A Specification-Oriented Framework for Information System User Interfaces

Eliezer Kantorowitz and Sally Tadmor

Computer Science Dept.
Technion – Israel Institute of Technology
32000 Haifa, Israel
{kantor, sally}@cs.technion.ac.il
http://www.cs.technion.ac.il/~kantor/

Abstract. A costly part of software development regards verification, i.e., checking that the code implements the specification correctly. We introduce the concept of *specification-oriented* frameworks, with the purpose of facilitating verification. A specification-oriented framework enables direct translation of the specifications into code, whose equivalence with the specification is easy to establish. The feasibility of this concept was investigated with the experimental *Simple Interfacing* (SI) framework for construction of user interface software of interactive information systems. Such an information system is constructed by translating the natural-language use-case specification into SI based code. SI provides high-level methods for data entry and data display. The use-case coding assumes that a database schema and required complex data manipulations are developed separately. The code produced for the use-cases of five small projects corresponded quite closely to the natural-language specifications and facilitated considerably the verification. Further research is required to assess the usefulness of the approach.

1 Introduction

The software engineering challenge regards the manufacturing and management of complex systems. One mean to overcome these complexities is the use of high-level tools such as very high-level languages and frameworks of high-level software components. This investigation is concerned with the design of frameworks for construction of graphical user interfaces (GUI) for interactive information systems (IS). It considers frameworks for use-case oriented software development processes, such as the Unified Software Development Process (USDP) [1]. The USDP was designed in connection with design of the Unified Modeling language (UML) [2]. We are especially interested in use-case oriented processes as they facilitate the manufacturing of systems with high usability levels, i.e., systems that enable the users to accomplish all required tasks with a minimum of effort and in a pleasant way. This is achieved by employing usability considerations in the design of the use-cases. The USDP process begins with requirement elicitation and continues with the specification of the system by its use-cases. First a natural-language use-case specification is developed and its

usability is validated. The use of natural language enables validation by domain experts that are not familiar with formal specification methods. The validated use-cases are then translated into formal UML use-case diagrams. The process continues with the analysis of the formal use-case specification, system design, implementation and testing. The testing phase includes *verification*, i.e., checking that the code implements the use-case specification correctly. This is usually done by testing each one of the different use-cases with sets of test data that cover the different kinds of possible scenarios (black box testing). The verification may also be accomplished with formal methods that assume that the natural-language use-case specification has been correctly translated into a formal use-case specification. Using formal methods, e.g. [3], to show that the code implements the formal specification is not a trivial task. The verification process involves a considerable effort. In some situations it is possible to produce the code automatically from the specifications and thus completely avoid the costly verification. One example is a statecharts specification of reactive systems [4] that may be translated automatically to code [5]. We are, however, not aware of a general method for producing the code of information systems from their specification. This study suggests therefore a less ambitious approach to reduce the verification effort.

In the approach suggested in this paper, the natural-language use-case specification is translated into high-level code, which implements the use-cases. The verification effort in this approach is reduced to showing the equivalence between the natural-language use-case specification and the code. We denote a framework that enables a direct coding of the specifications as a *specification-oriented framework*. The process proposed in this paper differs from the USDP process where the use-cases are not coded, but form the basis for the analysis and design of the system. The feasibility of the approach proposed in his paper depends on whether it is possible to design an appropriate high-level language or component framework for coding of use-case activities. In this research we investigate the feasibility of a use-case specification-oriented framework for construction of the GUI software of interactive information systems. To the best of our knowledge, none of the frameworks existing today has been especially designed to support use-case specification-oriented system development.

2 Model of the Experimental System

This section describes our use-case oriented model of the GUI software of interactive information systems. Later we describe our experimental implementation of this model, using our Simple Interfacing (SI) framework. Our model suggests IS software to be composed of four parts. The first part is the implementation of the use cases, based on some basic actions. The second part is a framework implementing these basic actions. The third part is a database and the last part is data manipulation software.

In SI we employ the following basic use case actions:

1. displaying data to the user,
2. getting data from the user
3. getting data from the database
4. inserting and updating data in the database
5. general purpose data manipulation

By database we mean a persistent database that may be employed for sharing of data between different applications. Our five basic actions hide the geometrical properties of the Graphical User Interface (GUI), its related event handling and the database access methods. The basic actions of our model only specify the flow of data between the user, the system and the database. In other words, they specify the input and output of data to the system. Our basic actions do not specify how the in/output is done. The motivation for selecting this abstraction is that we consider the input and output of data to the system to be the semantics of the user-interface system. The purpose of the graphical layout of the user-interface is to facilitate the work of the user.

General-purpose data manipulation is done by tools such as the SQL language. However, a use case specification may beyond the above five basic operations employ special kinds of data manipulation, e.g. scheduling tennis games by special tournament rules, finding the least expensive air line connection between two cities. Such special data manipulation operations cannot be part of our general purpose SI use case framework. It is therefore assumed that frameworks for such special operations are developed separately. A use case is coded by combining the classes of the general-purpose framework with the classes of operations that are specific to the application.

The design of the experimental SI system is described in Appendix 1. An example of the use of SI is shown in Appendix 2. The next section evaluates the results of five small SI based projects.

3 Evaluation

3.1 Code is Data-Centered

Data input and output are considered to be the basic operators of the user interface. SI based code is basically data-centered. The I/O operators of SI are implemented by methods of the `PaneController` class listed in Tab. 1 and Tab. 2.

Table 1. Input from the user and output to her/him.

Input methods	Output methods
`RequestInput`	`Display`
`GetInput`	`AddLabel`
`AddButton`	`AddPicture`

Table 2. Input to the database and output from it.

Input methods	Output methods
`Update`	`Display`

3.2 Ease of Verification and Traceability

In the tested examples, the use of the SI framework produced code that resembles the natural-language use-case specification. This facilitated the verification considerably. We have examined the verification of the use-case *Select Offer* (Appendix 2). We compared the above SI based implementation with another implementation employing only Java and its Swing and JDBC packages. The SI based code hides the GUI and database access details and is therefore shorter and seemingly easier to understand. As shown in Table 2, the SI based code is only 61 lines long, while the code using the Swing and JDBC is 147 lines long. Another expression of the shortness of the SI based code is that each statement in the natural-language use-case specification translates on the average into 2.5 lines of SI based code [6], while it takes 7 lines of Java code using the Swing and JDBC packages.

The verification of the SI based code was further facilitated by the observed ease of tracing of the lines of code that corresponds to each statement in the natural-language use-case specification [6]. This ease of tracing is because the code of a particular use-case specification is found in a class that has the same name as the use case and extends the SI class `UseCase`. Tracing the code that implements a use-case specification in the longer non-SI based Java code was noticeable more difficult. The ease by which the code that implements a use case can be traced in SI based code is also expected to facilitate future modifications of the use-cases. However, verifying specifications of the user-interface appearance requires familiarity with the interaction-styles of SI, since the SI based code hide these details.

Table 3. Length of code to implement a use-case, with SI and with Swing and JDBC.

	Total number of lines of the use-case class code	Average number of lines of code implementing a single statement in the natural-language use-case specification
Implementation with Java, Swing and JDBC	147	7
Implementation with Java and SI	61	2.5

3.3 Learning Time and Interaction-Styles Limiting

Coding with SI requires understanding the use of the underlying interaction styles, which requires some learning. These standard interaction styles were satisfactory in

four of the five tested projects. In one project the programmers wanted a different GUI appearance, e.g., using different widgets, and placing them differently. This required developing a new interaction style. Writing a new style requires understanding of the internal structure of SI, i.e., the methods of the `StyledPane` and `StyledFrame` classes, inheriting from one or two of these classes, and overriding some of their methods. Reaching this higher level of understanding required further learning.

4 Conclusions and Further Research

The use of SI produced code that is relatively easy to verify. It is also expected to facilitate future use case modifications. The use of SI also reduced the coding effort. In one example the code written with SI had only 40% of the length of equivalent Java code exploiting the Swing and JDBC packages. Using the ready-made interaction-styles standardizes the appearance of the GUI.

It is interesting to compare the specifications-oriented SI package with the GUI-oriented Swing package of Java. Swing has classes for detailed construction of GUI widgets. Input and output with these Swing widgets requires detailed coding. SI is on the other hand centered on abstract input and output operations. The SI widgets are also at a high level of abstraction hiding the geometrical details in interaction styles.

The interaction-styles provided by the current version of SI were not satisfactory in one of the five test projects. The design of an interaction-style involves a difficult tradeoff between an interaction-style being general-purpose and being able to meet special requests. More research on design of interaction-styles is needed.

SI has been tested with small systems. To fully assess the approach it must be tried on large systems. Considering the quite promising results, we believe the approach deserves further investigations.

References

1. Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Software Development Process, Addison-Wesley, (1999).
2. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual, Addison-Wesley, (1999).
3. Chechik, M., Gannon, J.: Automatic Analysis of Consistency between Requirements and Designs, IEEE Transactions on Software Engineering, 27,7,(July 2001).
4. Harel, D.: Statecharts: A Visual Formalism for Complex Systems, Science of Computer Programming, 8,(1987) 231-174, North Holland.
5. Rhapsody product of I-Logix company, see http://www.ilogix.com/products/rhapsody/index.cfm
6. Tadmor, S.: A Framework for Interactive Information Systems, An M. S. thesis, Technion Institute of Science, Israel, (2002).
7. Goldberg, A. J., Robson, D.: SmallTalk-80: The Language and Its Implementation, Addison-Wesley, (1983).
8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns, Addison-Wesley, (1995).

9. Kantorowitz, E., Sudarsky, O.: "The Adaptable User-interface", Commun.ACM, 32, 11,(Nov 1989),1352-1352.

Appendix 1. Structure of the Experimental SI framework

A1.1 Design Decisions for SI

This section discusses the design decisions made for the experimental SI system:

1. Implementing SI with Java and its Swing and JDBC (Java Data Base Connectivity) packages. These tools are considered to be * the state of the art and yet sufficient mature.
2. Using a relational database and SQL, powerful and widely available technologies.
3. Using the Model-View-Controller (MVC) design pattern [7] [8] for the design of SI. Using the MVC pattern is expected to facilitate modifications that may be needed at later stages
4. Using a use-case as an implementation unit. Each use-case is implemented separately, by one Java class, which extends the SI `UseCase` class. In terms of MVC the use-case class is the `Controller`. The schema of the relational database may implement the Model of the MVC pattern.
5. Separating the specification of the *interaction-style,* i.e., the way the GUI appears, from the systems functionality. This will enable the interaction-styles to be reused in different applications. From the MVC point of view, the interaction-style is part of the View.

The idea of interaction-styles is inspired by the *dialogue modes* [9]. We define an *interaction-style* of a GUI as the set of all its properties, e.g., colors of widgets, sizes, layout of widgets on the pane, and also the types of widget used. Once constructed, an interaction-style may be employed in a number of different information systems. SI comes with a number of ready-made interaction-styles. The implementer of an information system may, however, define additional reusable interaction-styles to meet special needs.

A1.2 Using SI

The manufacturing of an information system with the SI framework begins with the usual requirement elicitation and detailed natural-language specification of the use-cases. The process continues with an analysis resulting a schema of the system's database. The analysis and design of database schema is, however, not needed in the many cases, where it is required to employ an existing database of a given enterprise. The system may now be implemented by writing a use-case class for each one of the specified use-cases. By convention we name this class with the name of the use-case that it implements. This code is essentially a translation of the statements of the natural-language use-case specification into appropriate method calls. These methods belong either to classes of the SI framework or to classes provided by the programmer for

application specific manipulations. Each statement in the natural-language use-case specification is typically translated into one or two method calls. The verification of the resulting code is done by checking the equivalence between this code and the natural-language use-case specification. The database is created separately using a relational database system.

The following are the five service classes of SI that are visible to the programmer. Their important methods are explained in the following sections.

1. `UseCase` – all use-case classes inherit from this class. This class has a documentation role: since all use-case classes inherit from it, it is clear what the use-case classes of the system are. Apart from this role, this class gives some hidden flow-of-control services.
2. `FrameController` – the code of the use-case classes use this class to build *frames* of the GUI. A *frame* is the basic GUI unit defined by the Java Swing package. Frames have Swing *panes* added to them.
3. `PaneController` – the code of the use-case classes use this class to build panes, and attach them to frames of the GUI.
4. `DBProxy` – gives database services to all framework and system classes. System classes use this class when the database services of the `PaneController` class are not sufficient.
5. `Constants` – keeps the constants of the system.

The above few classes and the methods that are listed in the next section are what a use-case programmer has to learn. According to our experiments [6], it takes 4 hours. When the interaction-styles that come with **`SI`** are not satisfactory, new interaction-styles must be implemented. Learning to do this took 4 more hours

GUI Construction. In SI panes and frames are implemented by instances of the `PaneController` and `FrameController` classes. A singular human-computer interaction is in SI handled with one particular pane. The `PaneControllers` give all the services of the basic actions: displaying data to the user, getting data from the user, and updating the database.

The GUI is designed according to the MVC model [8]. The `PaneController` and the `FrameController` are the Controllers of the GUI. Two classes, called the `StyledPane` and `StyledFrame`, implement the View and Model.

The services of the `FrameController` class are:

1. `addPane(PaneController pane)` – adds a pane to the frame.
2. `finallySet()` – sets the frame to be visible and packs it.
3. `close()` – closes the frame.

The services of the `PaneController` class are:

1. `display(String query)` – executes the query by accessing the database, and displays the resulting relation on the pane, in a few text-boxes or in a table. The appearance of the pane is determined by the selected interaction-style.

2. `requestInput(String[] SQLTypes, String[] captions)` – adds widgets to the pane, to request these input data types. The widgets have these captions.
3. `getInput()` – after requesting input from the user, this methods gets the data inserted by the user in the widgets of the pane. The method returns a vector of objects, containing the data inserted by the user.
4. `update(String SQLUpdateQuery)` – executes the query on the database.
5. `addButton(String method, String caption)` - adds a button to the pane, with this caption. When the button is pressed, this method is called.
6. `addButton(String caption)` - adds a button to the pane. Pressing this button will call no method. This method is used for prototyping.
7. `addLabel(String caption)` – adds a label with this caption to the pane.
8. `addPicture(String fileName)` – adds the picture in this file to the pane.

Connecting to the Database. A system can connect to the database using the `DBProxy` class. The services of this class are:

1. `ResultSet executeQuery(String query)` – executes the query, returning a result set.
2. `executeUpdate(String query)` - executes the update query.
3. `close()` - closes the connection to the database.

Flow of control services. These services enable a use-case to return control to the calling use-case or other use-cases. These services are given by the UseCase class, and explained in [6].

Appendix 2. Example – SelectOffer Use Case

In this section we bring a short example, to demonstrate the quality of code written using SI. We take the *Select Offer* use-case of the *Offers* system [6]. The use-case natural-language specification is:

"The user sees a frame with two lists, a list of cities and a list of items. The user selects a city and an item, and can press one of two buttons: the Cancel button, or the See Suppliers button. If the user presses the Cancel button, the frame is closed. If the user presses the See Suppliers button, a list of suppliers appears..".

By analyzing all the use cases of the application, the schema of the relational data employed by the code below was designed. The analysis and design of this schema was done traditionally. The resulting SI based code of the use case appears on the next page. It produces the frame shown in Fig. 1.

```
import java.util.*;
import si.*;

public class SelectOffer extends UseCase{

  private String customerID = null;

  private FrameController frame = null;

  private PaneControllerListStyleWidth2 pane = null;

  public SelectOffer(String customerID){
    super();
    frame = new FrameController("What offers are you
interested in?");
    pane = new PaneControllerListStyleWidth2(this);
    this.customerID = customerID;
    pane.addLabel("Select city:");
    pane.display("select distinct city from Supplier");
    pane.addLabel("Select item:");
    pane.display("select distinct name from Item");
    pane.addButton("seeSuppliers","See Suppliers");
    pane.addButton("cancel","Cancel");
    frame.addPane(pane);
    frame.finallySet();
  }

  public void seeSuppliers(){
      //…
  }

  public void cancel(){
      //…
  }

}
```

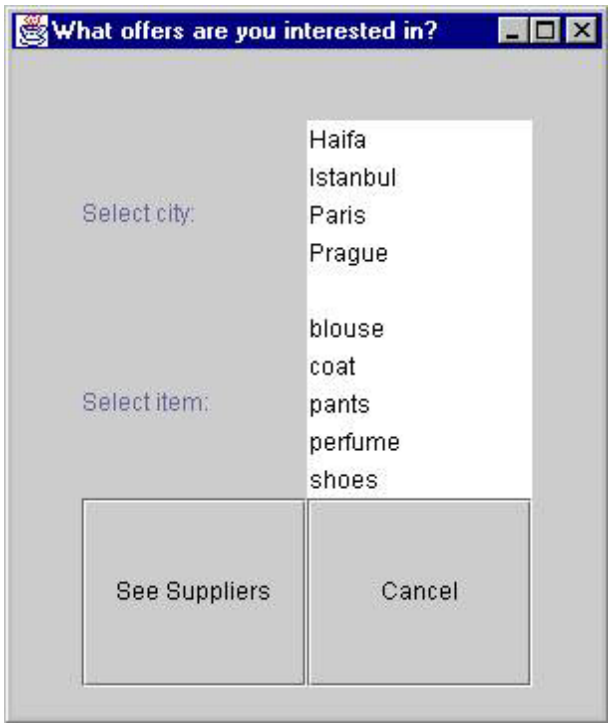

Fig. 1. The frame displayed by the *Select Offers* use-case.