Available online at www.sciencedirect.com
SCIENCE DIRECT®
Science of Computer Programming 56 (2005) 5–21
Science of Computer Programming
www.elsevier.com/locate/scico

Use-case components for interactive information systems

Eliezer Kantorowitz*, Alexander Lyakas

Computer Science Department, Technion—Israel Institute of Technology, Haifa 32000, Israel

Received 28 November 2003; received in revised form 31 August 2004; accepted 6 September 2004

Abstract

Specification-oriented components (SOC's) are designed to facilitate the implementation of a system directly from its specifications. An earlier study has shown cases in which SOC's enabled information systems to be implemented with considerably less code than when implemented with components designed by a typical object-oriented approach. This study goes a further step by considering the essence of an information system to be the flow and processing of data. The components based on this abstraction attempt to hide code that is not implementing data flow or data processing. Based on this approach, an experimental framework called $\mathcal{WebSI}$ has been developed. $\mathcal{WebSI}$ components hide the code for the construction of the user interface (UI), the database access code and the Web-related code. $\mathcal{WebSI}$ was designed to facilitate the manual translation of English language use-case specifications into Java code. $\mathcal{WebSI}$ enabled the construction of information systems with a modest amount of code. The similarity between the $\mathcal{WebSI}$-based Java code and the English language use-case specifications facilitated verifying that the code implements the specifications correctly. The automatically produced UI's were relatively easy to learn and to use. The modification of $\mathcal{WebSI}$-based legacy code was facilitated by the high level of the code and its use-case structure, but remained a labor-intensive task.

Keywords: Component; Specification-oriented; Specification-oriented component; Use case; Information system; User interface; World Wide Web; Web

* Corresponding author.
E-mail addresses: kantor@cs.technion.ac.il (E. Kantorowitz), lyakasal@cs.technion.ac.il (A. Lyakas).
URLs: http://www.cs.technion.ac.il/~kantor (E. Kantorowitz), http://www.cs.technion.ac.il/~lyakasal (A. Lyakas).

0167-6423/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2004.11.002

1. Introduction

A software component can be defined as a piece of software which is designed such that it can be reused in many different systems. The design of components that meet this requirement has proven to be a difficult task. These difficulties may be alleviated by designing components which are limited to a particular application domain and to a particular method of combining the components into a software system. Such a set of components and the corresponding construction method are sometimes called a component *framework*.

One of the measures of quality of a framework is the developer's effort required to design, implement and test a software system with the help of the framework, i.e., the *development costs*. Another quality measure of a framework is the developer's effort required to extend the software, produced with the help of the framework, at a later point in time, i.e., the *extension costs*. This quality may be expressed by the *extension complexity* [10] of the software produced with the framework. A further quality measure of a framework is whether it facilitates manufacturing software systems that have a high *usability* level. Usability expresses both the extent to which the system provides the functionalities that the users need, and whether the system enables the users to accomplish their work with a minimum of human effort and in a pleasant way. Usability is possibly the most important system property, as it may be the determining factor for whether or not the software will sell.

This paper focuses on designing frameworks facilitating the construction of high-usability software systems at low development and extension costs. Other aspects of component design, e.g., the problems of interfacing components from different sources, are not considered in this paper.

One approach in frameworks design is to use a classical object-oriented (OO) design methodology, i.e., to construct a component for each one of the important objects in the application domain. An example of such a framework is the Java's Swing package for graphical user interfaces (GUI) construction. This package contains components for such GUI controls as buttons, menu entries and text fields. A GUI is constructed by combining these components in a special way. The construction involves the specification of the size of the controls, their colors and the layout on the screen. The programmer must also code the operations carried out by the software, when a particular GUI control is operated by the user. Programming at this level of detail can be quite labor intensive.

Another approach in frameworks design is the *specification-oriented* approach, suggested in [12], where the framework's components are designed to enable a direct coding of the system 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 specification of such a system can be executed [4]. The developer of such a system has thus only to specify it by its state chart, and need not invest further time in designing and implementing the system. The state chart can be executed, because it explicitly specifies all the computations in the system. Our study considers large complex information systems, for which it is difficult to produce such complete specification of the entire system, as done in the state chart approach. A common industrial method to manage this high complexity is to develop such a system through a software development process

(often shortened to *software process*). Each phase of such a software process focuses on one aspect of the development, so that the developers can concentrate on doing it right.

A number of different kinds of software processes have been designed over the last decades. This research was specially influenced by [7], which introduced the *use-case* concept and its applications in software processes. The use-case concept and additional concepts from other software processes were later elaborated into the Unified Software Development Process (USDP) [6], which was developed in connection with the popular standardized Unified Modeling Language (UML) [18].

The activities of a software process may be divided into two major groups. The first is the *specification development* group, which includes requirements elicitation, specification development and validation of the developed requirements and specifications. The second group of activities regards the *construction* of a system that meets the specifications.

The USDP and some other software processes employ use cases to specify the developed system. A use case is a single application of a system. Use cases are often described in a natural language, e.g., English, and may be accompanied by drawings of the user interfaces (UI). We consider an example of a use case in a sales management system. The example use case specifies the addition of a new sales offer by a supplier. The detailed formulation of this use case is:

(1) The system presents the set of items for sale, for which the supplier has not given offers. For each item, its name and description are presented.
(2) The supplier selects a single item 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 items, for which the supplier has not given offers. The specification also provides no information on where and how the details of the new offer are stored. So as opposed to a state chart specification, a use-case specification does not specify how the system computes its outputs from the user's inputs. It is essentially only a specification of the user's input data and the corresponding system's output data.

The use of natural language and UI descriptions enable domain experts and human–computer interaction (HCI) experts, who 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.

The construction activities of a software process involve an analysis of the validated specifications. This includes developing methods for computing the system's outputs from the user's inputs and database values. Based on this analysis, the developers design and code the software. The construction stage 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 will retain this usability.

The construction activities are usually labor intensive, since they involve both specifications analysis, system design, coding and verification. In order to reduce these high construction costs, one may implement the system with the help of high-level reusable software components. Such a set of components was suggested in [12], where the concept of specification-oriented components (SOC's) was introduced. One of the goals of this approach is to facilitate the verification of the code. SOC's are designed to enable a manual translation of the natural language use-case specifications into a high-level code that resembles the specifications. The equivalence between the code that resembles the specifications and the specifications is expected to be relatively easy to establish. The use of high-level code is also expected to save implementation costs and produce a running system at an early stage.

The suggested approach was tested with the experimental framework, named $\mathcal{SI}$ (*Simple Interfacing*) [12], for construction of interactive information systems. In the experiments, the English language use-case specifications were manually translated into $\mathcal{SI}$-based Java code. Typically, 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, first using the Java's GUI Swing framework and the Java's database connectivity package (JDBC). The second implementation employed the components of $\mathcal{SI}$. There were 2.4 times as many lines of code in the Swing- and JDBC-based implementation, than in the $\mathcal{SI}$-based code. This suggests that using components that model use cases, as done in $\mathcal{SI}$, saves coding, as compared to using components that model the objects of the domain in a classical OO design. The suggestion supports the observation made by [15], that a classical OO component design does not necessarily produce the most appropriate components. The remarkable code savings obtained with $\mathcal{SI}$ were achieved mostly by hiding much of the GUI and database access code in the components of $\mathcal{SI}$. A later design of $\mathcal{SI}$, called $\mathcal{SI}+$, is reported in [13,14].

Based on the experience with $\mathcal{SI}$ and $\mathcal{SI}+$, we have developed an experimental framework, called $\mathcal{WEBSI}$ (*Web Simple Interfacing*). $\mathcal{WEBSI}$ is a set of Java components, designed to facilitate the implementation of elementary Web-based information systems. These information systems are assumed to be specified by use cases written in English. The translation of the use-case specifications into Java is done manually by the developer. $\mathcal{WEBSI}$ may thus be employed for the construction activities of the software process, in which it is required to implement a system from given use-case specifications.

2. The architectural principles of the $\mathcal{WEBSI}$ framework

$\mathcal{WEBSI}$ assumes that the essential activities of an information system are the flow and processing of data. The design of $\mathcal{WEBSI}$, therefore, attempts to hide all the code, which is not related to these activities. This hiding enables the developer of the use-case specification and the programmer, who translates the use cases into Java code, to focus on the essence of the information system. In this and the following sections we discuss how the emphasis on flow and processing of data influenced the design of $\mathcal{WEBSI}$ architecture, its UI facilities and database access facilities.

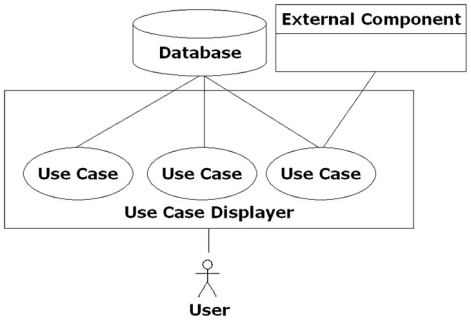

Fig. 1. The structure of $\mathcal{WebSI}$ applications.

Similarly to many information systems, $\mathcal{WebSI}$ employs a single relational database, which stores the *state of the information system*. $\mathcal{WebSI}$ represents the abstraction of a table (relation) by the `Table` interface, enabling high-level manipulation of tables, computed by SQL queries or constructed manually. The database is manipulated by transactions, designed to retain its consistency.

The *usage model* of a $\mathcal{WebSI}$-based information system is that the user initiates a use case and then performs several *input–output sequences*. We define a single input–output sequence as:

(1) The user inputs the data requested by the system.
(2) The system fetches the data supplied by the user.
(3) The system computes the output based on the supplied data and on the database.
(4) The system updates the database if required.
(5) The system presents the output to the user.
(6) The system requests the user to supply additional inputs, if further computations are needed.

When employing $\mathcal{WebSI}$, each use case is implemented by a separate class, called a *use-case class*, which is derived from the $\mathcal{WebSI}$ `UseCase` class. Each use-case class has a number of different *interaction methods*, that contains the translation of the English use-case specification to Java, employing $\mathcal{WebSI}$ components. Each interaction method implements one input–output sequence, and together they implement the interaction between the user and the system, as specified in the English use-case description. Each use-case class must implement an interaction method named `start`, which is invoked automatically by $\mathcal{WebSI}$, when the use-case execution begins. Other interaction methods are invoked by $\mathcal{WebSI}$ according to the user actions and the use-case specification.

A use-case implementation may, besides the $\mathcal{WebSI}$ components, employ components from other sources (see Fig. 1). For example, an external `CreditCardValidator` class may be used to validate credit card details, when processing a customer's order in the sales management information system mentioned above. In a $\mathcal{WebSI}$-based information system, each use case is thus implemented by a class derived from the `UseCase` class and possibly a number of components from other sources.

```
public void start() throws Exception {
1   Table offers =
        DB.read("SELECT offer.offerid, item.name, item.description, offer.price " +
                "FROM offer, item " +
                "WHERE offer.itemid=item.id AND offer.supplierid=" +
                UserID.getUsername());
2   String[] visibleColumns = {"name", "description", "price"};
3   Select.one("selected_offer_id", offers, visibleColumns, "offerid", null,
                "Select the offer you wish to edit:");
4   UserComposed.any("new_price", "0.0",
                "Give the new price for the selected offer:");
5   Action.action("updateOffer", "Update the selected offer!");
6   Action.action("deleteOffer", "Delete the selected offer!");
}
```

Fig. 2. The implementation of the beginning of 'Edit Offer' use case.

It is assumed that a use-case component can accomplish its computations with the data obtained from the user, the database and possibly some external components. Such a use case, therefore, only interfaces with the user, the database and the external components, but not with other use cases. This architecture is illustrated in Fig. 1. The developer of a particular use case, therefore, usually need not know anything about the other use cases, which means that a particular use case is not coupled to any other use-case components. Therefore, the use-case developer usually needs only to be familiar with the use-case specification and the structure of the database schema.

Since a $\mathcal{WebSI}$-based application is essentially a set of use-case classes, we obviously need a kind of a component, which "glues" all the use cases together into one application. For that purpose, $\mathcal{WebSI}$ provides a ready-made internal component, called a *use-case displayer* [14]. The use-case displayer enables the user to initiate and execute the various use cases of the information system. In addition, the use-case displayer enables the user to log in to and log out from the various systems' *security roles*. In order to reduce the possibility of user mistakes, the use-case displayer is designed to allow only the permitted actions. For example, Fig. 3 shows a use-case displayer, after a user, identified as supplier, has already logged in. That is why a 'Logout' hyperlink appears in the upper-left corner of the screen, enabling the supplier to log out. A logged-in supplier is only permitted to execute use cases, relevant to her security role, which are the 'Add Offer', the 'Contractor's Info', the 'Edit Offer' and the 'Sales Statistics' use cases. The execution of a use case is initiated by activating one of the hyperlinks on the left side of the screen. The currently executing use case is 'Edit Offer'; its GUI is presented to the user on the right side of the screen.

3. $\mathcal{WebSI}$ UI facilities

From the $\mathcal{WebSI}$ 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. In contrast to traditional

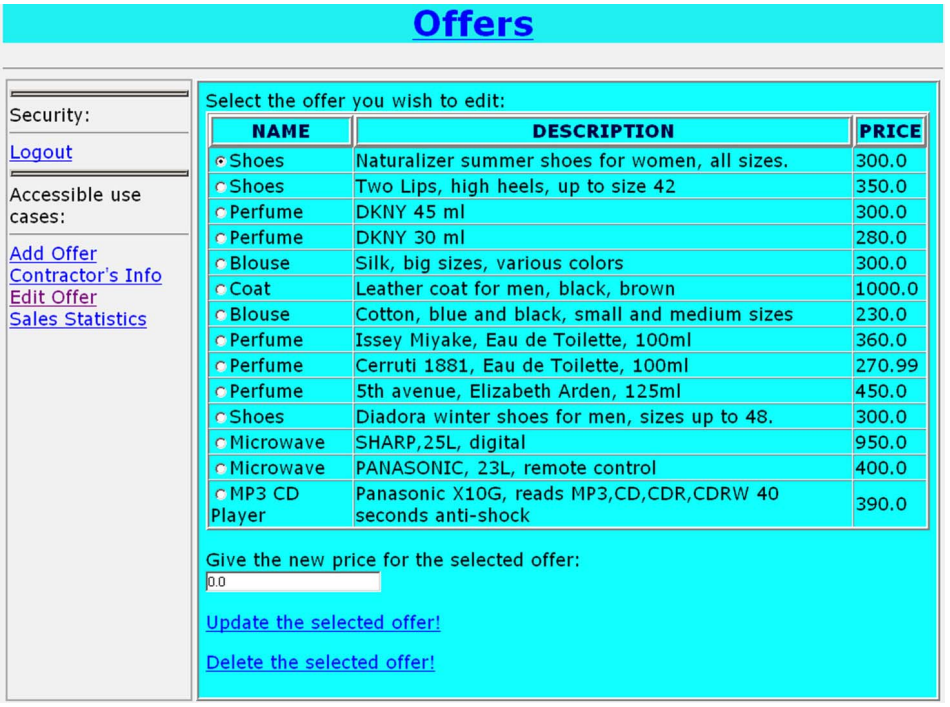

Fig. 3. The beginning of 'Edit Offer' use case.

UI specifications that involve specific UI controls, their layout, colors and so on, *WebSI* employs *semantic UI specifications*. This means that the programmer specifies **what** the UI enables the user to do, i.e., the desired UI functionality. The programmer, however, does not specify **how** this functionality should be implemented, e.g., which specific UI controls should be employed—this is the *syntax*. An example of a semantic UI specification is enabling the user to select a single item out of a set of items. The implementation of this semantics, i.e., the syntax, may involve, for example, a pull-down menu, a single-selection listbox or a set of radio buttons.

The UI that will implement the semantic specifications is determined by a *WebSI* component, called *interaction style* [11]. This component specifies all the UI properties, such as which controls should be employed, their colors, fonts and the layout on the screen. *WebSI* offers a set of ready-made interaction styles. Additional interaction styles can be developed and integrated into *WebSI*. The interaction style to be employed is selected at deployment time, and no changes or recompilations of the use cases implementations are required.

To illustrate these techniques we consider again the sales management information system. The database of this system has the tables "Item", "Supplier" and "Offer". The domains (columns) of these tables are listed below:

- Item (id, name, description)

- Supplier (id, name, city, address, description, password)
- Offer (offerid, itemid, supplierid, price)

The columns, whose names are underlined, are the primary keys of the tables.

The information system has a use case called 'Edit Offer', which enables a supplier to change the price of an offer or to delete an offer. Below is a part of the English language specification of this use case:

- The system presents the set of all the currently valid offers of the supplier that executes the use case. For each offer, item name, item description and price are displayed to the supplier.
- The supplier selects one offer from the set.
  - The supplier specifies a new price for the selected offer.
  - The supplier approves.
  - The system updates the price of the selected offer and presents a success message.

  OR
  - The supplier instructs the system to delete the selected offer.
  - The system deletes the selected offer from the database and presents a success message.

Note that the above description contains no UI details; they will be determined automatically by $\mathcal{WebSI}$.

The programmer translates the above English specification into code of the `EditOffer` use-case class, whose `start` method is shown on Fig. 2. The UI produced by this code is shown in Fig. 3. Let us explain the code of this method in brief.

Line 1 issues an SQL SELECT query, which obtains all the offers of the logged-in supplier that is executing the use case (as determined by the `UserID.getUsername` method provided by $\mathcal{WebSI}$). The query returns a four-column table into a local variable, called `offers`. The type of this variable is `Table`, which is a $\mathcal{WebSI}$ interface that represents tabular data. The `DB.read` method is also provided by $\mathcal{WebSI}$ and will be discussed in Section 4.

In lines 2–3 the `Select.one` method is invoked. This method produces a UI that enables the user to select exactly one row out of the `offers` table, produced by the SQL query in line 1. Note, that the method only specifies the input semantics, but not the UI controls to be employed. In line 2, the programmer specifies that the "`name`", "`description`" and "`price`" columns of the `offers` table will be visible to the user. In the fourth argument to the `Select.one` method, the programmer specifies that the "`offerid`" column of the `offers` table will serve as the *return values column*. That is, for example, if the user selects the third row of the `offers` table, the programmer will later fetch the third value of its "`offerid`" column. The "`selected_offer_id`" string serves as a textual ID, which will be used by the programmer later, i.e., in the next input–output sequence, to obtain the user's selection, like this:

```
...
int selectedOfferID = Fetch.Int("selected_offer_id");
...
```

This statement obtains the user's selection, by fetching the value in the "`offerid`" column of the `offers` table, in the row that has been selected by the user. The `Fetch.Int` method employed here is provided by $\mathcal{WebSI}$ for obtaining the user's input in the integer form.

The fifth argument to the `Select.one` method defines which row should be selected by default. Here, the programmer leaves it unspecified, and $\mathcal{WebSI}$ will select one of the rows.

Additional input methods, used by the programmer on Fig. 2, are: `UserComposed.any`, which enables the user to supply a user-composed value, and `Action.action`, which enables the user to initiate an action in the system. The second argument of the `UserComposed.any` method defines a default input value. The `Action.action` method receives as an argument the name of the interaction method, which will be invoked by $\mathcal{WebSI}$, when the user initiates this action. Again, it should be noticed that the programmer does not specify concrete UI controls; they will be determined automatically by the interaction style component.

Fig. 3 presents the UI produced by the code on Fig. 2, employing the default $\mathcal{WebSI}$ interaction style. Note, that the interaction style presents tabular data as a table, employs radio buttons to implement the single-selection semantics, a text field for supplying a user-composed value, and hyperlinks for the action semantics.

If the programmer is unhappy with the appearance of the produced GUI, she may exchange the employed interaction style with an alternative one. This is demonstrated in Figs. 4 and 5. Both figures show the beginning of the execution of the 'Edit Offer' use case. Fig. 4 demonstrates an interaction style, in which the use-case execution is initiated by selecting the use-case name in the 'Accessible use cases' menu and clicking it. In addition, this interaction style employs tables, which are sortable by clicking any column header. On Fig. 4, the user has sorted the table by the 'Price' column, in descending order. No changes or recompilations have been required to the code in Fig. 2. Fig. 5 demonstrates yet another interaction style. With this interaction style, the use-case execution is initiated by clicking one of the four yellow buttons on the top of the screen. In addition, note that this interaction style presents tabular data as a set of records. Finally, the interaction style employs push buttons instead of hyperlinks. Again, no changes or recompilations have been required to the code in Fig. 2.

We now consider a further use case of our example system, called 'Select Offers', which enables the customer to browse the offers of the various suppliers. The use case begins with the customer selecting a single city, a single item and approving her choice. Fig. 6 presents the code that implements the beginning of this use case. Passing `null` for the visible columns parameter of the `Select.one` method makes all the table columns visible.

Fig. 7 presents two different UI's, which were produced from the same single code of Fig. 6, by employing two different interaction styles. The interaction style on the left employs space-economic pull-down menus and may, therefore, be employed on the small screen of a PDA (Personal Digital Assistant). The interaction style on the right employs single-selection listboxes and may, therefore, be employed on a large PC screen. The example demonstrates that the same single application code may be employed on different devices, by using an interaction style that fits to the device. It is thus not required to make

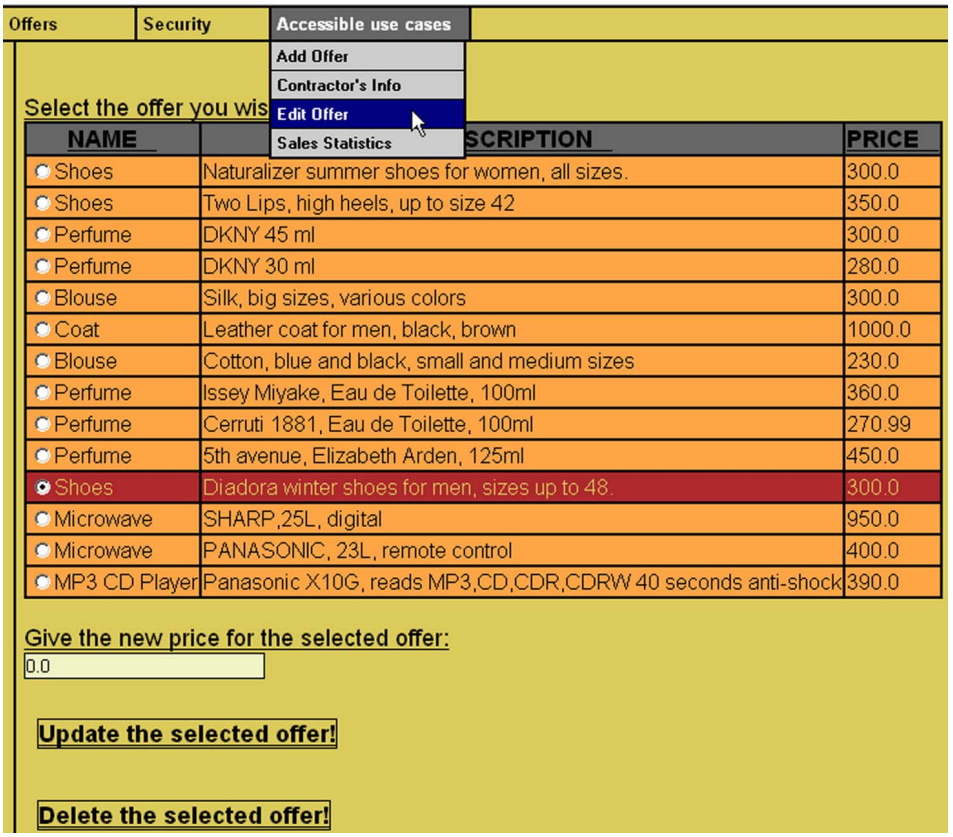

Fig. 4. An alternative interaction style.

any change in the code of a $\mathcal{WebSI}$-based system, when adapting it to a new device kind.

Table 1 lists some of $\mathcal{WebSI}$'s input and output semantics.

4. Database access with $\mathcal{WebSI}$

We have assumed that the essence of an information system is the moving and processing of data. Moving of data to and from the database is, therefore, anticipated to be a frequent operation, which deserves to be supported by $\mathcal{WebSI}$. Moreover, it is expected that a considerable part of the computations in an information system may be accomplished by the powerful SQL language. $\mathcal{WebSI}$ employs, therefore, SQL to manipulate database data. The programmer needs not know the actual location of the database; it is specified during the deployment of an information system. The various technical activities, such as connecting and disconnecting from the database, managing database transactions

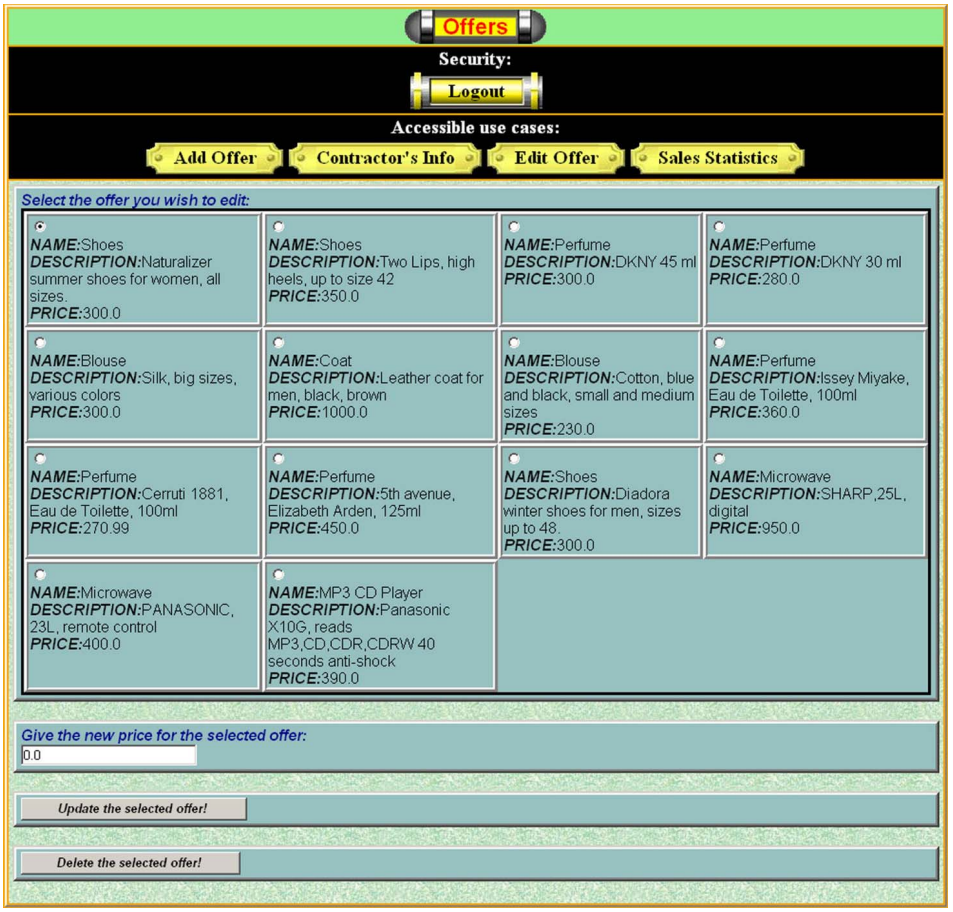

Fig. 5. Yet another interaction style.

```
public void start() throws Exception {
    Select.one(”selected_city”,
        DB.read(”SELECT DISTINCT city FROM supplier”), null, ”city”, null,
        ”Select the desired city:”);
    Select.one(”selected_item”,
        DB.read(”SELECT DISTINCT name FROM item”), null, ”name”, null,
        ”Select the item you are interested in:”);
    Action.action(”seeSuppliers”, ”See suppliers”);
}
```

Fig. 6. The implementation of the beginning of ‘Select Offers’ use case.

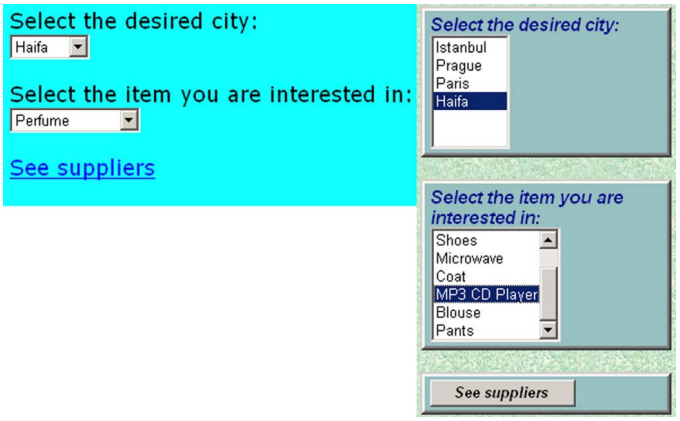

Fig. 7. Different implementations of the same semantics.

Table 1
Some of $\mathcal{WebSI}$'s input and output semantics

Input/output method	**Semantics**
`Output.scalar`	Present a scalar value (i.e., some text) to the user.
`Output.table`	Present a `Table` to the user. The `Table` possibly originates from an SQL SELECT query. All or some of the table's columns are visible.
`UserComposed.any`	Demand the user to supply a user-composed value.
`UserComposed.anyOptional`	The user may optionally input a user-composed value.
`UserComposed.yesNo`	Demand the user to answer a yes-no question.
`UserComposed.yesNoOptional`	The user may optionally answer a yes-no question.
`UserComposed.date`	Demand the user to supply a date value.
`UserComposed.dateOptional`	The user may optionally input a date value.
`UserComposed.time`	Demand the user to supply a time value.
`UserComposed.timeOptional`	The user may optionally input a time value.
`Action.action`	Enable the user to initiate an action in the system.
`Select.one`	Demand the user to select a single row out of a `Table`. The `Table` possibly originates from an SQL SELECT query. All or some of the `Table`'s columns are visible.
`Select.oneOptional`	Same as previous, but the user may also not select a row.
`Select.oneOrMore`	Demand the user to select one or more rows out of a `Table`. The `Table` possibly originates from an SQL SELECT query. All or some of the `Table`'s columns are visible.
`Select.oneOrMoreOptional`	Same as previous, but the user may also select zero rows.

and recovering from various database-related errors, are handled automatically by $\mathcal{WebSI}$.

4.1. Retrieving data from the database

WebSI provides a `DB.read` method, which performs an SQL SELECT query and returns the resulting table. The signature of this method is:

```
Table read(String SQL_SELECT_query);
```

where `Table` is an interface defined by *WebSI*. This interface provides a set of methods, which inspect the individual table values. However, in many cases obtaining the individual values is not required, since *WebSI* provides many input and output methods that work with `Table`s directly. For example, the `Select.one` method (Figs. 2 and 6) produces a set of possible user selections directly from a `Table`, obtained, e.g., by an SQL SELECT query. Another example is the `Output.table` method, which presents a `Table` to the user:

```
void table(Table table, String[] visibleColumns, String caption);
```

4.2. Modifying the database

Similar to reading data, there is a single method, `DB.write`, for modifying the data in the database:

```
int write(String SQL_query) throws DBConstraintViolationException;
```

The argument to this method is an SQL INSERT, UPDATE or DELETE query, and the return value is the number of rows affected by the query.

4.3. Transactions and database constraints

As already mentioned, *WebSI* employs database transactions to preserve the consistency of the database. A new transaction is started, when the code of an interaction method issues its first SQL query. When the interaction method ends, *WebSI* tries to commit the transaction, i.e., to commit all the queries the programmer has issued during the interaction method execution. In case the transaction cannot be committed, *WebSI* rolls it back and sends an explanatory error message to the user.

Database systems are capable of checking various integrity constraints, such as PRIMARY KEY and FOREIGN KEY constraints. When a constraint is violated, *WebSI* conveys the violation to the programmer, so that an appropriate action can be coded. The violation is delivered as a `DBConstraintViolationException`, or one of it subclasses defined by *WebSI*, according to the specific constraint type, e.g., a `ForeignKeyViolationException`.

5. Evaluation

WebSI has been employed by students for developing systems for air travel planning, assigning faculty members to courses, scheduling and managing theater performances, scheduling football games (teams, fields and referees), managing a Web discussion forum, assigning referees to school basketball games, as well as for managing a Web community of Magic card game players. The students, most of which were in the last year of their

Computer Science studies, got brief descriptions of their projects and were told that their grades depended on the usability of their systems. The students recorded their time usage and filled questionnaires with their comments. Seven hours of instruction were sufficient to enable students familiar with Java to start using $\mathcal{WebSI}$. The projects with the better usability could be specified and implemented with only 200–300 working hours. Some of the students, who have also implemented systems with a USDP-like software process using standard Java facilities, estimated that $\mathcal{WebSI}$ enabled a saving of around 30–40% of the total system development effort.

The reduction of the coding effort enabled investing more time in requirements elicitation and specification development. As expected, this shift of emphasis was beneficial for the usability of the systems; many of the produced projects were quite realistic and correspondingly complex. A team that produced a quality system invested 86 h in requirements and specification development and only 74 h in database design, implementation and verification. Another team that produced a relatively poor system invested 67 h in requirements and specification and 161 h in the implementation. In general, all teams, which developed reasonably good use-case specifications, obtained UI's, which were quite easy to learn and to use. Whether this useful result is true in general is a question that requires further investigations.

The programmer that employs $\mathcal{WebSI}$ is not required to construct and manipulate the UI, handle the various Web-related issues, access the database server and deal with security. The code to be written involves by and large only the input and output of data, database queries in SQL, and the required data manipulations. The code is quite close to what may be called pure *application logic*. When employing only SQL for the required computations, which was sufficient in many cases, each statement of an English use-case specification is, usually, implemented by 1–3 Java statements. This facilitates verifying by code inspection that the code implements the specifications correctly.

The *UI-free* $\mathcal{WebSI}$ code of an application should run without any change on any platform, for which an appropriate interaction style exists. Using the same code on all platforms may facilitate maintenance, preparation and distribution of new application versions. Note that $\mathcal{WebSI}$ interaction style components are not application-specific; therefore, they are not considered as a part of a code of a certain information system, and can be reused in multiple systems.

The appearance of a $\mathcal{WebSI}$-based system can be controlled by selecting one of the interaction styles, currently available in $\mathcal{WebSI}$. Figs. 3–5 and 7 illustrate what is possible with these interaction styles. The appearance can be improved by developing more sophisticated interaction styles. Implementing an interaction style from scratch may, however, be quite labor intensive.

The ease, by which an existing (legacy) system may be modified, is an extremely important software engineering quality. In a $\mathcal{WebSI}$-based information system, a use-case component usually interfaces only with the user and the database, but not with other use cases. The developer of such a use-case component, therefore, need not know anything about the internals of other use cases; she is required to understand the use-case specification, the database schema and be familiar with $\mathcal{WebSI}$ API. The effort required for extending an information system with a new use case is thus often independent of the

number of already existing use cases, i.e., the extension complexity [10] 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 [7], in that the schema of a database changes much more slowly than database applications. 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 usually met in our student projects. Indirect use-case dependencies may, however, occur between use cases, which participate in the same single work-flow, relying on 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). Such use cases have to be designed to work correctly together.

The modifiability of a $\mathcal{WebSI}$-based information system was checked experimentally in our student projects. After the air travel planning project was completed by one student team, another student team was asked to do some major modifications of the system, such as enabling the users to order tickets for flight routes, introducing different kinds of flights (daily, weekly and special) and new statistics reporting. $\mathcal{WebSI}$ facilitated the modification of the system in a number of ways. The students reported that having the code of each use case in a separate use-case class facilitated the location of the places where changes were needed. The students also reported that the high-level $\mathcal{WebSI}$-based code of the legacy system was relatively easy to understand. Furthermore, the direct correspondence between a use-case specification and its use-case class implementation facilitated verifying the modified code. Still, even with $\mathcal{WebSI}$'s facilities, modifying the legacy code was a major effort. The need to gain a sufficient understanding of a legacy system is by its nature labor intensive.

The current version of $\mathcal{WebSI}$ has no high-level support for the flow of control between the different use cases. A limited flow of control is provided through the UML `<<include>>` relationship, which is supported by $\mathcal{WebSI}$. The primary purpose of this facility in both UML and $\mathcal{WebSI}$ is, however, to enable the programmer to avoid duplicating common system behavior.

Additional research is required on the applicability and limitations of the $\mathcal{WebSI}$ approach in demanding, large real-life systems. It may be expected, for example, that using the same single interaction style in a system will produce a more uniform UI appearance than a hand-coded UI. However, there are cases, in which inconsistencies in the UI are required [3]. The possibility of developing interaction styles that implement the UI standards of particular software manufacturers is worth further investigations.

6. Related work

The "Play-In/Play-Out" [5] method was designed for capturing and directly executing the specifications of reactive systems. Following this methodology, first a 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 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.

This work differs from ours by being targeted at reactive systems, while $\mathcal{WebSI}$ is designed for information systems construction. In addition, the "play-in" stage involves a manual GUI construction, i.e., it is required to decide on the system's GUI at an early development phase. This contrasts to $\mathcal{WebSI}$'s automatic UI generation. Another difference is that "playing-in" enables an interactive specification development, while $\mathcal{WebSI}$ assumes a manually developed specification.

A novel technology for UI construction, called JavaServer Faces (JSF) [8], which extends the popular JavaServer Pages (JSP) [9] technology, enables a more convenient UI construction method than with basic HTML tags. JSF is similar to $\mathcal{WebSI}$ in facilitating the altering of UI controls employed in an application. In JSF, however, the UI controls specification is interwoven in the application and cannot be readily employed in other applications. This contrasts to $\mathcal{WebSI}$'s interaction styles, which can be reused in different applications. Using the $\mathcal{WebSI}$ framework, the developer of a new system need not invest time in UI construction at all, when an acceptable interaction style already exists.

Similarly to $\mathcal{WebSI}$, other systems, e.g., [16,17,1,2], employ various abstract UI specifications in order to produce UI's for multiple platforms. $\mathcal{WebSI}$ differs from these systems by enabling to the programmer to write application-specific code, which is completely free of UI details. This code should run on any platform without change.

7. Conclusions

This study suggests the feasibility of an SOC framework for the construction of Web information systems. The high-level abstractions of the $\mathcal{WebSI}$ framework enable manual translation of English language use-case specifications to Java code. This code resembles the specifications, making it easy to verify that it implements them precisely.

The developed framework hides the code for UI construction and manipulation, database access and Web programming. This enables the developers to focus on the flow and processing of data. Those are considered to be the essence of the information system. The framework saved in one case about 30–40% of the development effort.

The UI's, produced automatically from good quality use-case specifications, were easy to learn and to use. This suggests that the usability of the UI is to a considerable extent dictated by the use-case specifications.

The modification of a $\mathcal{WebSI}$-based legacy system by new programmers was facilitated by the relative readability of the $\mathcal{WebSI}$-based code and by its use-case structure. Modifying legacy code remained, however, a labor-intensive task.

Further studies are needed for a more complete understanding of the applicability of SOC's.

Acknowledgement

The assistance of Mr. Kobi Sasson in the implementation of some interaction styles is thankfully acknowledged.

References

[1] G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouillon, J. Vanderdonckt, A unifying reference framework for multi-target user interfaces, Journal of Interacting With Computers 15 (3) (2003) 289–308.
[2] S. Gilroy, M. Harrison, Using interaction style to match the ubiquitous user interface to the device-to-hand, in: EHCI-DSVIS'04, 11–13 July, Hamburg, Germany, 2004.
[3] J. Grudin, The case against user interface consistency, in: CACM, October 1989, pp. 1164–1173.
[4] D. Harel, Statecharts: a visual formalism for complex systems, Science of Computer Programming 8 (1987) 231–274.
[5] D. Harel, R. Marelly, Specifying and executing behavioral requirements: the Play In/Play-Out approach, Software and System Modeling (SoSyM) 2 (2003) 82–107.
[6] I. Jacobson, G. Booch, J. Rumbaugh, The Unified Software Development Process, Addison-Wesley, 1999.
[7] I. Jacobson, M. Christerson, P. Jonsson, G. Overgaard, Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley, 1992.
[8] JavaServer Faces Technology. http://java.sun.com/j2ee/javaserverfaces/.
[9] JavaServer Pages Technology. http://java.sun.com/products/jsp/.
[10] E. Kantorowitz, Algorithm simplification through object orientation, Software Practice and Experience 27 (2) (1997) 173–183.
[11] E. Kantorowitz, O. Sudarsky, The adaptable user interface, Communication of the ACM 32 (1989) 1352.
[12] E. Kantorowitz, S. Tadmor, A specification-oriented framework for information system user interface, in: Workshop of Object-Oriented Information Systems 2002, OOIS'02, Springer LNCS, vol. 2426, 2002.
[13] E. Kantorowitz, A. Lyakas, A. Myasqobsky, Use case-oriented software architecture, in: ECOOP'2003, Workshop 11, Correctness of Model-based Software Composition, 2003.
[14] E. Kantorowitz, A. Lyakas, A. Myasqobsky, A use case-oriented user interface framework, in: SwSTE'03, IEEE International Conference on Software—Science, Technology & Engineering, November 2003, Herzelia, Israel, 2003.
[15] D.H. Lorenz, J. Vlissides, Designing components versus objects: a transformational approach, in: ICSE 2001, 12–19 May, Toronto, Ontario, Canada, 2001, pp. 253–262.
[16] F. Paterno', A. Leonardi, A semantics-based approach to the design and implementation of interaction objects, Computer Graphics Forum 13 (3) (1994) 195–204.
[17] F. Paterno', C. Santoro, A unified method for designing interactive systems adaptable to mobile and stationary platforms, Interacting with Computers 15 (3) (2003) 347–364.
[18] J. Rumbaugh, I. Jacobson, G. Booch, The Unified Modeling Language Reference Manual, Addison-Wesley, 1999.