Domain Analysis for Transformational Reuse

Melody M. Moore
Spencer Rugaber
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
melody@cc.gatech.edu

Abstract

Domain analysis is an effective technique for enabling both reuse and reverse engineering. This paper shows how domain analysis can provide a framework for combining reverse engineering and forward engineering to implement transformational reuse for information system user interfaces.[1]

Keywords: Reverse engineering, domain analysis, user interfaces, reuse

1. Introduction

Domain analysis—the process of collecting and organizing information about a specific class of problems—has primarily been a technique for facilitating software reuse [PRE91]. The structure of a defined domain can provide the basis for retrieving information about software components, for managing complexity and for making reuse more efficient and effective [NEI89]. Recent work [DEB94], [TIL94], [DEB96], [CLA97] has begun to explore domain analysis as a means of providing a framework for reverse engineering as well.

Transformational Reuse is a process in which components of an existing system are transformed to evolve the system [BIG89]. For example, when migrating a system to a new environment, such as moving from Unix to MS-Windows, many components of the system must change even though the functional requirements remain constant. Such a project requires a reverse engineering step, to understand the existing system components, a transformation step, to determine mappings between old and new components, and a forward engineering step to produce the new system.

The Model Oriented Reengineering Process for Human-Computer Interface (MORPH) [MOO95], [MOO96], [MOO97] is a technique and toolset to support migration, or transformational reuse, of character-oriented user interfaces to graphical user interfaces. This paper shows how domain analysis was used to establish a modeling framework for both the reverse engineering process and the transformation process for the domain of information system user interfaces.

2. Problem Domain

Industry surveys have shown that up to 80 percent of businesses are migrating their information systems to updated platforms and environments [UNI93]. Many of these old systems run on mainframes and have character-oriented user interfaces. Since graphical workstations have become prevalent in industry, a typical migration path is to move the old systems from the mainframe to a graphical environment. A major cost of this migration is that the user interfaces for these legacy systems must be completely rewritten, since the changes constitute replacement of old character-oriented constructs with appropriate graphical components. Since half or more of the code of an interactive system is devoted to implementing the user interface [MEY92], this can be prohibitively expensive.

MORPH supports migration by applying reverse engineering techniques to the legacy application to construct a

1. Effort sponsored by the Defense Advanced Research Projects Agency, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-2-0229. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

model of the existing character-oriented user interface. User interface code is detected by pattern matching and heuristic rules in order to understand the interaction between the user and the system [MOO96]. MORPH also supports transformation between the resulting user interface model and the corresponding user interface construct expressed as a specific toolkit widget. In order to do both of these steps, the MORPH technique must have a robust domain model from which to derive the coding patterns and rules, and also to serve as the vehicle for transformation.

As described in [PRE91], there are two aspects of reusability: infrastructral and operational. The following sections elaborate on how domain analysis was used to support each of these activities in the development of MORPH.

2.1 Infrastructural domain analysis

When creating a library of software components for reuse, infrastructural analysis is the process of "defining, populating, and evolving the repositories of reusable information" [PRE91]. In reverse engineering, infrastructural domain analysis is used for a slightly different purpose: to define the set of abstractions that will be the target of the detection process.

In the case of MORPH, the set of user interface abstractions that occur in character-oriented information system user interfaces needed to be identified, in order to define coding patterns that implement those abstractions in the legacy code. These abstractions are modeled using a knowledge representation language [MOO96], and formalized as a set of coding patterns and rules in the REFINE language [REA90]. The process for the MORPH infrastructural domain analysis is diagrammed in figure 1.

2.2 Operational domain model usage

Once the domain model is defined and the coding patterns are in place, we need a way to store the information collected about a specific application's user interface during the reverse engineering process. This activity is also supported by the domain model, as it provides general classes of user interface components that can be instantiated as patterns are recognized in code. For example, the user interface domain model defines *selection* as one of the abstractions that can be implemented by a variety of coding patterns. Code in a legacy application that implements a menu is recognized during detection as a *selection object*, and an instance of a *selection object* class is placed in the model of the user interface being constructed. The domain representation provides a framework that allows individual components to be modeled.

Another use for the domain model is in transforming the detected user interface components into a new implementation. To continue the example, the menu that was detected during reverse engineering and represented in the model as a *selection* might be transformed into a row of pushbuttons in the tcl/tk toolkit, depending on its attributes. The domain model includes specific toolkit information, and enables the inferencing that perform the mappings to specific toolkit components. Figure 2 presents the use of the domain model in the MORPH operational approach.

3. Approach

The MORPH domain analysis task underwent four phases: an empirical study, a survey of the domain from experts, an

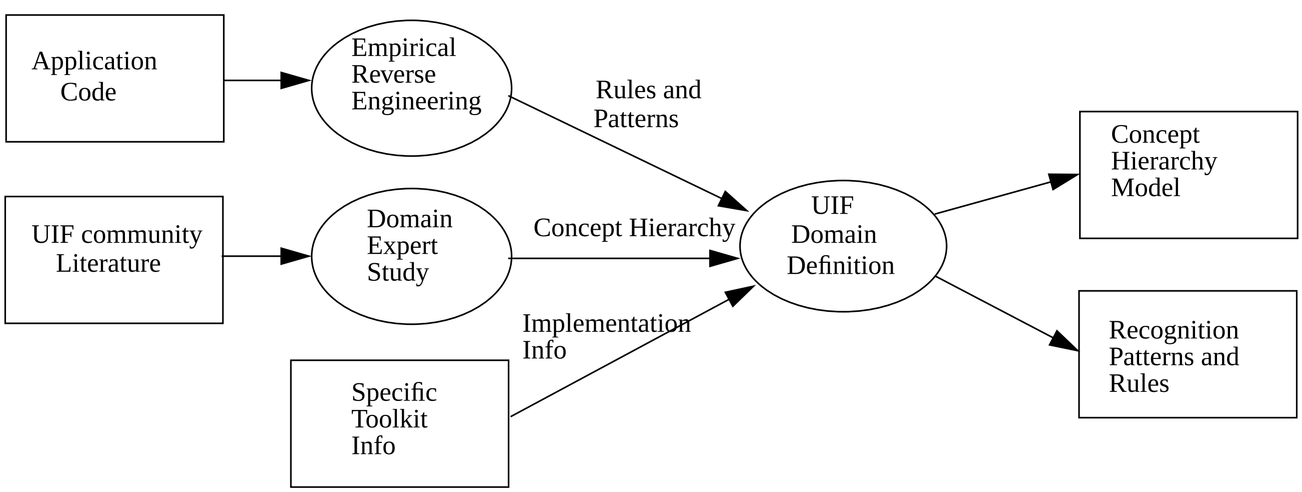

Figure 1: MORPH Infrastructural Domain Analysis Process

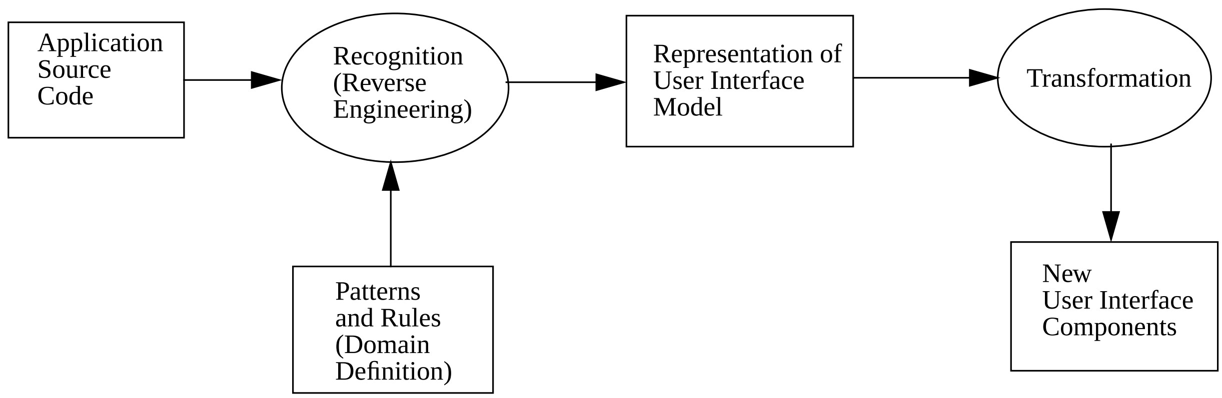

Figure 2: MORPH Operational Process

organizational knowledge engineering phase, and a modeling phase.

3.1 Empirical Approach

The first step in the infrastructural domain analysis was a manual reengineering experiment to determine what code patterns were likely to be found in legacy information systems [MOO95]. Twenty-two applications written in various languages ranging from Cobol, Ada, and Pascal to C, were examined manually (with only the use of a text editor and listings). The applications were also examined dynamically by running the programs while simultaneously looking at the code that implemented each part of the user interface. As user interface components were identified, a list of code patterns that implemented those components was generated. With each successive application, the list of patterns was refined and augmented, and a set of heuristics for recognizing the constructs was created.

To validate the set of rules, several volunteers with a wide range of computing experience were asked to manually execute the MORPH technique, applying the rules to reverse engineer a small database system. The resulting models of the user interface were very consistent, supporting the efficacy of the rules.

3.2 Domain Expert Survey

Although the empirical study yielded a set of rules to identify user interface components that worked for the set of applications that were examined, questions arose about the completeness of the abstraction set. Also, since the applications in the empirical study all had character-based user interfaces, it was likely that there would be additional abstractions from the graphical user interface (GUI) world that should be represented. Previous research in user interface reengineering [MER95] employed interviews with experts in the application domain. As an alternate approach, we performed an extensive search in the user interface literature to discover the missing abstractions.

An interesting side effect of this approach was the introduction of a framework for a concept hierarchy. An *interaction task* is the entry of a unit of information that is meaningful to an application by the user [FOL90]. Interaction tasks classify the fundamental types of information the user enters while interacting with the application. Interaction tasks are defined by what the user accomplishes, not how it is accomplished.[FOL90] lists four basic interaction tasks that we used as a top-level organization:

- Selection
- Text data
- Position
- Quantify

Other works containing surveys of user interface concepts [SHN93],[HIX93],[DIX93] filled in the details of the constructs that implemented each of the basic interaction tasks. For example, the selection interaction task can be implemented by menus, pushbuttons, lists, and other mechanisms. The original rules were classified by the type of construct that they recognized, and new rules were formulated to identify code patterns that might implement the GUI abstractions. Another development during this phase was the addition of attributes, based on work described in [DEB92], that allowed declarative information to help determine appropriate transformations from the abstractions to the specific implementations from a usability standpoint. For example, a selection task that has twenty choices in its selection list is probably best implemented with a cascading menu or a scrolling list rather than a set of pushbuttons. This type of qualitative knowledge, known as *design critics* [FOL90], was also incorporated into the domain model.

3.3 Knowledge Engineering

In order to organize the information gathered from the empirical study and the domain expert study, the knowledge engineering method described in [BRA90] was employed to represent the user interface domain in the CLASSIC knowledge representation language [RES93]. First, the object types were identified and enumerated. From the set of abstractions, the highest-level concepts (classes) were identified, along with roles (attributes) and fillers (specific instances of attribute values). The concepts were then grouped into a cohesive hierarchy. (The hierarchy for the selection basic interaction task is illustrated in Figure 3.) Lastly, the value restrictions for the roles were defined in order to establish the heuristics for the transformation that will occur after the user interface has been modeled.

3.4 Domain Model Representation

Once the concept hierarchy was defined and attributes identified for the components, the concepts were implemented in the CLASSIC language. The basic interaction tasks became the top level of abstraction, and lower-level abstractions were defined as subconcepts to the basic interaction tasks. For example, the following pseudocode shows how the abstract selection interaction task is defined as a concept with its associated roles (attributes):

```
SELECTION-OBJECT: INTERACTION-OBJECT
  Action: Procedural-Action or
          Visible-State-Change
  Choices: integer minimum 1
  Variabiliy: Fixed or Variable
  Grouping: Grouped or Not-Grouped
```

The SELECTION-OBJECT concept inherits attributes from its parent concept, INTERACTION-OBJECT, but further specifies the new attributes unique to a selection interaction task. Concepts that are specializations of the selection interaction task are then defined in terms of the SELECTION-OBJECT concept, as shown in the definition of the MORPH-BASIC-MENU concept below. The narrower definition serves to define values that are acceptable as role fillers to match this more specific concept:

```
MORPH-BASIC-MENU: SELECTION-OBJECT
    Action: Procedural-Action
    Choices: Minimum 2, Maximum 10
    Variability: Fixed
    Grouping: Not-Grouped
```

Each interaction task and object is described in the concept hierarchy in its appropriate place. The CLASSIC code and further details of the MORPH knowledge representation can be found in [MOO97].

In order to test the transformational capabilities of the knowledge representation, two implementation toolkits, tcl/tk and the Java Abstract Windowing Toolkit (AWT) were also added to the concept hierarchy. This was also an interesting exercise from the perspective of testing the concept hierarchy to ensure that it was able to represent the provisions of real toolkits. The toolkit widgets were defined in terms of the concept hierarchy alongside the abstract MORPH concepts. For example, the tk menu widget had the following pseudocode description:

```
TK-MENU: SELECTION-OBJECT
    Action: Procedural-Action
    Choices: Minimum 2, Maximum 10
    Variability: Fixed
    Grouping: Not-Grouped
```

Since this is almost identical to the MORPH-BASIC-MENU description, it is not surprising that CLASSIC infers a sibling connection in the concept hierarchy.

4. Results

An automated MORPH pattern recognizer is currently under development, so the results described here are from the empirical study, the domain expert study, and development of the MORPH concept hierarchy. Experiments to test the validity of the domain model are described in the "validation" section below.

4.1 Initial Rule Base

The results of the original manual reverse engineering experiments yielded a list of rules for identifying the user interface components in the applications that were examined. For example, a common construct we found in the empirical study was a menu, in which the application gave the user a set of choices to select from. Dynamic analysis of one of the programs produced this output:

```
               Menu
[a] answer
[b] browse
[c] make a match
[d] delete record
[e] quit

Your choice => __
```

In examining the application code, one code pattern for a character-oriented menu was identified when encountering the following Pascal code segment:

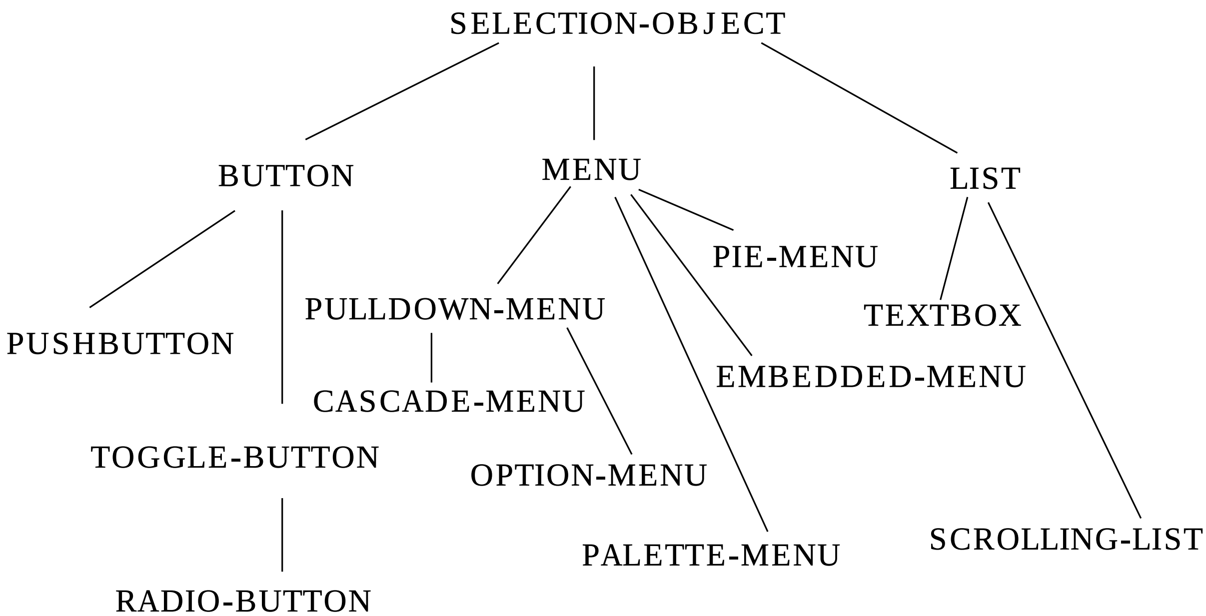

Figure 3: Concept hierarchy for Selection Interaction Task

```
writeln(output, 'Menu');
writeln(output,’[a] answer');
writeln(output,’[b] browse');
writeln(output,’[c] make a match');
writeln(output,’[d] delete record’);
writeln(output,’[e] quit');

choice := getanswer(lastchoice);

case choice of
  'a': answer;
  'b': browse;
  'c': match;
  'd': delete;
  'e': writeln(output)
end
```

This code segment implements a simple menu, the first set of `writelns` displaying the choices to the user, and the case statement implementing the user input and selection. The `getanswer` routine implements input validation for the characters that the user types. Through dataflow analysis, the `choice` variable is identified as a *user input variable:* a structure that gets its value either directly or through assignment from an input statement. The fact that the `choice` variable’s value comes from the user and then is used in a decision is significant in the user interface; it implements a selection of some kind. Thus, an informal English description of the rules to recognize the above constructs would be:

```
 If a statement performs text output
    then identify text block

If an output statement follows a text
 block
    then include that output statement
          in the text block

 If a case statement contains a User
 Input Variable for a discriminator
    then identify Selection
```

The last rule illustrates recognition of an attribute of the selection (number of choices) that could give information about appropriate implementation choices for this abstraction during the transformation process.

The initial rule base constructed from the empirical study consisted of sixteen rules similar to the ones listed above. For more detail on MORPH’s rule-based detection, please refer to [MOO96].

4.2 Building a hierarchy

With the four basic interaction tasks as the basis for organization, user interface components described in numerous surveys were categorized into a hierarchy. Relationships between components sometimes made this difficult, as in the case of the mutually-exclusive pushbutton (known as a *radio button* in many toolkits). The property of mutual exclusion occurs only when these widgets are grouped; pressing one radio button in a group un-presses any others that were previously pressed. The radio button falls under the toggle button (a pushbutton with only two states) with attributes denoting mutual exclusivity in groups. This is important because the presentation, or appearance, of the radio button is different from a regular pushbutton in many toolkits and therefore we need to identify the correct semantics.

There was also some overlap in the abstractions. A *textbox*, an area for the user to input text, could be used to enter unvalidated data (such as an address), or could also be used as a selection mechanism (as when entering a date or typing a selection from a list of choices). Therefore some of the lower-level abstractions may have multiple parents in the concept hierarchy. (Textbox in figure 3 above is shown only in the SELECTION-OBJECT hierarchy; it actually has another parent in the TEXT-OBJECT hierarchy as well.) An advantage of CLASSIC is that it directly supports the representation of this mulitple-parent hierarchy.

Code patterns for the additional concepts identified in this stage were added to the pattern recognition set. Most of these additions were to detect attributes of one of the basic interaction tasks that might differentiate among lower-level abstractions. For example, a small static number of choices in a selection might indicate that an option menu is appropriate, whereas a dynamic number of choices suggests a more flexible mechanism such as a scrolling list. One of the rules that was added as a result of detecting attributes was:

```
For each choice in the list of an
identified selection case statement
    increment number-of-choices
```

This rule implements part of the heuristic described above to determine the appropriate implementation mechanism for the selection component. The `number-of-choices` attribute is then stored in the knowledge base with the information for that selection object.

4.3 Operational strategy

Once the framework of the concept hierarchy and the coding patterns and rules have been defined, the MORPH reverse engineering technique [MOO95] can be applied to legacy systems code. As the pattern matching identifies potential interaction tasks and their attributes, the rules generate CLASSIC code that creates individuals in a knowledge base describing each interaction. For example, the code segment implementing the text menu presented earlier in the paper is recognized as a SELECTION-OBJECT, and the rules generate the following code to create a CLASSIC individual representing this interaction object:

```
(cl-create-ind 'OBJ1
   '(and SELECTION-OBJECT
        (fills action Procedural-Action)
        (fills number-of-choices 5)
        (fills variability Fixed)
        (fills grouping Not-Grouped)
    )
)
```

The interaction object (OBJ1), is identified as a selection interaction task and the role fillers are determined from the attributes found in the code.

4.4 Validating the concept hierarchy

In order to determine if the concept hierarchy was complete enough to represent GUI as well as character-oriented interfaces, two popular widget toolkits were defined in terms of the hierarchy and added to the knowledge base. The selection and text data basic interaction tasks for the tcl/tk toolkit were added first, to see if the model would support these general abstractions. Then, the Java AWT widgets were added. Although translation to Java is not a current goal of MORPH, the AWT widget set is significant because it is very recent. Both toolkits fit into the concept hierarchy, although minor additions to some of the attributes were made in order to be able to differentiate some of the widgets.

After the toolkits were defined in terms of the concept hierarchy, classification and subsumption inferencing were used to test the transformational capabilities of the model. The CLASSIC system allows queries to be composed in the terminology of the domain model, so it is possible to create a new concept with particular attributes (as we would during the detection process) and find the closest match in either the concept hierarchy or in a specific toolkit. As an illustration, the CLASSIC knowledge base can be queried to determine the hierarchical ancestors of the OBJ1 selection object in the previous example:

```
Classic> (cl-ind-ancestors @obj1)
(@c{THING} @c{CLASSIC-THING}
@c{INTERACTION-OBJECT}
@c{SELECTION-OBJECT}
@c{MORPH-BASIC-MENU}  @c{TK-MENU} )
```

The results show that OBJ1 is most directly related to the lowest-level concepts, MORPH-BASIC-MENU, and also the toolkit widget TK-MENU. This example also shows how the concept hierarchy can suggest replacement toolkit widgets for interaction objects detected in legacy code. These experiments showed that transformations can be suggested with varying amounts of information about attributes.

4.5 Validating the Domain Model

In addition to the original manual experiment validating the application of the rule set, there needs to be a validation of the entire domain model. A toolset implementing the

MORPH technique is currently under development, and part of the testing of the toolset will include running a suite of application code through MORPH, including applications in the 80-100,000 line range. The percentage of user interface constructs recognized by the automated analysis process will be measured, and changes to the domain model will be noted. We hope to show convergence of the domain model with each successive application of the MORPH technique requiring less and less modification to the domain model.

5. Conclusions and Future Work

The goal of using domain analysis to support both reverse engineering and reuse was met by the combination of empirical methods and domain-expert knowledge. Initial experiments have shown the resulting domain model for user interfaces to be robust enough to express the constructs of two commonly-used GUI toolkits. Furthermore, the concept hierarchy supports the transformation process necessary for transformational reuse. More strenuous domain model validation experiments will be conducted when the implementation of the MORPH toolkit has been completed. Additional widget toolkits, such as MOTIF and MS-Windows, will also be added to test the flexibility and coverage of the domain model.

6. References

[BIG89] Biggerstaff, Ted, and Perlis, Alan. *Software Reusability*, ACM Press, Frontier Series, Addison Wesley, 1989.

[BRA90] Brachman, Ronald, McGuinness, Deborah, Patel-Schneider, Peter, Resnick, Lori, and Borgida, Alexander. "Living with CLASSIC: When and How to Use a KL-ONE-Like Language", *Principles of Semantic Networks*, J. Sowa, ed., Morgan Kaufmann Publishers, 1990.

[CLA97] Clayton, Richard; Rugaber, Spencer; Taylor, Lyman; and Wills, Linda. "A Case Study of Domain-based Program Understanding", in *Proceedings of the International Workshop on Program Comprehension*, Dearborn, Michigan, May 1997.

[[DEB92] deBaar, Dennis; Foley, James D.; and Mullet, Kevin E. "Coupling Application Design and User Interface Design", *Proceedings of CHI '92*, May 3-7, 1992.

[DEB94] DeBaud, Jean-Marc; Moopen, Bijith; and Rugaber, Spencer. "Domain Analysis and Reverse Engineering", in *Proceedings of the 1994 International Conferences on Software Maintenance*, Victoria, Canada, IEEE Computer Society Press, September 1994.

[DEB96] Debaud, Jean-Marc. "Lessons From a Domain-based Reengineering Effort", in *Proceedings of the Third Working Conference on Reverse Engineering*, Monterey, CA, Nov 8-10 1996.

[DIX93] Dix, Alan; Finlay, Janet; Abowd, Gregory; and Beale, Russell. *Human-Computer Interaction*, Prentice Hall International (UK) Limited, 1993.

[FOL90] Foley, James D., van Dam, Andries, Feiner, Steven K., and Hughes, John F. *Computer Graphics Principles and Practice*, Second Edition, Addison-Wesley Publishing Company, Addison-Wesley Systems Programming Series, 1990.

[HIX93] Hix, Deborah, and Hartson, H. Rex. *Developing User Interfaces - Ensuring Usability Through Product and Process*, Wiley Professional Computing Series, John Wiley and Sons, Inc., 1993.

[MER95] Merlo, Ettore; Gagne, Pierre-Yves; Girard, Jean-Francois; Kontogiannis, Kostas; Hendren, Laurie; Panangaden, Prakash, and DeMori, Renato; "Reengineering User Interfaces" *IEEE Software*, Vol. 12 No. 1, January 1995

[MEY92] Myers, Brad, and Rosson, Mary Beth. "Survey on User Interface Programming", *Proceedings of SIGCHI 1992, Human Factors in Computing Systems*, Monterey, CA, May 1992.

[MOO95] Moore, Melody. "Reverse Engineering User Interfaces: A Technique", in *Proceedings of the 1995 Software Developer's Conference*, San Francisco, CA, April 1995.

[MOO96] Moore, Melody. "Rule-Based Detection for Reverse Engineering User Interfaces", *Proceedings of theThird Working Conference on Reverse Engineering*, IEEE Computer Society Press, Nov 8-10, Monterey, California, 1996.

[MOO97] Moore, Melody and Rugaber, Spencer. "Using a Knowledge Representation for Understanding Interactive Systems", in *Proceedings of the International Workshop on Program Comprehension*, Dearborn, MI, May 1997.

[NEI89] Neighbors, J.M. "DRACO: A Method for Engineering Reusable Software Systems", in *Domain Analysis and Software Systems Modeling*, IEEE Computer Society Press, Los Alamitos, California, 1991. Reprinted from Association for Computing Maehinery, with permission from Addison-Wesley Publishing Co., Reading MA, 1989.

[PRE91] Prieto-Diaz, Ruben, and Arango, Guillermo. *Domain Analysis and Software Systems Modeling*, IEEE

Computer Society Press, Los Alamitos, California, 1991.

[REA90] Reasoning Systems Inc. *REFINE User's Guide*, Copyright Reasoning Systems Inc., 3260 Hillview Avenue, Palo Alto, CA 94304, 1990.

[RES93] Resnick, Laurie Alperin et al. *CLASSIC Description and Reference Manual for the Common LISP Implementation Version 2.1*, AT&T Bell Labs, Murray Hill, N.J., May 15, 1993.

[SHN93] Shneiderman, Ben. *Designing the User Interface, Strategie for Effective Human-Computer Interaction, Second Edition*, Addison Wesley, 1993.

[TIL94] Tilley, Scott. "Domain Retargetable Reverse Engineering II: Personalized User Interfaces" in *Proceedings of the 1994 International Conferences on Software Maintenance*, Victoria, Canada, IEEE Computer Society Press, September 1994.

[UNI93] Uniforum, "Uniforum Research Released: '93 to be the Year of Change", *UniNews,* Vol VII, Number 6, April 7, 1993.