A Method for Extracting Requirements that a User Interface Prototype Contains

Alon Ravid and Daniel M. Berry
Faculty of Computer Science, Technion—Israel Institute of Technology, Haifa 32000, Israel, dberry@csg.uwaterloo.ca

1 Introduction

User interface (UI) prototyping (UIPing) (Connell and Shafer 1989, 1995) is a commonly employed requirements elicitation and validation technique, used to determine the functionality, UI, data structure, and other characteristics of a system. User requirements are explored through experimental development, demonstration, refinement, and iteration. The UI prototype (UIP) is built during the requirements analysis and specification phases of a software project. The goal of UIPing is to discover user requirements through early implementation of the UI and the functionality behind it. A UIP permits users to relate to something tangible and to obtain a requirements definition that is as complete as possible, so that the requirements can be validated by the user on the basis of realistic examples. The products of this process are various documents such as a software requirements specification (SRS), an occupation analysis document (OAD), and the prototype itself. (An OAD is essentially a draft version of a tutorial and user's manual for the system to be built.) The process of creating a UIP involves some difficulties. Once the prototype is developed and agreed upon, how is the information that it contains captured and represented in the other analysis documents?

2 The Project that Exposed the Problem

In recent years, Ravid, the first author, was the software system engineer for a highly complicated simulator for generating infrared (IR) scenes, called the Target Scene Generator (TSG). Because of the many difficulties faced during the early stages of the system specification, The team decided to rapidly develop a throwaway UIP (Andriole 1994). This UIP was to improve the communication with the customer and to help complete requirements specification within a reasonable amount of time. Software engineers, human engineering people, and users group representatives were involved in the development of the prototype. This approach turned out to be successful. Some major misunderstandings with the customer and many contradicting requirements were discovered. As is recommended by many including Fred Brooks (1995), the prototype was thrown away, and the development of the production version was started from scratch.

3 Problem Description

The process of creating the TSG prototype involved some difficulties, which are common to other prototyping-oriented projects as well. Given an informal problem description, it is not obvious how to systematically and efficiently construct a prototype. It is hard to distinguish between requirements and design details. After completing the prototype, a method is needed to integrate it to other models of the system. A portion of the information implicit in the prototype is left implicit and is not known until the prototype is used to answer questions. A method is needed to capture the prototype's semantics, and state them formally, in order that they will provide a suitable and traceable base for further development and testing. This problem was discovered unexpectedly while the team was preparing the project to be reviewed for ISO 9000.3 compliance by the Israeli Standards Institute. During a preceding internal review, Ravid was asked to present the project and the software development documents produced up to the day the review was conducted. Ravid presented the system specification documents, the SRS, the OAD, and the prototype. The prototype aroused additional questions about the system. Obviously, the prototype was used to answer these questions. After Ravid finished answering these questions, one of the reviewers asked "Where is all this information written?" Part of the information appeared in the SRS, part was expressed indirectly by other statements in the SRS, part was written in the OAD, and part was not written anywhere even though it was known, understood, and agreed upon by all the people involved in the project by virtue of their having worked together to produce the prototype. This undocumented information included indispensable knowledge about the system, knowledge which seemed to be essential for new programmers joining the group and for maintenance personnel who will have to support the system in the near and far future.

4 Key Question

We ask two key questions about UIPs.

1. What does a UIP say and what does it not say?

2. What is the right way to formalize and present requirements which are specified and embodied in a UIP?

We have concluded that there is no way to extract this information from a previously written prototype in the absence of knowledge of how it was developed. This is because there are aspects of the prototype's behavior that are artifacts of the prototyping process, often rapid, that are not intended as specified behavior. Just examining a prototype does not allow distinguishing intended from non-intended behavior. The problem is the same as that of reverse engineering from code. Therefore, it is necessary to have carried out prototyping explicitly as part of a fully traced requirements elicitation process in which it is decided and documented ahead of time what behavioral aspects, usually in the user-interface, are being modeled in the prototype. The tracing links allow easy access to the documentation of this decision so that in the future, when one is following the trace links to track down an answer to a requirements issue, one sees the explicit decision and knows whether to consult the prototype or another document in the requirements specification suite.

5 The Proposed Solution

We decided to use existing methods to model requirements, because we believe there are more than enough. The modeling technique should be chosen based on the characteristics of the system being modeled. Most importantly, the problem addressed by this research does not result from the lack of modeling methods, but rather from the fact that existing methods fail to identify the kinds of information that a UIP contains and thus do not provide a way to capture and present that information. Instead, the solution presented here consists of steps to be taken in addition to existing UIPing and requirements modeling methods, to be applied *before* and *during* UIPing and requirements modeling, to ensure that later, developers are able to answer the questions of what the UIP specifies and what it does not.

There is no advice here on what to prototype and what not to prototype. What is prototyped and what is not are the specifier's decisions. This decision is made based on his or her experience in specifying other systems in the same application domains; it will be based on what is already well understood and what is not. There is advice here on what to do once the decision is made so that the developers and maintainers know what of the prototype is intended and where to look, in the prototype or in other documents, for answers about their questions. The advice concerns documenting this choice and creating tracing links among the documents that help search for answers to these questions.

The basic idea of the proposed approach is to perform exploratory requirements prototyping in a systematic fashion that is based on tailoring a prototype construction process to the system under development, on identifying, before UIP-ing begins, the kinds of requirements information that the UIP will contain and will not contain, and on choosing modeling techniques that properly represent this information. Tailoring a prototype construction process is done in six recurrent steps, recurrent in that as one is following the steps in sequence, one can go back to any previous step to redo it based on new information.

1. Define the system's operational environment and its interfaces to other systems.
2. Identify to which application domains the system belongs.
3. Characterize the principal properties and the main features of an application belonging to these domains.
4. Identify which of these properties are applicable to the system under development.
5. Decide which of the identified properties requirements will be prototyped.
6. Prototype the system's interfaces and chosen properties.

The concept of application domains is an important one. All programs in a domain share common data, attributes, and operations. Examples of domains are simulation systems, information systems, data acquisition systems, and UI-intensive systems. An application can find itself in several domains, depending on the functions it must provide. The point about a domain is that from past experience, much is known about the data, attributes, and operations that are needed by any application in the domain. Often, there are even libraries of data definitions, procedures, and functions that can be used by simple inclusion to simplify programming the application. For example, there are many GUI building libraries available to be used to build any UI-intensive application.

The kinds of requirements information that a UIP contains can be classified into the following types,

1. the functionality and behavior of the application, that is, its reactive nature, including constraints placed on this behavior and operational logic,
2. the application's data model, data dictionary, and data processing capabilities,
3. the application's taxonomy along with a dictionary for it,
4. a partial specification of the interfaces to other systems, and
5. general knowledge about the system and intent.

Prototyping of the system interfaces and chosen properties is conducted an iterative manner as recommended in many publications about prototype-oriented software development. Developers and users can go over this list of properties jointly in a systematic fashion. They can discuss them by means of interviews, modeling, implementation, demonstration, refinement, and validation. The requirements information can be grouped and organized according to the kinds of information that the developers want to discover, which is also the kind of information the UIP will contain. The developers and customers can address all related issues and, when they come to an agreement, state the requirements formally. Appropriate models for stating the agreed upon requirements formally should be chosen based on the application domain, on the nature of the application to be developed, and on many other factors. The number and types of these models and the level of detail can be chosen after deciding if the prototype will be thrown away. All prototyped aspects have to be modeled. The chosen method should have the capability of representing the aspects listed above, in order to produce a good requirements model and avoid losing valuable information about software requirements.

6 Conclusions

The presented approach attempts to deal with the every-day prototyping reality, the approach to prototype construction, the distinction between the information the prototype contains and the modeling method, and the use of the prototype as a requirements elicitation aid and as a requirements model itself.

To validate the approach, we have carried out a case study applying the solution approach to the development of the TSG system. Space limitations preclude giving any details of the case study. The reader is referred to the first author’s thesis for the missing details (Ravid 1999). In that document, the reader will find an adaptation of the general approach to the specific UIPing technique used to learn the requirements for the TSG, a detailed description of the case study, an evaluation of the effectiveness of the solution, and lessons learned.

7 References

Andriole, S.J. (1994). Fast Cheap Requirements: Prototype or Else!. *IEEE Software*, 14(2), 85–87.

Brooks, F.P., Jr. (1995). *The Mythical Man Month*, Second Edition. Reading: Addison Wesley.

Connell, J.L. & Shafer, L.B. (1989). *Structured Rapid Prototyping*. Englewood-Cliffs: Prentice Hall, Yourdon Press.

Connell, J.L. & Shafer, L.B. (1995). *Object-Oriented Rapid Prototyping*. Englewood-Cliffs: Prentice Hall, Yourdon Press.

Ravid, A. (1999). A Method for Extracting and Stating Software Requirements that a User Interface Prototype Contains. M.Sc. Thesis, Faculty of Computer Science, Technion, Haifa, Israel, available at ftp://www.cs.technion.ac.il/pub/misc/dberry/Thesis.doc .