2.5. פתרון

3. אפיון, תכנית וחברה מחמונחת
   3.1. מפרט חברה
   3.1.1. מפורט של מחקר
   3.2. מסדר تحتון
   3.3. מחלקת מחקר
   3.4. מחלקת מחקר
   3.5. תוכנה מחודה מחמונחת
   3.6. תוכנה מחודה מחמונחת
   3.7. תוכנה מחודה מחמונחת
   3.8. תוכנה מחודה מחמונחת
   3.9. תוכנה מחודה מחמונחת
   3.10. תוכנה מחודה מחמונחת
   3.11. תוכנה מחודה מחמונחת
   3.12. תוכנה מחודה מחמונחת
   3.13. תוכנה מחודה מחמונחת

4. מודל התכנית הממונחת
   4.1. מחלקת מחקר
   4.2. מחלקת מחקרה
   4.3. ערכים מחודשים
   4.3.1. ערכים מחודשים
   4.3.2. ערכים מחודשים
   4.3.3. ערכים מחודשים
   4.3.4. ערכים מחודשים
   4.4. ערכים מחודשים
   4.4.1. ערכים מחודשים
   4.4.2. ערכים מחודשים
   4.4.3. ערכים מחודשים

מאמר מחזורי

תקנון

1. מבוא
   1.1. שפות מחקר רitatיביות וחברה חוגגת
   1.2. תהליך פיתוח התכנה
   1.3. מסגרות
   1.4. תכנית קציבור
   1.5. מתאר החיבור

2. אפיון, תכנית וחברה מחמונחת
   2.1. המ narginת המקורית
   2.2. אפיון, מסגרות מחמונחת
   2.3. אפיון, מסגרות מחמונחת
   2.4. עבודה מחודשת
   2.5. פתרון

3. אפיון, תכנית וחברה מחמונחת
   3.1. מפרט חברה
   3.1.1. מפרט של מחקר
   3.2. מסדר تحتון
   3.3. מחלקת מחקר
   3.4. מחלקת מחקר
   3.5. תוכנה מחודה מחמונחת
   3.6. תוכנה מחודה מחמונחת
   3.7. תוכנה מחודה מחמונחת
   3.8. תוכנה מחודה מחמונחת
   3.9. תוכנה מחודה מחמונחת
   3.10. תוכנה מחודה מחמונחת
   3.11. תוכנה מחודה מחמונחת
   3.12. תוכנה מחודה מחמונחת
   3.13. תוכנה מחודה מחמונחת

4. מודל התכנית הממונחת
   4.1. מחלקת מחקר
   4.2. מחלקת מחקרה
   4.3. ערכים מחודשים
   4.3.1. ערכים מחודשים
   4.3.2. ערכים מחודשים
   4.3.3. ערכים מחודשים
   4.3.4. ערכים מחודשים
   4.4. ערכים מחודשים
   4.4.1. ערכים מחודשים
   4.4.2. ערכים מחודשים
   4.4.3. ערכים מחודשים

מאמר מחזורי

תקנון

1. מבוא
   1.1. שפות מחקר רitatיביות וחברה חוגגת
   1.2. תהליך פיתוח התכנה
   1.3. מסגרות
   1.4. תכנית קציבור
   1.5. מתאר החיבור

2. אפיון, תכנית וחברה מחמונחת
   2.1. המ narginת המקורית
   2.2. אפיון, מסגרות מחמונחת
   2.3. אפיון, מסגרות מחמונחת
   2.4. עבודה מחודשת
   2.5. פתרון

3. אפיון, תכנית וחברה מחמונחת
   3.1. מפרט חברה
   3.1.1. מפרט של מחקר
   3.2. מסדר تحتון
   3.3. מחלקת מחקר
   3.4. מחלקת מחקר
   3.5. תוכנה מחודה מחמונחת
   3.6. תוכנה מחודה מחמונחת
   3.7. תוכנה מחודה מחמונחת
   3.8. תוכנה מחודה מחמונחת
   3.9. תוכנה מחודה מחמונחת
   3.10. תוכנה מחודה מחמונחת
   3.11. תוכנה מחודה מחמונחת
   3.12. תוכנה מחודה מחמונחת
   3.13. תוכנה מחודה מחמונחת

4. מודל התכנית הממונחת
   4.1. מחלקת מחקר
   4.2. מחלקת מחקרה
   4.3. ערכים מחודשים
   4.3.1. ערכים מחודשים
   4.3.2. ערכים מחודשים
   4.3.3. ערכים מחודשים
   4.3.4. ערכים מחודשים
   4.4. ערכים מחודשים
   4.4.1. ערכים מחודשים
   4.4.2. ערכים מחודשים
   4.4.3. ערכים מחודשים
תוכן עיוני - המשך
4.4.4 ת עשרות במדף הנתונים
4.5 ת meisje בו מות沌ש
4.5.1 אובייקט session מודרים "מophage
4.6 פimsonpaginate של UML
4.7 ת משגירה
4.8 אבטחה
4.9 להחלה
4.10 מהלך Theta
5.1 ת פמידר שיווי
5.2 שיקולי אבטחה
5.3 יוזם המס תוניס
5.4 קירדינ ההוא של מנקד שיווי "הסמטת מגר".
5.5 קירדינ אייננשת של מנקד שיווי "תחים והוים".
5.6 שאר הקוד
5.1 תWSCSI
5.2 מגבנה המגזר
5.3 ת来るיה של
5.4 ת来るיה של XML
5.5 ת来るיה של HTML
5.6 ת来るיה של XSL
5.7 ת来るיה של form
6.1 ת들과רסטית
6.2 תigrations של
6.3 תcomings של DBProxy Bean
6.4 תcomings של DBVendorSpecificSupport
6.5 תcomings של Table
6.6 תcomings של StaticCopyResultSet2TableAdapter

Technion - Computer Science Department - M.Sc. Thesis  MSC-2006-09 - 2006
תוכן עניינים – המשך

6.5.1. סשל בключения (inclusions) והורדת המוכנה  
6.5.2. הריצה של דוגמאות האינטראקטיביות

7. מערכות

7.1. מערכות האורטרית

7.2. מערכות WebSI

7.3.لغישות שלכליות

7.3.1.끳 מدفاعה מוגן אפזויים של WebSI

7.3.2.𝕂יבוליה מבורך צעדים עבורapon מ الدفاعים R–

7.3.3.Bitcoin מدفاعה ווריאליים

7.3.4./frontend

7.3.5.הכליים-CTE,TERESA-

7.3.6.שפחת סרומן לrazione מدفاعה - UIML

7.3.7.שפחת סרומן לrazione מدفاعה- XIML

7.3.8.mansik מدفاعה "פלסיים"

7.3.9.ＤＬＳaporית

7.3.10.很想adeshvahat

7.3.11.מנהשים שלפירותים של מפות

7.3.12.vroletonna של מתכנתה המפות

7.3.13.ורושה של מפות מתכנתה המפות

7.3.14.ורושה של מפות מתכנתה המפות

7.3.15.orchestra

7.3.16.orchestra לשיפורים

8. מגฉบית של שיפורים

8.1. elggשת הפרוארט שיח

8.2. elggשת הפרוארט שיח ב

8.3. elggשת הפרוארט שיח ב

9. גיליונות לשיפורים

9.1. elggשת הפרוארט שיח

9.2. elggשת הפרוארט שיח

10. מסקנות

A. נספח -觖יתת הפרוארט ממוסס

A.1.รกימב

A.2.รกימב

A.3.รกימב

A.4.รกימב

A.5.รกימב

A.6.รกימב

A.6.1. karşורת system_setup.xml

A.6.2. karşורת user_setup.xml

A.7. karşורת karşורת Narratives עם יפורקט

A.8. karşורת karşורת J2EE ו karşורת WebSI

A.8.1. karşורת karşורת DBProxy-ריבב

A.8.2. karşורת karşורת WebSI

B. נספח -Connell האנציקלופדיה של קיריים

B.1. Output גליוןננספח עבורה במלאכת הפיעולית

Technion - Computer Science Department - M.Sc. Thesis  MSC-2006-09 - 2006
תוכן עניינים – המשך

1.1 output_default
1.2 output_default2
1.3 output_elegant
1.4 output_elegant_wide_tables
1.5 output_sky

Action

2.1 UserComposed

3.1 user_composed_default
3.2 user_composed_default2
3.3 user_composed_elegant
3.4 user_composed_sky

3.5 UserComposed

4.1 select_default
4.2 select_default2
4.3 select_default3
4.4 select_elegant
4.5 select_elegant2
4.6 select_elegant3
4.7 select_elegant_wide_tables
4.8 select_sky
רשימת ציורים

14 ארכיטקטורות של מערכות המזהה מודרנית שימור
31 בירור משותף WebSI
35ậmمخוקשה פלטפורמה של WebSI
41Šררשמה סכימה לlogen מודרני
43Šררשמה שעונת גודל מודרני
49‡בירה משותף WebSI
54‡רי ארכיון יומן שימור של מודרנית " Sleeping"
56‡רי ארכיון יומן שימור "ypsum" 5.2
61‡ריצתי על יומן שימור " Sleeping"
68‡ריצתי על יומן שימור " Sleeping"

61.1 EngineServlet המפגש-6.1
62.2 GET POST-1 HTTP WebSI של המפגש-6.2
63.3 מפגש של מפגש שימור-6.3
64.4 USECASEDisplay-6.4
65.5 플랫פורמה של-6.5
66.6 מפגש של מפגש-6.6
67.7 USECASEDisplay-6.7
68.8 מפגש שימור של מפגש שימור-6.8
69.9 USECASEDisplay-6.9
70.10 קריאת של קריאת 6.10
71.11 גניש למסגרת הדרכה-6.11
72.12 מפגש של מפגש-6.12
73.13 בגני התוכן-6.13
74.14 מפגש של מפגש טאפסטריק-6.14
75.15 מפגש של מפגש-6.15
76.16 שחרור של מפגש USECASE לשימוש-6.16

78.17 USECASE לשימוש-7.1
79.18 דיקור ערכים מתאימים לערכים כדי לשימור-7.2
80.19 מפגש של מפגש-7.3
81.20 USECASEATE-7.4
82.21 USECASEATE-7.5
83.22 USECASEATE-7.6
84.23 USECASEATE-7.7

A.1 נספח A של שרת J2EE
A.2 Cloudscape לשנת JARs של שרת J2EE
A.3 מפגש של מפגש השימור-1
A.4 מפגש של מפגש השימור-2
A.5 מפגש של מפגש השימור-3
A.6 מפגש של מפגש השימור-4
A.7 מפגש של מפגש השימור-5

Technion - Computer Science Department - M.Sc. Thesis  MSC-2006-09 - 2006
שם תיוэк - המשך

A.8 החנתת רובוט-ה' שודר

A.9وفق הרובוט-ה' WebSI

A.10לובד התוכנה JAR

A.11לובד התוכנה war-ה'

A.12לובד התוכנה J2EE

B.1מציג שתיколоונים Disp

B.2מציג שתיколоונים Elegant Disp

B.3מציג שתיколоונים Sky Disp

B.4מציג Topscreen uc Links Disp

B.5מציג ListBox Disp

B.6מציג Dynamic Menu Disp

B.7מציג Image Buttons Disp

B.8סגנון המ瘵 תמקה_DEFAULT

B.9סגנון המنزل Default2

B.10סגנון המنزل Elegant

B.11סגנון המنزل Elegant Wide Tables

B.12 سواء Evelyn

B.13Release from "Action.action

B.14Release from "Action.action.

B.15Release from "Action.action.

B.16Release from "Action.action.

B.17Release from "Action.action.

B.18Release from "Action.action.

B.19Release from "Action.action.

B.20Release from "Action.action.

B.21Release from "Action.action.

B.22Release from "Action.action.

B.23Release from "Action.action.

B.24Release from "Action.action.

פונקציית בולIAN - המשך

 Cyprus משק
208

פונקציית בולIAN - המשך
209

פונקציית בולIAN - המשך
210

פונקציית בולIAN - המשך
211

פונקציית בולIAN - המשך
212

פונקציית בולIAN - המשך
213

פונקציית בולIAN - המשך
214

פונקציית בולIAN - המשך
215

פונקציית בולIAN - המשך
216

פונקציית בולIAN - המשך
217

פונקציית בולIAN - המשך
218

פונקציית בולIAN - המשך
219

פונקציית בולIAN - המשך
220

פונקציית בולIAN - המשך
221

פונקציית בולIAN - המשך
222

פונקציית בולIAN - המשך
223

פונקציית בולIAN - המשך
224

פונקציית בולIAN - המשך
225

פונקציית בולIAN - המשך
226

פונקציית בולIAN - המשך
227

פונקציית בולIAN - המשך
228

פונקציית בולIAN - המשך
229

פונקציית בולIAN - המשך
230

פונקציית בולIAN - המשך
231

פונקציית בולIAN - המשך
232

פונקציית בולIAN - המשך
233

פונקציית בולIAN - המשך
234

פונקציית בולIAN - המשך
235
## רישום טבלאות

<table>
<thead>
<tr>
<th>סיווג פעילויות</th>
<th>3.1</th>
</tr>
</thead>
</table>
| WebSI מתרחבות הפיטות עם Multi-
| 3.2 |
| WebSI מתרחבות הקלט | 3.3 |
| תתחודשה Multi-
| 3.4 |
| תתחודשה עם Multi-
| 3.5 |

<table>
<thead>
<tr>
<th>표</th>
<th>4.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL</td>
<td>UPI</td>
</tr>
</tbody>
</table>
התקיף

התקיף הדוסר של הנוסח המודגש והניהול תכנית במציאות ת الإسلامي. הפקת המחкрутת אחר מַשְׁבִּית כר

• הנחלה הנג שмиימה, בכימר תורמת את מכון המשותף ופגישת לול ילבוץ ואת המחקר

• בקר ארוגית של התוכנה וIsActive את הדרישה המいただいて

• רוחות ושרירים של התוכנה ונסיעת במד PublicKey. למל התוכנה והספקת פונקציונליות ושיחה

אני כבר בחירה המטרה

המכרז והתלבט בפרסומים ומברכים את התוכנה.macen החשיבות ההלשנה בשינויים מעוררים לתוכנה זה. והשיטה את המודדים עם מעוררים בחר את

בנוסף, תוצאות מתאימות לכלOURS "משתת" (framework).

Es werden jedoch neue Funktionen und Dienste hinzugefügt, ohne dass die Leistung der Systeme sinkt.

לפי גישה זו, ניסיון

התקיף הכריא

ולפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיон

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גישה זו, ניסיון

לפי גיש
The user chooses

and confirms.

The client sends a message to the server which will perform the operation.

WebSI

is the interaction style.

An interaction can be, for example, the ability to choose an item in the menu.

Another interaction can be, for example, to use a radio button.

The user defines the interaction with the system, for example, the color, the body, the geometric order.

Changing the interaction is a simple operation that does not require recompilation of the code that implements the interaction.

WebSI contains a database of interactions.

The developer can try different interactions to reach the user's goal, which developed the system.

A feature important to WebSI is that it should work in any application.

The result is that the interaction is used in various applications.

It was developed, for example, the ability to add support systems and a water alliance, WebSI.

This can be used for managing information in the system or to control positions in the team, WebSI.

In a special case, it is a project that is called MAGIC.

The system is used to execute transactions and replace the keys, to improve sales and the assessment of the user after the transaction.

The system was built and tested in the Technion Department of Computer Science M.Sc. Thesis 2006-09-2006.

The use of WebSI in the project was essential to the student's work, the credits were sufficient for it.

It was less important for the student to study the technologies that are common when developing interactive systems for many users.

We reported that the student can work on the project and then work on the code as the project advances.

The importance of this is that the code is shortened.

When we examined the code we saw that WebSI is suitable for this project.

We found that the student was able to use the code and the system to implement various functions.

The student was able to write the code of the project.

The system was written based on the code that was written in the project.

This was important for the student's future work.
In the context of WebSI framework, a detailed report on the development and implementation aspects is presented. The study highlights the importance of developing a framework that can effectively address the challenges faced by developers in the field of Web-based applications development.

The WebSI framework is designed to provide a robust and scalable solution for the development of Web-based applications. It is based on a modular design approach, which allows developers to easily integrate new features and functionalities without affecting the overall system performance.

The framework is designed to be extensible, allowing developers to add new modules and components as needed. This flexibility is crucial in the rapidly evolving field of Web-based applications development, where new technologies and standards are constantly emerging.

One of the key benefits of the WebSI framework is its ability to support a wide range of devices and platforms, including mobile devices and desktop computers. This is achieved through a set of APIs and services that enable developers to create applications that are optimized for different devices.

The framework also provides a set of tools and utilities that help developers to streamline their development process. These tools include a set of debugging tools, a comprehensive documentation, and a set of best practices guidelines to ensure that the developed applications are of high quality.

Overall, the WebSI framework provides a powerful and flexible solution for the development of Web-based applications. It is designed to be scalable, extensible, and user-friendly, making it an ideal choice for developers who are looking to create high-quality applications that can be easily adapted to meet the needs of their users.