# Using Knowledge Representation to Understand Interactive Systems

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

**Abstract**

*System migration presents a myriad of challenges in software maintenance. The user interfaces of interactive systems can undergo significant change during migration. Program understanding techniques can be used to create abstract models of the user interface that can be used to generate a new user interface on the target platform. Using a knowledge representation to model the abstractions has the advantage of providing support for transformation to the new user interface environment. This paper details the knowledge base and representation incorporated into the Model Oriented Reengineering Process for HCI (MORPH)[1] toolkit, which supports program understanding for interactive systems. It illustrates the process with an example transforming MORPH abstractions to the Java Abstract Windowing Toolkit (AWT).*

**Keywords:** User Interface, reengineering, migration, reverse engineering, knowledge bases

## 1. Introduction

The migration of interactive systems presents a unique challenge for program understanding. Often the user interface must be completely replaced, even though the functional requirements of the system remain the same. Manual reengineering of the user interface is costly, since half or more of the code for an interactive system is devoted to the user interface [6]. Using an automated process to extract information about user interface tasks and constructing an abstract model can significantly reduce the effort required for migration [5].

Even though the tasks the user accomplishes do not change, the way that the user interacts with the system may be altered significantly. Conceptually, the migration process creates a mapping from the user interface tasks in the old system, to the implementation of those tasks in the new system. Nuances of the user interactions in the original system can help determine the most appropriate replacements in the new system. A transformation process is needed to determine the best mapping for user interface replacement.

Representation is also a key issue in any process for program understanding [14]. The quality and utility of an abstract model derived from code can be largely determined by the capabilities of the representation for the model. Therefore, a representation method must be chosen carefully to adequately support the program understanding task being undertaken [9]. A representation method that accommodates both the capture of detailed information about the interactions and also provides the mapping capability is knowledge representation [7]. The Model Oriented Reengineering Process for HCI (MORPH) [10] provides a framework for deriving abstract models of user interfaces and support for transformation to the new graphical implementation. This paper focuses on the MORPH abstract modeling method using the Classic [13] knowledge representation language and the advantages and issues in using a knowledge representation for program understanding.

### 1.1 Terminology

*Interaction tasks* define what the user accomplishes while interacting with a user interface; for example, select-

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.

ing from a list of choices, or entering data [3]. *Interaction objects* are the controls (such as buttons, textboxes, or sliders) that define how the user interacts with the system [2]. Interaction tasks are defined by the application requirement, and do not change when the system is migrated to a different platform. Interaction objects are defined by the implementation environment and may be completely replaced during migration. Interaction objects can be classified by the interaction task they implement, differentiated by identifiable attributes. For example, a menu is an interaction object that implements the *selection* interaction task, with the attributes "small, static choice set". A scrolling list is an interaction object that also implements the selection interaction task but can be used to implement a "potentially large, dynamic choice set".

Specific interaction objects are implemented in *toolkits* that allow composition of new user interfaces, such as Motif, MS-Windows, and the Java Abstract Windowing Toolkit (AWT). *User Interface Management Systems (UIMS)* are environments that allow rapid development and modification of graphical user interfaces (GUIs) [3]. The most ubiquitous form of GUI is the Windows, Icons, Menus, and Pointers (WIMP) interface, which evolved from the Apple MacIntosh. A *legacy system* is an application that has been developed and maintained over a period of time, typically its original designers and implementors are no longer available to perform the system's maintenance. Often specifications and documentation for a legacy system are outdated, so the only definitive source of information about the system is the code itself.

## 2. Approach

### 2.1 The MORPH Process

MORPH is a process for reengineering the user interfaces of text-based legacy systems to graphical user interfaces, and a toolset to support the process. There are three steps in the reengineering process [8]:

- *Detection* - Also called program understanding, analyzing source code to identify user interaction objects from the legacy system.
- *Representation* - Building and expressing a model of the existing user interface as derived from the detection step.
- *Transformation* - Manipulating, augmenting, or restructuring the resulting model to a graphical environment.

In the *detection* step, identifying interaction tasks from code enables construction of an abstract model of the user interface (expressed by a *representation model*). From this abstract model, *transformations* can be applied to map the model onto a specific implementation. It has been shown that interaction tasks can be detected from character-oriented application code by defining a set of coding patterns and replacement rules [10]. Attributes of interaction tasks can also be detected from code, in order to construct a higher quality abstract model of the user interface [11]. With more detail in the model, the transformation step can select more appropriate interaction objects for the implementation of the new user interface.

As illustrated in figure 1 below, the process begins with extracting the user interface from the computational legacy code, using program understanding techniques to

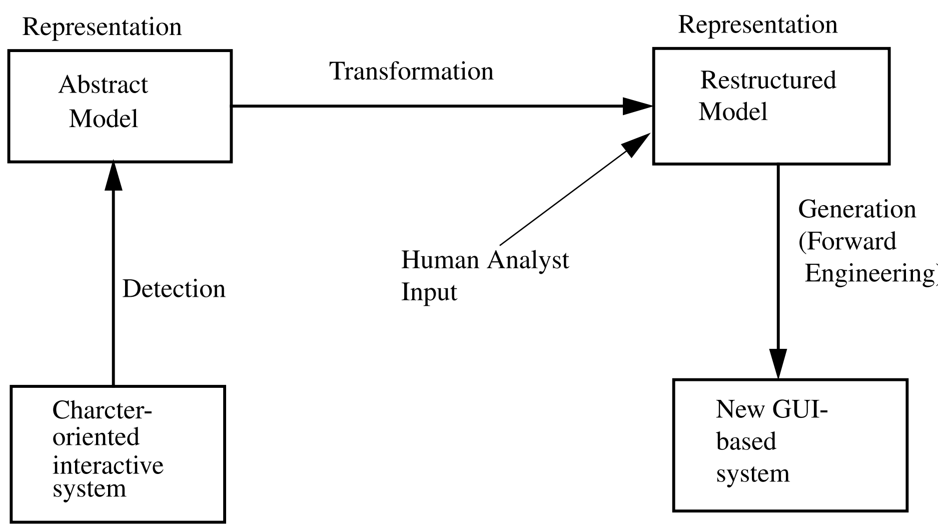


**Figure 1: The MORPH Reengineering Process**

build an abstraction, or model, of the existing interface. This path is depicted by the detection edge and the abstract model box in Figure 1. The legacy system model can then be transformed to a model structured to support a WIMP interface, shown by the transformation edge. Input from the human analyst is added at this stage, to define presentation details and to enhance the model. Once the model has been restructured, a forward-engineering tool, such as a User Interface Management System (UIMS), can be used to automatically generate a new graphical interface for the system.

## 2.2 Abstraction Hierarchy

A key element of the MORPH process is the set of user interface abstractions that can be recognized from legacy code and represented in the user interface model. These abstractions start with the four basic user interaction tasks described in [3]:

- Selection - the user makes a choice from a set or list of choices
- Quantification - the user enters numeric data
- Position - the user indicates a screen or world coordinate
- Text Entry- the user enters text from an input device

Coding patterns that describe the implementation of each of these basic interaction tasks can be used to detect these tasks in text-based user interfaces [10]. In a graphical user interface, however, there are many different possibilities for the implementation of a particular interaction task. For example, a selection task may be implemented by a bank of pushbuttons, or by a menu. Therefore, once the interaction tasks are identified, then the code can be examined for further clues to possible attributes of the interaction object that will be most appropriate in the GUI. For example, if a selection task is identified, then the number of choices may determine whether that interaction object should map to a menu or a scrolling list. MORPH maintains a hierarchy of concepts, composed of the set of abstract interaction objects categorized under the basic interaction tasks. These abstractions are described in the MORPH knowledge base. In order to allow transformations between the abstractions and specific GUI toolkits to be accomplished by inferencing, components of each toolkit are also described in the knowledge base. Figure 2 below shows the organization of the MORPH abstraction hierarchy. As the interaction tasks are identified, attributes are collected to pinpoint the appropriate abstract interaction object. Inferencing is then used to map to the appropriate implementation of the abstract object in a particular toolkit, such as Java AWT.

## 2.3 Declarative Models

In [2], deBaar, Foley, and Mullet present a declarative approach to designing a user interface by specifying a data model, and then using an inference engine to generate the user interface from the data model. This method also has applicability to reverse engineering, since it is straightforward to detect a data model from code. Detecting interaction tasks and objects of a user interface from text-based code depends largely on the data structures involved in I/O. Identification of attributes of variables that are used in I/O with the terminal may be used to determine which components of a specific toolkit are appropriate to substitute in a GUI interface. Using this approach, we are able to identify salient features that can differentiate between interaction objects that implement the same interaction task. For example, in the selection interaction task, a large, dynamically sized choice list indicates a scrolling list, whereas a smaller, static choice list may be best implemented with a basic menu. A table of attributes for each interaction task was generated based on the declarative model idea. The Selection task table is presented as an example (Table 1) in the Results section below.

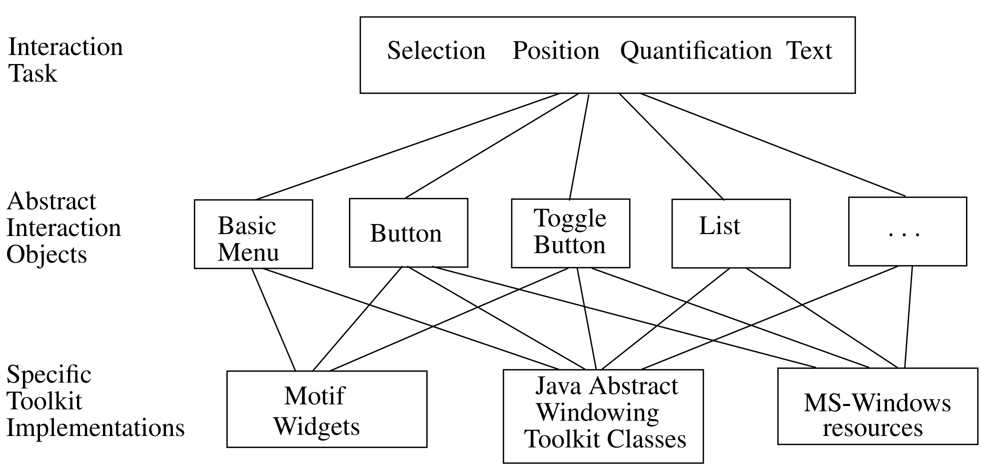


**Figure 2: MORPH Concept Hierarchy**

## 2.4 The CLASSIC Knowledge Representation Language

The addition of a knowledge base strengthens the declarative approach. Frame-based (slot-and-filler) knowledge base systems work well with pattern-based detection methods [4]. If data structure and attribute patterns for interaction tasks and objects can be identified, then a frame-based knowledge representation can be used to describe them. This knowledge base can then be used to aid in the detection of user interface components from legacy code, and then used to identify the appropriate replacement interaction objects within a specific toolkit such as MS-Windows or Motif.

The CLASSIC Knowledge Representation from AT&T Bell Laboratories [13] is a frame-based system that is particularly well suited for our purposes. CLASSIC focuses on objects rather than logical sentences, allowing interaction tasks and objects to be described in an object oriented fashion. CLASSIC is a deductive system based on the definition of *concepts*, which are arbitrarily complex, composable descriptions of object classes. *Roles* are properties that can be used to define attributes and restrictions on the concept descriptions, and *individuals* are instantiations of objects.

The power of CLASSIC for the MORPH knowledge base is in the organization of concepts and individuals into a hierarchy. More abstract, or general, concepts will be higher in the hierarchy, while more specific concepts and individuals will be lower. This allows individuals to be matched with classes that they satisfy. For example, the "selection task" is a high-level concept, while the concepts for the abstract objects "menu" and "pushbutton" are underneath "selection". Concepts and individuals for specific toolkit components, such as the "Choice" in Java AWT, are underneath the abstract objects in the hierarchy.

CLASSIC also provides a variety of inferencing capabilities. Its completion inferences include contradiction detection, combination, inheritance, and propagation. It also provides *classification*, which allows concepts to be identified if they are more general or more specific than a given concept, and *subsumption*, which answers questions comparing the relative order of two concepts in the hierarchy. Classification in particular can be used to infer the most appropriate specific toolkit component from a description of an abstract interaction object.

# 3. Results

The MORPH knowledge base was built in CLASSIC according to a domain analysis for user interfaces. This section describes the method for organizing the user interface hierarchy and gives an example of the representation.

## 3.1 Knowledge Base Hierarchy

In building the concept hierarchy, we implemented two approaches to analyzing the user interface domain:

- Bottom-up - Since the domain was limited to character-oriented interfaces, a study was performed on 22 legacy systems, using static and dynamic analysis to identify user interface components. A taxonomy of possible user interaction tasks and interaction objects was built from entities that were found in the code.
- Top-down - in order to make the taxonomy more complete, user interface community literature was searched to find definitions for interaction tasks, objects, and their associated semantics. The taxonomy was augmented to include interactions that weren't discovered in the code study, but could conceivably be a part of a character-oriented interface.

According to the knowledge engineering methodology described in [1], roles and concepts for each of the interaction tasks and objects were defined for the CLASSIC knowledge representation. Definitional attributes became roles, and interaction tasks and interaction objects became concepts.

## 3.2 Knowledge Representation Language

This section illustrates the construction of the concept hierarchy using the *selection* interaction task as an example.

Selection, allowing the user to make a choice from a set of alternatives, is one of the most-used basic interaction tasks, especially in character-oriented applications. Selection choices can allow the user to submit commands, such as save or edit, or allow the user to set system attributes and states (such as turning on boldface type). Selection interaction objects range from buttons of various kinds to text entry and menus. Attributes that are important in determining the appropriate selection mechanism include:

- Action when selected - When a selection mechanism is invoked, it either causes an action to occur (procedural), or it changes a state in the user interface or in the application.
- Number of States - All selection objects have a range of states or choices, from one to infinity. This attribute describes the number of states available for that selection object.
- Choice List Type - whether the number of choices in the choice list is fixed, or whether it can be added to dynamically (variable).

The following table, patterned after the declarative description method described in [2], details selection attributes for each abstract interaction object:

**Table 1: Selection Attributes**

| Abstract Interaction Object | Action When Selected | Number of States | Choice List Type | Grouping |
|---|---|---|---|---|
| Pushbutton | procedural (action) | One | Fixed | Not Grouped |
| Toggle Button | State Change | Two | Fixed | Not Grouped |
| Radio Button | State Change | Two | Fixed | Grouped |
| Basic Menu | procedural (action) | Few (< 15) | Fixed | Not Grouped |
| Option Menu | State Change | Few (< 15) | Fixed | Not Grouped |
| Selection List | State Change | Many - (> 15) | Variable/ Dynamic | Not Grouped |

The attributes of these selection tasks are formalized in Classic as roles, with the following concepts defining their possible values. The concept definition names the possible values of a choice set by incorporating a higher-level concept, SELECTION-PROPERTY. CLASSIC provides operators, such as *one of*, that allow the concept to specify a choice:

```
(cl-define-concept  'SELECTION-ACTION
     '(and SELECTION-PROPERTY
     (one-of
       Procedural-Action
       Visible-State-Change) ))

(cl-define-concept
     'SELECTION-NUMBER-OF-STATES
     '(and integer (min 1)   ))

(cl-define-concept
     'SELECTION-VARIABILITY
     '(and SELECTION-PROPERTY
           (one-of Fixed Variable)  ))

(cl-define-concept
     'SELECTION-GROUPING
     '(and SELECTION-PROPERTY
           (one-of Grouped Not-Grouped)
))
```

The Selection interaction task itself is defined by the composition of the various roles. The *at-least* operator asserts that the named role must be filled in any individual created from that concept:

```
(cl-define-primitive-concept
'INTERACTION-OBJECT 'classic-thing)

(cl-define-concept
     'SELECTION-OBJECT
     '(and INTERACTION-OBJECT
         (at-least 1 action)
          (all action  SELECTION-ACTION)
          (at-least 1 number-of-states)
          (all number-of-states
         SELECTION-NUMBER-OF-STATES)

         (at-least 1 variability)
         (all variability
            SELECTION-VARIABILITY)
         (all grouping
            SELECTION-GROUPING)
      )
)
```

The definition of INTERACTION-OBJECT is at the top of the CLASSIC concept hierarchy; all of the MORPH definitions are subsumed by it, allowing the CLASSIC inference engine to work. Notice that all roles except for SELECTION-GROUPING are required; this is because the only interaction object that refers to grouping is the radio button, which is used to implement a selection from a mutually exclusive set of choices.

The interaction objects are defined and differentiated by restrictions on the roles, as illustrated in the definition of radio button and basic menu:

```
(cl-define-concept 'MORPH-RADIO-BUTTON
      '(and SELECTION-OBJECT
       (fills action
          Visible-State-Change)
       (fills number-of-states 2)
       (fills variability Fixed)
       (fills grouping Grouped)
       )
)

(cl-define-concept 'MORPH-BASIC-MENU
       '(and SELECTION-OBJECT
         (fills action
            Procedural-Action)
         (all number-of-states
               (and integer
                  (min 2) (max 15)))
         (fills variability Fixed)
         (fills grouping Not-Grouped)
       )
)
```

### 3.3 Transformation by Inferencing

The next task is to perform transformations from the MORPH abstractions to specific toolkit instances. For our test, the Java Abstract Windowing Toolkit (AWT) [12] was added to the knowledge base. Following are the CLASSIC descriptions for the “Grouped Checkbox” (corresponds to the MORPH-RADIO-BUTTON) and the “Choice” (corresponds to the MORPH-BASIC-MENU). In order to get a fair test of the inferencing capabilities, no assumptions are made about the hierarchy and the parent concept is INTERACTION-OBJECT, not SELECTION-OBJECT:

```
(cl-define-concept
     'AWT-Grouped-Checkbox
     '(and INTERACTION-OBJECT
            (fills action
             Visible-State-Change)
            (fills number-of-states 2)
            (fills variability Fixed)
            (fills grouping Grouped)
       )
)
(cl-define-concept 'AWT-Choice
     '(and INTERACTION-OBJECT
      (fills action Procedural-Action)
      (all number-of-states
         (and integer (min 2) (max 10)))
      (fills variability Fixed)
      (fills grouping Not-Grouped)
     )
)
```

The inferencing test is performed by providing a MORPH abstract concept to CLASSIC and asking which higher-level concepts subsumes it, which gives us the mapping to the specific Java component AWT-CHOICE. The screen capture below shows the interactive CLASSIC session to perform this query, and the response from the CLASSIC knowledge base on the next line:

```
Classic> (cl-concept-parents
             @morph-basic-menu)
(@c{AWT-CHOICE} @c{SELECTION-OBJECT})
```

CLASSIC returns AWT-CHOICE as the best match for MORPH-BASIC-MENU, and also indicates that the interaction task is SELECTION-OBJECT. Even though the Java AWT components were not directly described as SELECTION-OBJECTS, the CLASSIC system makes the match because of the values of the role fillers. In this way, the MORPH abstract model objects can be mapped to any specific toolkit that is described in the knowledge base.

### 3.4 Design Critics

Another advantage of using a knowledge representation for transforming user interfaces is that design and usability heuristics can be incorporated into the knowledge base. A “design critic”, or guideline, is shown in the range of 2..15 defined in the MORPH-BASIC-MENU definition. A decision was made that any selection with a large number of choices should be handled by another interaction object, a scrolling list. Therefore only selection tasks with choice lists smaller than 10 will ever be mapped to MORPH-BASIC-MENU, and larger lists will be mapped to MORPH-LIST. This enhances the usability of the resulting user interface.

### 3.5 Current Status of MORPH Knowledge Representation

The MORPH knowledge representation for the selection and text entry and output interaction tasks and objects are implemented, along with the associated Java AWT classes. We have experimented with inferencing and with different organizations of the concept hierarchy in order to obtain the best possible mappings. Although the position and quantification interaction tasks are much less ubiquitous in character-oriented code, attribute tables for them have been developed and will be implemented next to complete the knowledge representation.

## 4. Issues and Conclusions

Even though the representation of the user interface model and the identification of the closest match to a specific toolkit widget were successful, we encountered several interesting issues in the study we conducted:

- *Defining the concept hierarchy* - native vocabulary vs. common vocabulary. The first attempt at defining the concept hierarchy was accomplished in a "native vocabulary" fashion - by looking at the toolkits and describing them in their own terms. The result was that the inferencing performed poorly [7] and few matches were made in the transformation step. When the concept hierarchy was designed top-down, with the toolkits and abstractions being described in a common vocabulary, the inferencing worked much better and correct transformations were identified. Therefore it is important to describe all of the user interface components using the same vocabulary, sharing at least the highest level concept and role definitions.
- *Transformation inferencing accuracy.* In order to obtain the correct mappings from the inferencing, the concept definitions had to be carefully crafted to make sure they were accurate. For example, leaving the "grouping" role filler off of a MORPH-RADIO-BUTTON caused matches with AWT-BUTTON and AWT-CHOICE instead of the desired AWT-GROUPED-CHECKBOX. All definitive roles must be present, and determining which roles are necessary is an important part of testing the knowledge base.
- *Individuals vs. concepts at the definition level.* It was tempting initially to define all toolkit objects as CLASSIC individuals, putting them at the lowest level of the hierarchy. However, we ran into problems with this as an overall scheme since some of the interaction objects required ranges. For example, the AWT-CHOICE (which is essentially a menu) can have a range of the number of choices. Defining it as an individual requires fixing the number of choices, which is incorrect. Therefore the toolkit objects must be defined by concepts. In the program understanding process, however, specific implementations in the legacy code should be represented by individuals, since information such as the number of choices is detectable from the code.

In conclusion, this study showed that user interaction tasks and associated interaction objects can be organized into a hierarchy, and used to represent an abstract model in a program understanding process. Interaction object attributes can be identified and used to aid transformation by subsumption and classifaction-based reasoning. The knowledge representation basis for the abstract model adds significant advantages to the process of migrating interactive systems.

## 5. References


[1] [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.

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

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

[4] [FRO85]Frost, Richard. *Introduction to Knowledge Base Systems,* MacMillan Publishing Company, New York, New York, 1985.

[5] [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.

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

[7] [MOO94]Moore, Melody, Rugaber, Spencer, and Seaver, Phil, "Experience Report: Knowledge-Based User Interface Migration", in *Proceedings of the 1994 International Conference on Software Maintenance*, Victoria, B.C., Sept 1994.

[8] [MOO94a]Moore, Melody. "A Technique for Reverse Engineering User Interfaces", *Proceedings of the 1994 Reverse Engineering Forum*, Victoria, B.C., Sept 1994.

[9] [MOO96a]Moore, Melody. *A Survey of Representations for Recovering User Interface Specifications in Reengineering*, Research Report number GIT-CC-96-34, College of Computing, Georgia Institute of Technology, Feb 19, 1996.

[10] [MOO96b]Moore, Melody. "Rule-Based Detection for Reverse Engineering User Interfaces", *Proceedings of the Third Working Conference on Reverse Engineering*, IEEE Computer Society Press, Nov 8-10, Monterey, California, 1996.

[11] [MOO96c]Moore, Melody. “Attributes of Interaction Objects for Knowledge-Based Program Understanding”, Research Report number GIT-CC-96-35, College of Computing, Georgia Institute of Technology, November 24, 1996.

[12] [NAU96]Naughton, Patrick. *The Java Handbook*, Osborne McGraw-Hill, Berkeley, California, 1996.

[13] [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.

[14] [RUG93]Rugaber, Spencer, and Clayton, Richard. “The Representation Problem in Reverse Engineering”, *Proceedings of the Working Conference on Reverse Engineering*, May 21-23 1993, Baltimore, MD. IEEE Computer Society Press, 1993.