

# Toward Specification-Oriented Frameworks

by

Eliezer Kantorowitz and Sally Tadmor<sup>1</sup>

Computer Science Dept.

Technion – Israel Institute of Technology

32000 Haifa, Israel

The software development process involves a costly verification, i.e., checking that the code implements the specification correctly. This study introduces the concept of *specification-oriented* frameworks, with the purpose of facilitating verification.

A specification-oriented framework is a framework that enables direct translation of the specifications into high-level code, whose equivalence with the specification is easy to establish. The code should resemble the specification to such an extent that their equivalence maybe readily seen. The feasibility of this approach is studied in this paper with the experimental **Simple Interfacing (SI)** framework for construction of graphical user-interface (GUI) software of interactive information systems. In this paper we discuss the design of **SI** and the results of the experimental evaluation.

This study assumes that the information system is developed using a use-case oriented process, such as the Unified Software Development Process (USDP) [Jacobson99]. 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. Formal methods of verification of this kind are for instance described in [Chechik01]. The verification process typically involves a considerable effort. In this study we consider an approach that is designed 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

---

<sup>1</sup> Emails: kantor@cs.technion.ac.il and sally@cs.technion.ac.il

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 framework for coding of use-case activities directly. To the best of our knowledge, none of the frameworks existing today has been especially designed to support use-case specification-oriented system development.

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 in our Simple Interfacing framework (**SI**). Our model suggests software to be composed of three parts. The first part is a high-level implementation of the specification, employing the basic actions:

1. Displaying data to the user
2. Getting data from the user
3. Getting data from the database, and
4. Inserting and updating data in the database.

The second part is a framework implementing these basic actions. The last part is a database that may be shared by a number of different applications. The four basic actions hide the geometrical properties of the 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, but not 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 basic semantics of the user-interface system. The purpose of the graphical layout of the user-interface is to facilitate the work of the user.

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 state of the art and sufficient mature.
  2. Using a relational database and SQL, powerful and widely available technologies.
  3. Using the Model-View-Controller (MVC) design pattern [Goldberg83] [Gamma95] 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<sup>2</sup>, which extends the **SI** UseCase class. In terms of MVC the use-case class is the Controller. The relational database implements the Model of the MVC pattern.
  5. Separating the specification of the *interaction-style*, i.e. the way the GUI appears. 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*
-

[Kantorowitz89]. 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 interaction-styles to meet special needs. Such additional interaction-style may also be reused in future projects.

The following piece of SI based code implements a part of a particular use case. The resemblance of this code to a natural language specification is expected to facilitate the verification of the code.

```

1.  pane.addLabel("Select city:");
2.  pane.display("select distinct city from Supplier");
3.  pane.addLabel("Select item:");
4.  pane.display("select distinct name from Item");
5.  pane.addButton("seeSuppliers","See Suppliers");
6.  pane.addButton("cancel","Cancel");
7.  frame.addPane(pane);
8.  frame.finallySet();
9.  }

10. public void seeSuppliers(){
11.     //...
12. }

```

Statement 2 of the code employs an SQL statement to display a menu of all cities present in the Supplier table of the database, i.e. all cities having a supplier. The user may select a city in this menu. Similarly, statement 4 displays a menu of all item names present in the Item table of the database. After selecting both a city and an item, the user can push the “See Suppliers” button, which activates the seeSuppliers method to display the suppliers of the selected item in the selected city. Alternatively the user may select the “Cancel” button to cancel the query. The resemblance the **SI** based code to a natural language use case specification is achieved by hiding the detailed code for GUI presentation, handling of user events and database access.

In order to understand the difference between the specification oriented code and traditional code we compared two implementations of the same use-case, i.e. an **SI** based implementation and an implementation using JAVA and its Swing and JDBC packages[Tadmor02]. The results are shown in the table below

	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 <b>SI</b>	61	2.5

The **SI** based code is thus only 61 line of code compared to 147 lines of traditional code. What we found especially useful for verifying that the code implements the specifications was [Tadmor02]:

1. That each statement in the natural language use case specification translated to only 2.5 lines of specification oriented code
2. It was quite easy to trace the lines of code that correspond to each statement in the natural-language use-case specification. This traceability was achieved by implementing each use-case in a separate class and by implementing each statement in the specification by a small number of method calls. Tracing the use-case specifications in the longer Java code that employed the Swing and JDBC packages was noticeable more difficult.

Verifying specifications regarding the user-interface appearance requires, however, some familiarity with the interaction-styles of **SI**, since the code of the GUI is hidden. A further problem with **SI** was that the GUI produced with a given interaction styles was not always satisfactory. In one our five **SI** test projects required considerable modifications of the interaction style. More research is thus required in interaction style design. **SI** has only been tested with five different quite small projects. Testing the system with complex real life system in different application domains is required to fully assess its applicability

## References

1. [Agresti86] – Agresti W. W., ed., *New Paradigms for Software Development*, IEEE Computer Society Press, 1986.
2. [Chechik01] - Marsh Chechik and John Gannon, *Automatic Analysis of Consistency between Requirements and Designs*, IEEE Transactions on Software Engineering, July 2001 (Vol. 27, No. 7). They describe formal method of verification. This requires that both the requirements and the design are defined very formally.
3. [Gamma95] – Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns*, Addison-Wesley, 1995.
4. [Goldberg83] - Adele J. Goldberg and David Robson, *SmallTalk-80: The Language and Its Implementation*, Addison-Wesley, 1983.
5. [Jacobson92] – Ivar Jacobson, Magnus Christerson, Patrik Jonsson and Gunnar Övergaard, *Object-Oriented Software Engineering*, Addison-Wesley, 1992.
6. [Jacobson99] – Ivar Jacobson, Grady Booch and James Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, 1999.
7. [Kantorowitz89] – Eliezer Kantorowitz and Oded Sudarsky, “*The Adaptable User-interface*”, ACM 0001-0782/89/1100-1352, 1989
8. [Tadmor02] – Sally Tadmor, *A Framework for Interactive Information Systems*, An M.Sc thesis, Technion Institute of Science, Israel, 2002.