Improvement of User Interface Development Methodologies through Rigorous Analysis

Rebecca E. Grinter
E-mail: rgrinter@ics.uci.edu

Richard N. Taylor
E-mail: taylor@ics.uci.edu

Technical Report 93-36

August 11, 1993

Rebecca E. Grinter is supported by a grant from the Science and Engineering Research Council of the United Kingdom. This material is based upon work sponsored by the Defense Advanced Research Projects Agency under Grant Number MDA972-91-J-1010. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Improvement of User Interface Development Methodologies through Rigorous Analysis

Rebecca E. Grinter
Department of Computer Science
University of California, Irvine[1]
92717-3425
E-mail: rgrinter@ics.uci.edu
714-856-5086

Richard N. Taylor
Department of Computer Science
University of California, Irvine[2]
92717-3425
E-mail: taylor@ics.uci.edu
714-856-6429

ABSTRACT

A variety of user interface development methodologies have been described over the last twenty years. Yet it is often difficult to understand exactly how to follow a methodology, to understand how one methodology differs from another, or how to combine two methodologies. This paper shows how methodologies can be characterized by applying a formal analysis technique, developed originally for examining software development methodologies. The technique is illustrated by applying it to the Shneiderman user interface development methodology. As a result of the examination, a number of improvements to the Shneiderman methodology are suggested.

INTRODUCTION

In the last ten years user interface development has grown into an important focus of research in both the computer human interaction and software engineering communities. One area of research is that of user interface development methodologies, which focuses on the processes used in user interface development. Recently, Xiping Song and Leon J. Osterweil have developed a model for characterizing and comparing software design methodologies. The Song and Osterweil model distills out the underlying processes involved in software design, providing an objective means of describing and comparing them.

This paper describes the results of formally analyzing a user interface development methodology using the Song and Osterweil model, explaining how this can aid research in their development. For this study we used Ben Shneiderman's methodology (SUIDM) defined in *Designing the User Interface* [Shn 87] [Shn 92]. Where the particular edition of SUIDM matters discussion is provided.

We begin by outlining the SUIDM life cycle and the unique concepts which it has. Then we introduce the Song and Osterweil model, followed by the formal analysis of the SUIDM using that technique. The model allows us to highlight some strengths of the methodology, including the unique psychological perspective which SUIDM offers, and highlights the problems surrounding module specification. We conclude by examining the benefits of using this technique to taxonomize SUIDM, and offer some insight into future research.

THE SUIDM

SUIDM was developed to address the issues involved in building interactive systems. The 1992 edition of the book reflected the advances in the field, but the user interface development life cycle remained similar.

In this section we outline the user interface development life cycle and then describe an important and unique contribution of the SUIDM to interface design, namely, the syntactic and semantic model of objects and actions (SSOA).

The Life Cycle

The SUIDM consists of an eight phase life cycle directed at building an interactive system. Each phase comprises various tasks, which when completed mark the end of the visit to that phase. Shneiderman points out that system design does not occur sequentially, and that phases may be visited several times throughout the process of design.

The first phase, *collecting information*, focuses on assembling the people and the information necessary for the project as a whole. A variety of methods for collecting information are included in the life cycle. Sur-

[1]Supported by a grant from the Science and Engineering Research Council of the United Kingdom.

[2]This material is based upon work sponsored by the Defense Advanced Research Projects Agency under Grant Number MDA972-91-J-1010. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

veying techniques are supplemented with psychological approaches, task analysis studies, case studies, and encouragement to read professional and academic literature.

It concludes when the designers know the users and their tasks, the structure of their design team, other similar systems and the development schedule. Contract relations documents, highlighting the schedule, tentative deadlines and estimates, as well as an acceptance strategy are produced.

The second phase, *defining requirements and semantics*, begins by establishing application independent issues and specific alternatives. Designers examine prior task analyses, and construct models of the task flows and select the most appropriate representation. Then, using SSOA, designers isolate user's computer knowledge and task knowledge so that they can split application functionality from interface requirements. This is discussed in the next section.

Having elicited the situation-dependent application and interface issues from tasks analysis and the SSOA, the methodology focuses on other necessary system constraints. Reliability, availability, security, privacy and integrity constraints are stressed. Finally, all the specifications are pulled together in guidelines documents and customer and manager agreements documents.

Next the *design syntax and support facilities* phase focuses the design requirements towards specific system alternatives. Designers decide on ways to specify the tasks so that the users find them easy to learn and remember. This is achieved using the SSOA, focusing on knowledge representations which users have about how to perform their tasks. System functionality requirements, including speed and support are also specified. This phase concludes with an intensive review of all the plans.

The fourth phase, *specifying physical devices*, is self-explanatory. The prior phases have sensitized designers to the needs of the user and environment, and the information they have gathered is used to help inform choices.

In the *developing software* phase a top-down modular approach is described. The previous phases have also followed this approach, and it is extended here. Human factors goals for design are emphasized in this phase. They help to interactively test the initial, and successive versions, of the software product.

SUIDM human factors goals include learning time for the system, how fast the system performs, the rate at which users make errors using the system, subjective satisfaction and the ability to remember the system over a period of time. The goals can be used throughout the cycle, as additional to software engineering goals.

Integration and dissemination of systems phase marks the beginning of introducing the system into the environment. The methodology advocates intensive testing to ensure that the system works together before integrating it with the environment.

Designers manage the integration part of the procedure by following support procedures outlined. As well as disseminating the system slowly, the methodology encourages introducing field test manuals and help systems at the same time as the system to ensure an early success.

Once the system is in place, the seventh phase, *nurturing the user community* begins. The SUIDM focuses on providing support for a system, offering user opportunities to suggest improvements and obtain remedies for problems to smooth the integration of the system into the environment and ensure that it survives over time.

The last phase of the life cycle looks to the future by *preparing an evolutionary plan.* By planning ahead, the methodology aims to provide a system which can be continued through several versions. SUIDM encourages designers to use interviews and questionnaires to extract information about potential areas for upgrades within the system. Shneiderman encourages designers to set aside certain time to focus on the issues of general and also evolutionary maintenance.

The SSOA

The syntactic and semantic model of objects and actions (SSOA) allows designers to model knowledge that users have about their tasks, which they keep in long-term memory [Shn 80]. Syntactic knowledge is described as low-level, detailed, device-dependent knowledge. This knowledge includes actions of erasing a character, inserting a line into a text document, and so on. This makes syntactic knowledge difficult to learn, and easy to forget, because the details vary from system to system, and have little intuitive meaning. From a design perspective, knowledge identified as syntactic should be reduced to a minimum.

Semantic knowledge is separated into task concepts (actions and objects) and computer concepts (actions and objects). Task actions focus on the processes which humans carry out. Task objects define the artifacts in the task. For example, consider writing a paper. Task actions include draft and revise, the task object is the paper. Computer-based actions in writing the paper include open file, close file, run file through format program and print. The computer objects would be the file and the word processor.

Users have semantic task knowledge about the work which they do. It is defined as being a well-structured sequence of actions and objects. The aim of user interface design then becomes determining and replicating the semantic knowledge. The computer knowledge aids in determining the application functionality while the task knowledge forms part of the interface requirements.

THE SONG AND OSTERWEIL BASE FRAMEWORK

Introducing the Base Framework

The Song and Osterweil model is a formal analysis technique for characterizing and comparing software design methodologies objectively [SO 92a] [SO 92b] [SO 92c]. It consists of a modeling formalism for comparing software development methodologies against each other, and a base framework which characterizes the *design methods* which comprise a software development methodology.[3]

We are concerned with characterizing SUIDM using the base framework. As such, we do not utilize the modeling formalism, which compares development methodologies after they have been characterized. This characterization is achieved by unpacking each design method in the SUIDM into a set of *method components*, for example, principles, measures and notations. By uncovering the method components we can objectively characterize the SUIDM at a fine level of granularity. To be able to adequately describe all method components within a development methodology the Base Framework has an extendable hierarchical structure of is-a relations.

The base framework consists of two parts, the type framework and function framework, which we describe in this section and which are the used in the next two sections.

The Type Framework

The type framework *identifies method components* within the chosen development methodology. This is formally achieved by using the method component type hierarchy (MCTH), which has four *top-level* method component types, and many *second-level* types, to classify the method components for a particular development methodology, see Figure 1. These basic types, and the extensions allowed through iteration of the framework characterize the internal structure of the methodology. The four top-level method component types are;

1. **Artifacts**: The descriptions or specifications of entities in software design activities.
2. **Concepts**: The ideas which underlie a software development methodology.
3. **Representations**: These provide the means for expressing artifacts.
4. **Actions**: Any processing steps towards developing artifacts.

Concepts, representations and actions have second-level method component types associated with them.

The concepts type is broken down on the nature of the concepts used in various software development methodologies. Song and Osterweil define the second-level types as;

1. **Properties**: The desired characteristics of artifacts.
2. **Principles**: Concepts used to produce artifacts which have the properties desired by customers and designers.
3. **Criteria**: Rules which designers use to decide what constitutes an artifact.
4. **Guidelines**: Pre-defined strategies, heuristics or techniques advocated for use in identification and specification of artifacts.
5. **Measures**: Standard measures which quantitatively compare or evaluate the quality of artifacts.

The representation second-level types distinguish between different types of notation;

1. **Structural**: Diagrammatic notations used to capture the artifact.
2. **Mathematical**: Mathematical Formalisms used to capture the artifact.
3. **Linguistic**: More informal language, English or templates, used to capture the artifact.

The action type has many second-level types associated with it which depend on the way technical actions are described in the software design methodology. Rather than listing the previously defined types here, we instead list those which are found in the SUIDM. In cases where no existing second-level action type adequately characterized the action in SUIDM we used the extension facilities of the base framework to define our own. This is consistent with the definition of the model [SO 92a] [SO 92b] [SO 92c] [SO 92d].

The Function Framework

The function framework *represents the design issues* which the method components of the development methodology address. These can be organized in many ways, but in the end, we decided to divide the categories

[3] Song and Osterweil define software development methodologies as a collection of design methods [SO 92a].

MCTH	SUIDM	
Top-Level	*Second-Level*	*Method Component Type Discovered*
Artifact		Schedule for Development (page 414) Testing Strategy Document (page 414) Requirements Specification (page 414) Physical Device Specification (page 228) User Aids and Manuals (page 357) Systems Documentation (page 415) Evaluations (page 397) Module (page 415) Semantic Computer Knowledge (page 43) Semantic Task Knowledge (page 43) Syntactic Knowledge (page 43)
Concept	Property	Functionality (page 9) Reliability, Availability (page 10) Security, Privacy and Integrity (page 10) Modifiability and Maintainability (page 415) Standardization, Consistency, Portability (page 415) Task Frequency (page 414) Task Flow (page 414) Transaction Unit (page 414)
	Principle	Scientific Reductionism (page 27) SSOA (page 43) Iterative Top-Down Modular Design (page 397) Empirical Experimentation (page 411)
	Criteria	Semantic Object and Action Details (page 51) Syntactic Details (page 47)
	Guidelines	Human Factors Goals (page 10) Interview Users (page 407) Schedules and Budgets (page 414)
	Measure	Prototype (page 394) Pilot Test (page 394) Realistic Load Test (page 415) Questionnaire for User-Interface Satisfaction (page 411)
Representation		None defined
Action		Model Data Collected (page 414) Derive Context (page 414) Develop and Design Test Strategy (page 414) Define High-level Goals (page) Define Middle-Level Requirements (page 414) Specify Goals (page 414) Select Physical Devices (page 228) Implement Software System (page 415) Training of Users (page 358) Evaluate System and Revise (page 397) Measure Errors and Revise (page 63)

Figure 1: Method Component Type Hierarchy for SUIDM in 1987

according to the design issues' modeling and documentation characteristics [SO 92c] [SO 92d]. This provides a way of characterizing the way in which a development methodology supports the building of a solution through models and documentation of the problem. Our resulting model of the software design life cycle (MSDL), which is an extension of one from [SO 92d], formally shows this, by highlighting the modeling and documentation *domains* and the transformations between them, see Figure 2.

The MSDL consists of five domains which are;

- **Problem Domain**: Elements are the real-world phenomena that the system must address.
- **Problem Model Domain**: The model of the problem made.
- **Requirements Document Domain**: Documents elicited from problem model which completely describe the problems, functions and constraints.
- **Solution Model Domain**: Models devised to solve the problem under the given constraints.
- **Design Document Domain**: These elements are documents which together describe the system and related rationales.

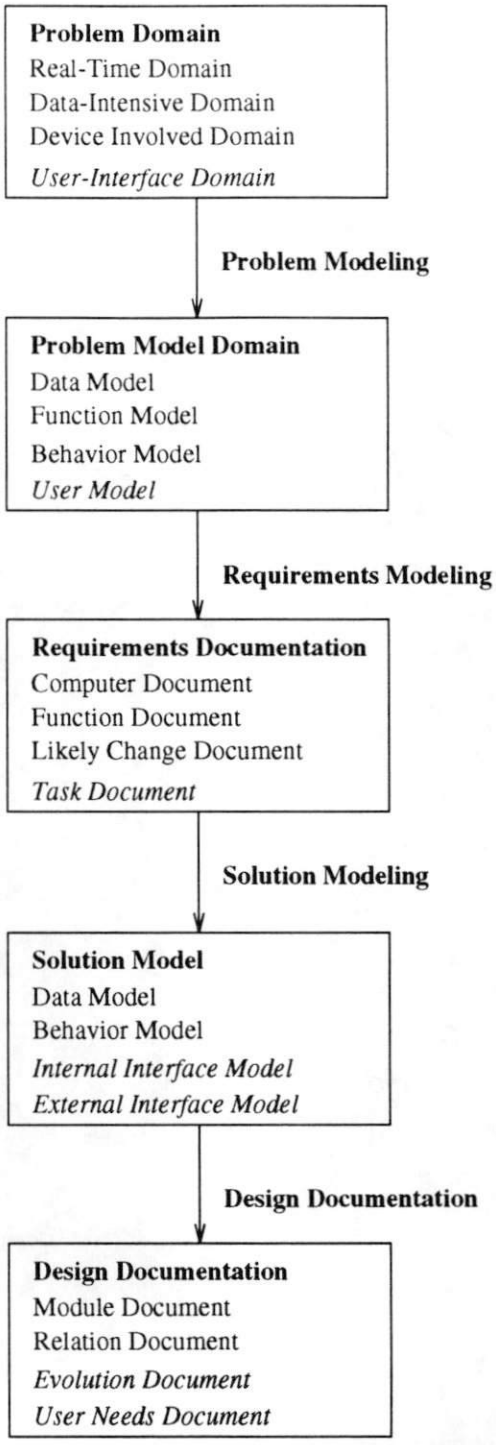

Figure 2: The Enhanced Model of Software Design Life Cycle

ANALYZING SUIDM

In this feasibility study we asked two questions. Firstly, can the MCTH describe SUIDM such that we can identify the strengths and weaknesses of the methodology? Secondly, does the MSDL encapsulate the transformation from application problem to design problem for the SUIDM? In this section we show that the answer to both of these questions is yes.

The SUIDM MCTH

We follow the outline of the five top-level types, and discuss what we found with respect to SUIDM. Figure 1 represents the MCTH of SUIDM we derived. Figure 3 highlights key additions Shneiderman made in the 1992 version of his book [Shn 92]. As well as showing how SUIDM appears in the MCTH we provide evaluation and commentary of the issues which we found during the analysis.

Artifacts

The artifacts embodied in SUIDM have two sources, software engineering and psychology. The first seven artifacts in Figure 1 are documents which summarize activities completed in the life cycle. The artifact module refers to the basic unit of software, the smallest compo-

MCTH	SUIDM 1992	
Top-Level	*Second-Level*	*Method Component Type Discovered*
Artifact		Guidelines Document (page 475)
Concept	Principle	User Interface Management Systems (page 477)
	Guideline	Example Updates
	Measure	Usability Laboratories (page 478) Iterative Testing
Representation	Linguistic	Multiparty Grammars (page 508)
	Structural	Menu Trees (page 509) Transition Diagrams (page 510) Statecharts (page 514) User Action Notation (page 514) DMsketch Notation (page 516)

Figure 3: Additions to the Method Component Type Hierarchy for SUIDM in 1992

nent. In SUIDM, we believe modules could benefit from further elaboration.

Modules can be developed functionally, to solve specific problems [GJM 91]. This approach has problems, including making code which becomes difficult to reuse, because it is reliant on the the specific details of the particular problem. Object-oriented modules support reuse, but have their own problems. They can be harder to derive from the systems requirements at certain levels of abstraction [Som 89]. The SUIDM does not specify whether functional or object-oriented modularity should be chosen. This gives the designer flexibility, but does not encourage the designer to consciously think about the implications of the choice made.

The last three artifacts identified focus on the psychological perspective SUIDM brings to user interface development. In particular the MCTH picks out artifacts which correspond to the SSOA. We believe that this establishes the connection between the SUIDM and the SSOA in a formal way.

In 1992 Shneiderman introduced the guidelines document, see Figure 3. This document comprises screen layout, i/o, action sequences and training specifications. He emphasizes identifying these issues early on in the development cycle to allow for timely resolution of potential conflicts.

Concepts

The first five lines of *properties* in Figure 1 are commonly recognized within software engineering. We likewise recognize the importance of these properties and suggest extending SUIDM to include reusability and interoperability. Building reusable software has become increasingly important as software complexity has increased. This will continue to be important, which the SUIDM identifies with the system evolution plans in the last phase of the life cycle. However, the SUIDM does not define this as a property of current software. We add interoperability because interface software needs to work with other applications resident, and also with other interfaces. The last three properties listed in the table stem from the SSOA.

The SUIDM relies on *principles* which define a general method of software design such as iterative top-down modular design. Note however that while these principles cover much of the methodology, they do not support the module artifact specifically. Top-down modular design can range from step-wise refinement to object oriented design, which produces different module types.

In 1992 a new principle for development, user interface management systems, appeared. These provide assistance to developers by supporting prototyping of interfaces and make the process more visible to a wider group of people. Building user interfaces with this help is a key principle for the SUIDM.

The *criteria* identified concern the SSOA, see Figure 1. The SUIDM articulates clear rules for what constitutes task knowledge. The SUIDM contains no rules for determining what a module should or could be.

The *guidelines* which SUIDM incorporate identify how software designers acquire knowledge about the users and the software requirements. The human factors goals provide a specific checklist of system qualities, from the user's viewpoint, and are supported by the interviewing. Both books discuss these techniques in detail. Schedules and Budgets reflects a concern for managing the methodology, encompassing enforcement, exemption and enhancement within the development process. In 1992 although the guidelines did not specifically change, the examples had been updated to reflect the developments in the field.

The methods for *measuring* system performance in

SUIDM include prototypes, pilot tests, realistic load tests and surveys. Prototyping and pilot testing can be used to measure both user approval or the technical performance of the system. The realistic load test focuses on the technical issues alone. The Questionnaire for User-Interface Satisfaction has been designed especially to gain a subjective understanding of the benefits and problems with a user interface.

In 1992 the SUIDM was extended to include usability labs and iterative testing as ways of measuring system performance, as indicated in Figure 3. Usability labs allow testing of the system in a simulated environment. This supports the SUIDM emphasis on empirical experimentation. Iterative testing guides the usability testing, with repeated experimentation on users until the system and users interact well. We believe that the definition of iterative testing should be extended beyond guiding the usability testing, to support testing of individual and clusters of software modules during the system development phase as well.

Representations

The representations section uncovered a significant problem with the 1987 version of SUIDM. The SUIDM did not define any specific languages, which makes building testing oracles and verifying specifications problematic. Specifications can also be used to bring rigor to the problem, and help in defining the kind of modularization strategy applicable.

The first draft of this paper was written prior to the 1992 book release. The use of the Song and Osterweil model allowed us to identify their absence before the second edition was released. In the 1992 version there are five specific representations, see Figure 3. The linguistic representation, and the first three structural representations concern themselves with developing specifications of how the interface works. The User Action Notation supports the SSOA by specifying how the model will be represented. DMsketch notation has a diagrammatic form but is somewhat different to the other specifications described. It is a technique to help designers exchange and record ideas about how interfaces could be manipulated.

Actions

SUIDM action types are similar to software development ones with the exception of Derive Context. When SUIDM advocates that designers should read academic and professional literature regarding systems development we believe that it is responding to the growing concern that developers and researchers must share ideas and read frequently to take advantage of the newest developments in the field.

The SUIDM MSDL

The MSDL defined in Figure 2 extends a previous version defined in "A Framework for Classifying Parts of Software Design Methodologies" [SO 92d]. We chose this particular version as it had already been iterated over several software development methodologies. In this section we describe the extensions made to it to incorporate SUIDM, which are highlighted in italics on Figure 2[4].

By extending the MSDL we incorporated the separation of concerns between application and interface design. We added the modeling and documentation domains which SUIDM provides to support interface development and create a division between application and interface design.

SUIDM brings an added focus to the problem domain which focuses on the user in a way which goes beyond the problem domain that the system will be used in. We labeled this the *user interface domain*, which is emphasized early on in the SUIDM, particularly by the human factors goals which set guidelines for human computer interaction.

The SSOA forms the *user model*, in the problem model domain. With the introduction of user action notation, the SSOA takes on a formalized definition of the user with reference to tasks.

The output from the SSOA, requirements documentation, forms three artifacts, which concern different aspects of tasks. Together these documents model the tasks which users perform so it is called the *task document*. The Song and Osterweil Base Framework captures the connection between the SSOA, modeling users, and user interface development designing the interface.

The solution model already contained an *interface model*. However, it remained unclear in the Song and Osterweil definition whether this model referred to an internal interface which embedded systems have or a human computer interface. This definition was tightened up to create an distinction between the two, *internal interface* and *external interface*.

SUIDM required two extensions to the design documentation. Firstly, the addition of the evolution document highlights the attention which SUIDM places on designing a system which can evolve over an extended period of time. The user needs document encourages system adoption, by focusing on the needs of users.

[4] The descriptions of different decompositions within each stage does not fully exhaust the currently known variants, but rather gives a few examples. For a complete description refer to [SO 92d].

DISCUSSION

Using the MCTH and extending the MSDL has helped the authors to capture the essence of the SUIDM. We believe that a number of potentially significant results have emerged from this study which we outline below. First we summarize our recommended changes for the SUIDM derived in the previous analysis. Then we outline the benefits of using the Song and Osterweil model to characterize SUIDM. Finally we look at the contributions which the SUIDM has made to the user interface development community.

Recommended Changes

Our chief proposal centers around raising designers' awareness of what a software module comprises with reference to user interface design. The existing criteria focused around SSOA; we suggest that rules about module definition are also needed. The currently unclear connection between the knowledge derived from the SSOA and software development could then be clarified. Supporting information about a module's nature (object-oriented for example) as well as how to perform the decomposition from system requirements to the module are also necessary. Together these three things will establish a key connection between the SSOA and the software aspects of user interface development.

We also recommend expanding the definition of iterative testing to include testing the software iteratively, before making iterative user tests. Finally, we believe that two additional important properties of software should be included, namely reusability and interoperability.

Contributions of Song and Osterweil's Model

Formal analysis of SUIDM using the Song and Osterweil base framework clarified how SUIDM functions and identified its key components. The base framework showed how further research in refining and extending the SUIDM could occur. It also captured the evolution of the model in the 1987 and 1992 versions.

In the definition of SUIDM there are no direct references to human factors goals or SSOA. As the SUIDM definition occurs in later chapters of both books it is not unreasonable to assume that readers will have a strong sense of their inclusion in the description. Both human factors goals and SSOA are discussed comprehensively early on in the books, but the crispness of the relationship between them and the SUIDM is lost later on. Analysis using the base framework identifies their presence, through a systematic evaluation of SUIDM. Identifying method components, and the relations between them link the life cycle together with the human factors goals and SSOA, creating a rigorous definition of SUIDM.[5]

We found that the Song and Osterweil model provided two ways to refine and extend the SUIDM. Firstly, the base framework provides a fine level of granularity which enables the internal consistency and rigor of the methodology to be examined. It allows the researcher to examine the components within the model, and determine what relationships exist between them. Secondly, the modeling formalism can be used to compare SUIDM with other software and user interface development methodologies characterized by the base framework. This allows the development of new methodologies based on methods of handling specific problems defined in other methodologies.

The base framework systematically highlighted the changes made between the 1987 and 1992 version. Internal consistency and rigor led to the detailing of representations in the second version. By examining other representation languages used in different domains of software development, designers may be able to interchange the representations which suit both user interface design and the particular problem domain.

We do not claim that these problems cannot be detected unless this technique is used, but rather that the Song and Osterweil model focuses the designers thoughts by providing a framework in which to consider the issues. This is a valuable tool for organizing perceptions of the SUIDM.

In its current form the Song and Osterweil Model provides static analysis of SUIDM and other development methodologies. One key area for development of the model would be to include a mechanism by which the evolution of the design process over time can be analyzed.

Contributions of SUIDM to User Interface Design

We have identified two major contributions which Shneiderman makes to the user interface development community, the SSOA and human factors goals. We discuss each briefly, with respect their identification within the Song and Osterweil base framework.

The SSOA provides a way to model user's knowledge with respect to the tasks defined. Classifying the SSOA in the base framework caused the MCTH and MSDL to differ significantly from Song and Osterweil's prior instantiations. This highlights the originality of this contribution, and also one of the key differences between

[5] For a detailed examination of the relationships which exist between the various type components refer to [SO 92c].

user interface development methodologies and software development methodologies, which involves incorporating knowledge about users into the methodology.

More generally, human factors goals are characterized in the model as guidelines for determining how the system should work. They influence the creation of specifications, which then influence testing. We believe that in further iterations of SUIDM drawing out details about how they help to measure system performance will prove beneficial.

CONCLUSIONS

We have introduced a user interface development methodology and a model for evaluating, through comparison, software development methodologies. By using the two we have been able to present a rigorous analysis of the benefits provided by and improvements needed within SUIDM. We have also been able to identify some ways in which the base framework needs extension to highlight the unique features of user interface development methodologies. We suspect that further work on user interface development methodologies would discover more changes necessary.

We have identified two potential routes for furthering understanding and creating user interface development methodologies. By adding more user interface development methodologies into the base framework and then using the modeling formalism to compare them we could develop a better understanding of precisely how they differ from each other and what they need to be robust. Also, by using the modeling formalism we could explore whether incorporating method components from other software development methodologies or user interface development methodologies could improve the state of the art in user interface design.

ACKNOWLEDGMENTS

We are particularly grateful to Ben Shneiderman and Lee Osterweil for their reviews can comments which provided us with important insights for improving the paper. The authors would also like to thank Ken Anderson, Craig MacFarlane, Jeanne Pickering and Jim Whitehead, for their comments and suggestions and help in refining the paper.

References

[GJM 91] Ghezzi, L., M. Jazayeri and D. Mandrioli (1991) *Fundamentals of Software Engineering* Prentice Hall. 1991.

[Shn 80] Shneiderman, B. (1980) *Software Psychology: Human Factors in Computer and Information Systems* Little, Brown and Co. 1980.

[Shn 87] Shneiderman, B. (1987) *Designing the User Interface: Strategies for Effective Human-Computer Interaction* Addison-Welsey Publishing Company, 1987.

[Shn 92] Shneiderman, B. (1992) *Designing the User Interface: Strategies for Effective Human-Computer Interaction 2nd Ed.* Addison-Welsey Publishing Company, 1992.

[SO 92a] Song, X. and L. J. Osterweil (1992) "Toward Objective, Systematic Design-Method Comparisons" *Arcadia Technical Report UCI-91-14* University of California, Irvine. 1991.

[SO 92b] Song, X. and L. J. Osterweil (1992) "A Framework for Classifying Parts of Software Design Methodologies" *Irvine Software Symposium* 1992.

[SO 92c] Song, X. and L. J. Osterweil (1992) "Toward Objective, Systematic Design-Method Comparisons" *IEEE Software* Volume 9 No 3. 1992.

[SO 92d] Song, X. and L. J. Osterweil (1992) "A Framework for Classifying Parts of Software Design Methodologies" University of California, Irvine. March 1992.

[Som 89] Sommerville, I. (1989) *Software Engineering 3rd Edition* Addison-Welsey Publishing Company 1989.