A Method for Extracting and Stating Software Requirements that a User Interface Prototype Contains

Alon Ravid (alonrv@earthlink.net)
Daniel M. Berry (dberry@csg.uwaterloo.ca)

Computer Science, Technion, Haifa, Israel

© 1999 Alon Ravid and Daniel M. Berry

User Interface Prototyping

User Interface (UI)

UI Prototype (UIP)

UI Prototyping (UIPing)

> **a technique for exploring requirements, mainly in the UI of an application, by having the client and user validate incomplete, quickly developed implementations**

© 1999 Alon Ravid and Daniel M. Berry

UIPing

Leads eventually to

- **UIP**
- **Requirements documents (RDs), including**
  - **Software Requirements Specification (SRS)**
  - **User's Manual (UM)**
  - **Requirements Rationale (RR)**
  - **etc.**

© 1999 Alon Ravid and Daniel M. Berry

After Delivery of Documents

The UIP and RDs are given to the implementors.

Requirements engineers (REs) might move on to other projects.

New members may join the project.

Knowledgeable REs may be unavailable to implementors for questioning.

© 1999 Alon Ravid and Daniel M. Berry

The Problem

In implementation phase, implementors work from UIP, SRS, UM, RR, etc. to implement application.

Sometimes, they have a question.

Assuming that the REs are not available for questioning, the implementors must ask the UIP, SRS, UM, RR, etc.

The Problem, Cont'd

What happens if the UIP and the RDs disagree?

Which, if any, do you believe?

Note that this problem occurs all too frequently in industrial UIPing.

Typical Situation

Typically, the SRS, UM, and RR agree with each other as a group more than with the UIP.

The normal requirements engineering (REing) process encourages consistency checking between the written documents, the SRS, UM, and RR.

Typical Situation, Cont'd

Lack of consistency between the SRS, UM, and RR is an old problem, studied elsewhere, with its own methods for overcoming it.

Finally, disagreement between a program and its specification is a very pervasive problem.

Crux of the Problem

Therefore, we use the SRS as the representative of the written documents from now on and consider the problem of divergence between the UIP and the SRS.

So the question reduces to:

When the UIP and the SRS disagree, which do you believe?

Believing the UIP?

YES:

The UIP is more concrete than the SRS and is thus easier to validate by the user and client (U&C).

U&C acceptance of UIP is more credible than U&C acceptance of SRS.

Believing the UIP?

NO:

Not all behavior of UIP is intended.

Some behavior is an artifact of rapid prototyping.

Some behavior is debris left over from discarded functionality.

Believing the UIP? Cont'd

NO, Cont'd:

Behavior that is missing cannot be assumed as not intended, because a UIP is *not* intended to cover all behavior, only the less well understood behavior.

Believing the SRS?

YES:

The SRS is intended to be complete and consistent.

It is intended to cover all and only the behavior of the application.

Believing the SRS?

NO:

It is harder for the U&C to validate words describing behavior than a UIP showing the same behavior.

Believing neither?

This is always a possibility and should always be considered.

Clarity

So it is clear that it is *not* clear what to believe among the UIP and the SRS.

Intent is the Key

It is ultimately an issue of what is *intended* behavior in the UIP.

Assume in the following that the UIP and the SRS differ on a behavior.

If the behavior *is* intended to be in the UIP, then I would believe the UIP more than the SRS description of the same behavior.

Intent is the Key, Cont'd

If the behavior is *not* intended to be in the UIP, then I would believe the SRS description of the behavior,

***unless* I have seen reasons for the SRS to be flawed or unvalidatable.**

In any case, you need to know intents and the history of the development of the UIP and SRS, i.e., of the REing process, that led to the UIP and the SRS.

Hopelessness of After-the-Fact

It is clear that which to believe, the UIP or the SRS, cannot be answered after the fact. If there is no documentation on intent, no access to the REs, when the UIP and SRS disagree, there is no way to decide for sure which, if any, to believe.

Only Solution

Therefore, must decide *before* beginning UIPing, at the beginning of REing, what behavior is intended to be in the UIP, to document that, and to set up tracing between the UIP parts, the sentences in the SRS, and the sentences in a new Intent Specification (IS)

We are back to the need for documentation.

Groan!!! There's no escaping it! Sigh...

Solution Goals

Don't want to invent YAREM (Yet Another REing Method).

There are more than enough of them out there.

Want to be able to use whatever method, *M*, we are already using.

Solution Approach

We prepend an intent decision and documentation phase to *M* and add to all steps of *M*, a tracing step.

The tracing step is to establish a trace of individual intents in the IS throughout the UIP and the SRS.

Steps of the Approach

1. **Define application's operational environment and interfaces to other applications.**
2. **Identify to which domains the application belongs.**
3. **Characterize principal properties and main features of an application belonging to these domains.**

Steps of the Approach, Cont'd

4. **Identify which properties are applicable to the application under development.**
5. **Decide which requirements of the identified properties will be prototyped.**
6. **Prototype the application's interfaces and chosen properties.**

Requirements Information

Step 5 is deciding the requirements information to be put into the UIP: the application's

1. **functionality and behavior**
2. **data model, dictionary, and processing capabilities,**
3. **taxonomy and dictionary**
4. **interfaces to other applications, and**
5. **intent, including general information**

General Information

The general information about an application includes such things as:

- **enterprise's state before application existed,**
- **user's work environment,**
- **user's work procedures,**
- **application domain in general,**
- **design concepts,**
- **design goals,**
- **constraints,**

General Information, Cont'd

- **assumptions made,**
- **limitations,**
- **possible tradeoffs,**
- **design options,**
- **decisions regarding human engineering,**
- **operational concept,**
- **rationale,**
- **how to use the application, and**
- **development artifacts.**

Domains

All programs in a domain share common data, attributes, and operations.

Examples of domains are:
simulation systems
information systems
data acquisition systems
UI-intensive systems

An application may be in more than one domain.

Domains, Cont'd

For each domain, from past experience, much is known about data, attributes, and operations that are needed by any application in the domain.

Domains make identifying their portion of the requirements more cut and dry.

Domains, Cont'd

For many domains, there are libraries of data definitions, procedures, and functions that simplify the programming of the requirements derived from the domain.

Case Study to Validate Approach

Ravid took almost completed project in which he had participated, and used it for an after-the-fact case study.

Very early in this project, he had noted the problem addressed by this work,

He carried out the approach, redoing some problematic steps in the original development.

Validation Case Study, Cont'd

He got better results than the first time around, and he fixed the application.

Ravid is convinced that had they followed the approach from the beginning, the problems would not have been as severe.

Caveat

Of course, in such an after-the-fact case study, we cannot be certain that he did not do better the second time around because of learning from the first-time mistakes.

However, results are good enough to warrant further and independent case studies from projects from the beginning.

Always Introspection

Since it is infeasible to do a large project twice, with and without the approach, not to mention that it would be impossible to factor out other influences in such a small sample size, these case studies will also be introspective.

Case Study Details
Target Scene Generator (TSG)

TSG simulator, a medium scale UI-intensive system, 200,000+ lines of code.

Although systems like TSG are not that common, TSG possesses many properties of more common systems.

Therefore, the TSG project is a good source for many typical examples.

TSG Details

TSG is a physical effect simulator that generates high-fidelity, multi-object IR images.

An IR image generated by TSG is composed of several objects that represent

- **targets,**
- **decoys, and**
- **a background.**

TSG Basic Requirements

TSG allows independent control of appearance and location of objects at any position within a large field of view.

IR images generated by continuously changing radiometric and dynamic characteristics of simulated objects in a controlled fashion.

TSG is to serve as an aid for testing and evaluating performance of electro-optical missile seekers.

Main components of TSG

- **static and dynamic optical elements,**
- **IR radiation sources,**
- **electro-mechanical servo-controlled units,**
- **servo-control systems,**
- **several computers, including**
  - **servo controllers,**
  - **radiation source controllers,**
  - **special-purpose embedded systems, and**
  - **standard PC-based workstations, and**
- **software on the computers.**

TSG Functions

Describe TSG by analogy to a VCR.

Local control (LC) = VCR's front panel, enabling basic VCR operations.

Remote control (RC) = VCR's remote control, enabling most LC operations and much more.

TSG Operations

Manual operation = each operation done on RC.

Automated operation = user can define (time-triggered) sequences of operations that VCR performs on its own.

TSG Overview

TSG = a recording studio, with several VCRs and other equipment, all operated by technician from single computer,

as opposed to a pile of remote controls, as is typical in a home.

TSG's UI

Most of TSG's HCI concentrated in host computer.

Host computer serves the TSG operator and supports operation of the entire TSG from a single post by means of a keyboard and a mouse.

Consequently, host computer is UI intensive.

Therefore, host computer is a potential subject for UIPing.

More on TSG and Approach

For more details on TSG, and the adaptation of the approach to the development method for the TSG, please consult Ravid's M.Sc. thesis, available at

ftp://www.cs.technion.ac.il/pub/misc/dberry/alon.ravid/Thesis.doc

Conclusions about UIPing

UIPing worked for us.

Primary benefits of UIPing were

- **improved communication with customers,**
- **completion of REing within reasonable amount of time, and**
- **validation of requirements and design concepts.**

Benefits of UIPing

During UIPing, we discovered

- **major misunderstandings with customer and**
- **many contradicting requirements.**

Actually first UIP was totally wrong and was thrown away.

Production version development started from scratch based on second UIP.

Lessons Learned about Methods

Recall that the solution approach is adapted to existing methods.

The chosen methods:

- **range from informal to formal, depending on needs of application.**
- **must be able to deal with the kinds of information in the UIP.**
- **should aid and require modeling the difficult aspects of the application.**

Other Lessons Learned—1

Lack of communication with U&C at beginning of REing exposed two serious disadvantages of original methods.

Other Lessons Learned—1, Cont'd

1. **Unless U&C sufficiently understand requirements modeling technique, and developers can *see* that they do, developers remain unconvinced that U&C really mean "This is what we want." U&C usually really mean, "You did a professional job. However, we don't understand what you wrote. On other hand, it does mention most of what we requested."**

Other Lessons Learned—1, Cont'd

2. It is pointless to create complete and consistent models for the wrong requirements.

Other Lessons Learned—2

Identification of the application's domains helped simplify

- **REing by allowing us to focus on the TSG's unique properties, and**
- **the SRS by providing a natural grouping of requirements.**

Other Lessons Learned—3

The application's taxonomy helps reveal inconsistencies, by forcing expressing all requirements in a common, agreed upon vocabulary with common, agreed upon definitions. Failure to have one leads to multiple terms for same concept and multiple concepts for the same term.

Other Lessons Learned—4

UIPing causes inflation of functional requirements. However, the additional requirements are those that would be discovered later anyway as users begin to experiment with the software.

Other Lessons Learned—5

The UIP cannot serve as the only requirements model. Even if the UIP is complete, without other models, extracting requirements becomes a reverse engineering activity.

Not Solved

We do not wish to create the impression that if developers follow the proposed approach, all their REing problems are solved.

Not at all.

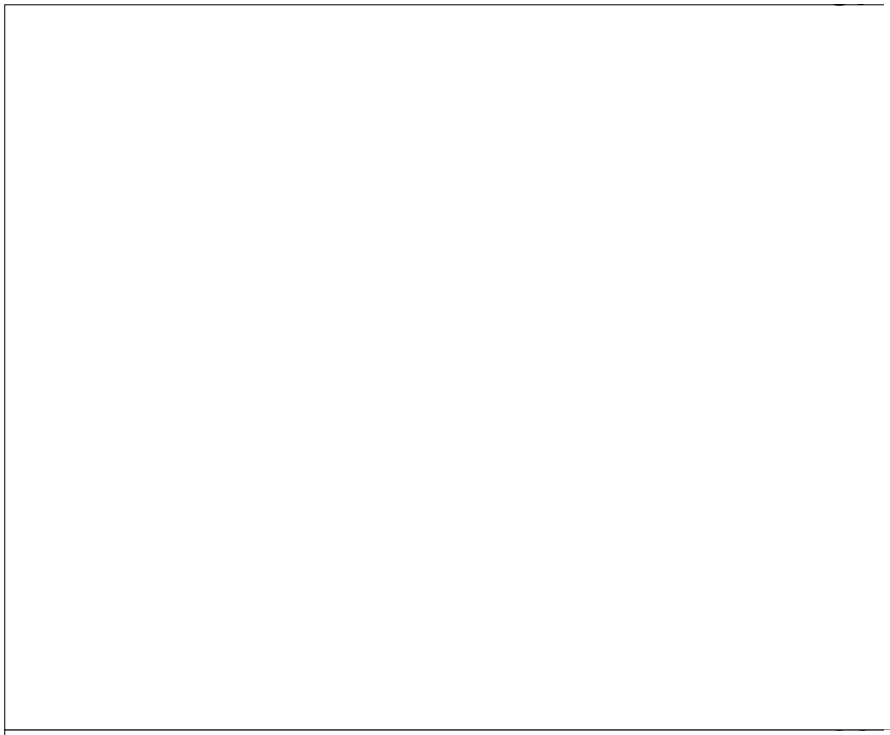
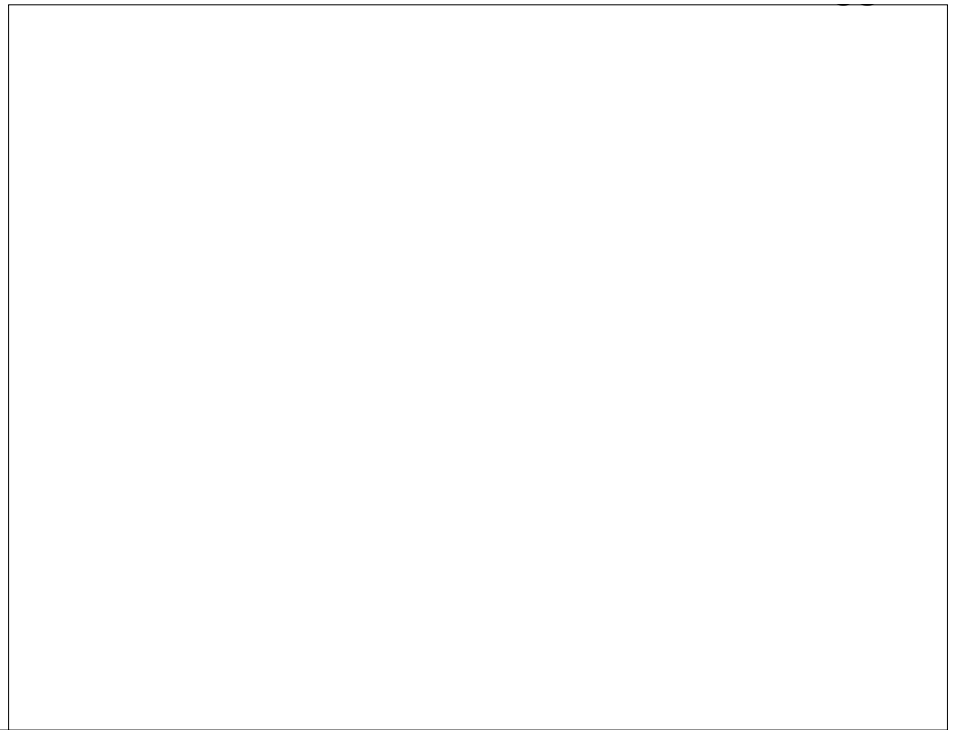
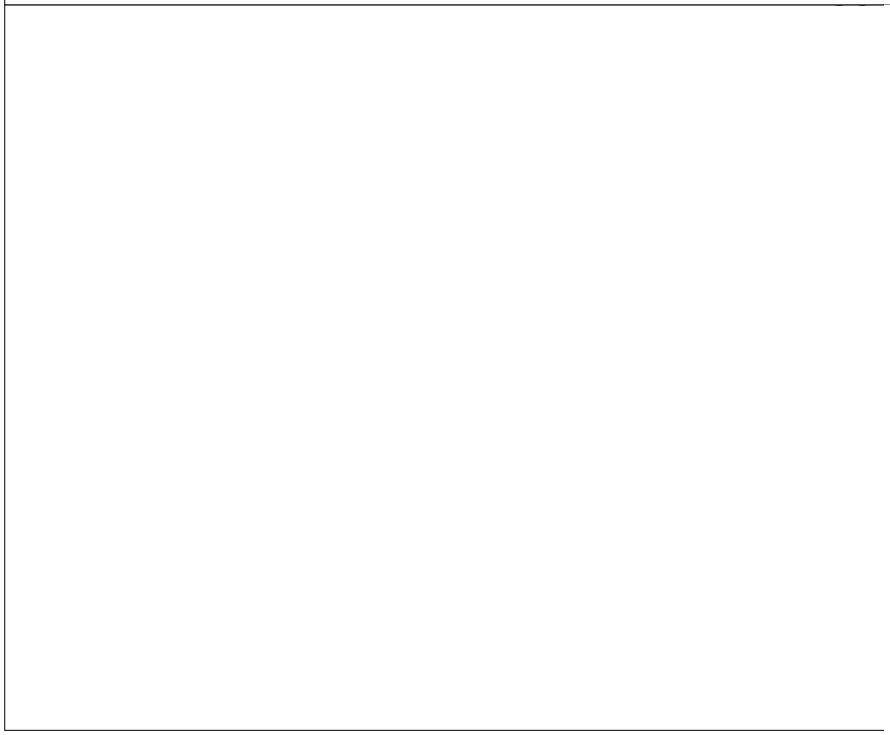
The real problem of finding what has to be done is still left to be solved anew by developers for each project.

Conclusion

We hope that we have convinced you that the approach has the potential to

- **be useful,**
- **yield valuable improvements to UIPing efforts,**
- **address the problems described in the beginning, and**
- **help to decrease these problems.**

Please look at the full thesis and try out the method on your own UIPing efforts.

Acknowledgments

The authors thank Don Cowan, Maria Augusta Nelson, Torsten Nelson, and Daniel Schwabe for their comments on previous drafts of the slides.