MORALE METHODOLOGY GUIDEBOOK
Model Oriented Reengineering Process for Human-Computer Interface (MORPH)

Melody M. Moore
College of Computing
Georgia Institute of Technology
melody@cc.gatech.edu

11. Introduction

MORPH is a technique for reengineering the user interfaces of text-based legacy systems to graphical user interfaces, and a toolset to support the process. This document is an overview of the MORPH reengineering process and a description of the MORPH toolset components and architecture.

11.1 Purpose

Information systems are critical to the operations of most businesses, and many of these systems have been maintained over an extended period of time, sometimes twenty years or more [JAY93]. With the advent of client-server technology, open systems, and high-powered graphical workstations, many businesses are choosing to reengineer their critical applications to better fit the needs of the organization and to take advantage of new technologies. In the information systems domain, this frequently entails reengineering applications that run on large mainframes, with character or form-based user interfaces, to environments that require graphical user interfaces.

It has been estimated that half or more of the code in an interactive system is devoted to implementing the user interface [SUT78]. User interface code tends to be very platform-dependent, often making it necessary to completely rewrite it when the system is migrated. If the user interface components can be extracted from the computational code and expressed in an abstract model, then maintenance and future migrations can be made much simpler, since only the user interface components would need to evolve. Forward engineering tools, such as a User Interface Management System could be employed to generate the new graphical user interface. Understanding the functional requirements of the user interface is an important part of reengineering, and adding automation to this process could substantially reduce the cost and effort for the migration, and allow large-scale systems to be migrated.

11.2 The overall MORALE architecture

(common across guidebooks)

11.3 Overview of method

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. This section is an overview of the MORPH reengineering process. A description of the MORPH toolset components and architecture follows in a later section.

11.3.1 The MORPH Process

There are three steps in the user interface reengineering process [MOO94A]:

- *Detection* - Analyzing source code to identify user interface components from the legacy system.
- *Representation* - Building a model of the existing user interface from the detection step.
- *Transformation* - Manipulating, augmenting, or restructuring the resulting model to allow forward engineering to a graphical environment.

Figure 1 below describes the reengineering process:

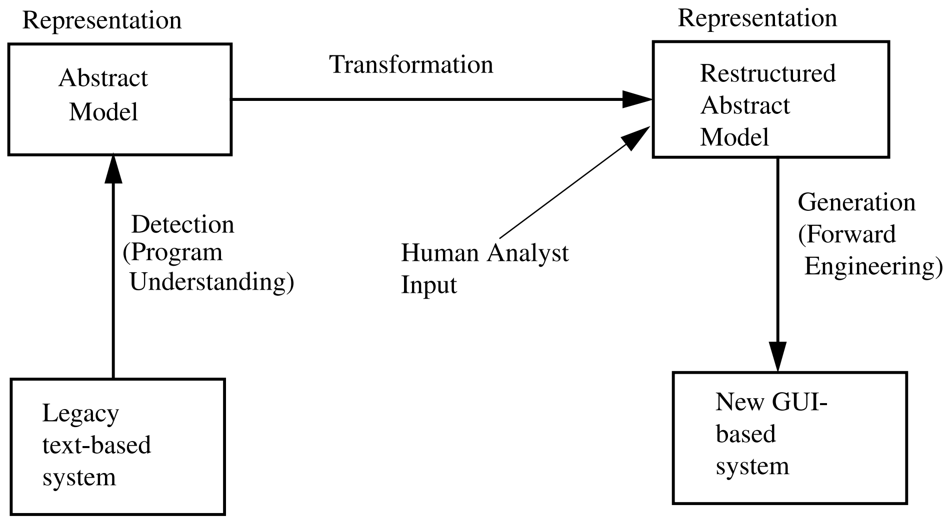

Figure 17 : A Reengineering Process for User Interfaces

The process begins with extracting the user interface from the computational legacy code, using program understanding techniques to 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 UIMS, could be used to automatically generate a new graphical interface for the system.

Defining User Interface Abstractions

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 [FOL90]:

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

Coding patterns that implement each of these basic interaction tasks can be used to detect these tasks in text-based user interfaces. Once the 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 set of interaction objects abstractions, categorized under the basic interaction tasks. These abstractions are described in a knowledge base, to allow transformations between the abstractions and specific GUI toolkits to be accomplished by inferencing.

The Detection Step

The MORPH process starts with a static analysis step which applies a pattern recognizer to the application code in order to locate user interface implementation patterns. First, user interface variables are identified in the legacy code. Abstract syntaxt trees (ASTs) are generated from the legacy code, and coding patterns are then compared to the ASTs at definition and use points for the user interface variables, in order to detect interaction tasks. When an interaction task is detected, the patterns to determine attributes of that interaction task are applied.

The Representation Step

The recognized interaction task and its attributes are stored in the MORPH Knowledge Base as an interaction object, in the correct place in the abstraction hierarchy. Figure 2 below shows the organization of this conceptual hierarchy:

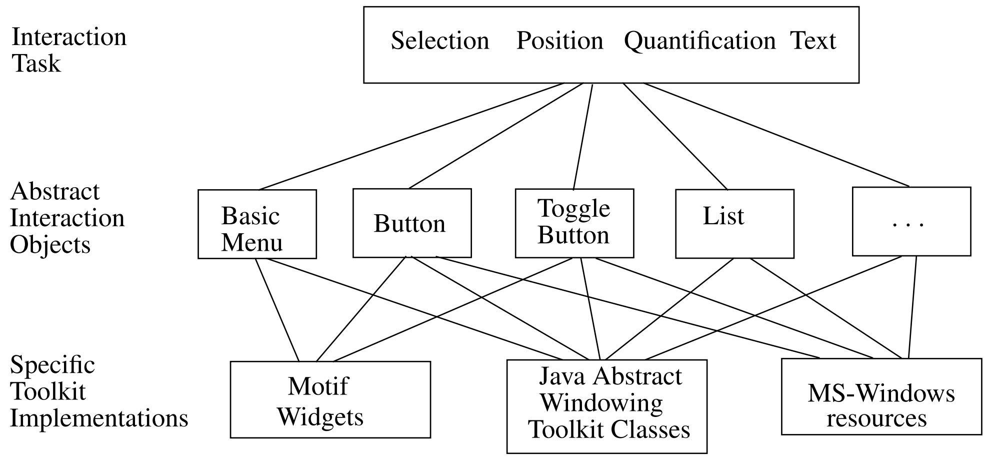

Figure 18 : MORPH Abstraction Hierarchy

The Transformation Step

Once the detection and representation steps are completed, the Knowledge Base can be employed to identify transformations from the abstract user interaction objects to specific toolkit widgets. If toolkit widgets are described in the Knowledge Base, inferencing can be used to find the best mapping from abstract object (by its attributes) to specific widget. When all of the abstract objects have been mapped to specific toolkit widgets, the new graphical user interface can be generated with an (existing) GUI builder tool.

11.4 Example Usage

(scenario)

12. Tutorial Example

13. Detailed Methodology Description

This section details the MORPH knowledge base, pattern set, and toolset.

13.1 Defining the User Interface Abstraction Knowledge Base

Interaction tasks define*what* the user accomplishes while interacting with a user interface - for example, selection, or entering data [FOL90]. *Interaction objects* are the controls (such as buttons, textboxes, or sliders) that define *how* the user interacts with the system [DEB92]. Interaction tasks are defined by the application requirements, 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”.

Migrating an interactive system to a new environment requires three steps: program understanding, representation, and transformation. In the *program understanding* step, identifying interaction tasks from code enables construction of an abstract model of the user interface (expressed by a *representation*). 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 [MOO96b]. This paper expands on these ideas to show that attributes of interaction tasks can also be detected from code, in order to construct a higher quality abstract model of the user interface. With more detail in the model, the transformation step can select more appropriate interaction objects for the implementation of the new user interface.

13.1.1 Declarative Models

In [DEB92], 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 able to determine which components of a specific toolkit are appropriate to substitute in a GUI interface.

The addition of a knowledge base strengthens this approach. Frame-based (slot-and-filler) knowledge base systems are particularly suited for pattern recognition [FRO85]. If data structure and attribute patterns for interaction tasks and objects can be identified, then a frame-based knowledge representation could be used to describe them. This knowledge base could 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.

This section details attributes of interaction tasks that define semantic differences in the associated interaction objects, in order to build a knowledge base of user interface abstractions for character-oriented interfaces. The results have been formalized by coding in the CLASSIC [RES93] knowledge representation language.

13.1.2 Interaction Tasks

The basic interaction tasks outlined in [FOL90] are the foundation for organizing a taxonomy of abstract interface objects [MOO96a]. These high-level abstractions are not particular to any specific toolkit or widget set. The interaction styles described in [HIX93] are classified by the interaction task that they implement, and are included as abstract interaction objects in each task category. The implementations of interaction objects included in a particular toolkit vary widely; however, care has been taken to choose those whose semantics can be implemented in most toolkits. The four basic interaction tasks are:

- *Selection* - allows the user to choose from a set of choices.
- *Text* - provides data entry and output messages to the user.
- *Quantify* - allows the user to specify a numeric value.
- *Position* - allows the user to specify a position in screen or world coordinates.

13.1.3 Attribute Taxonomy

This section details each interaction task, and provides tables showing defining attributes of each interaction object that could be detected from text-based legacy code.

Selection

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 be commands (file save, edit) or setting attributes and states (turn 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 infinite. This column 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 decision table details selection semantics, but not presentation feedback. (In other words, if a pushbutton reverses color when the cursor is moved over it, this is not considered semantics but rather presentation feedback, and not salient in this analysis).

Table 49:

Abstract Interaction Object	Action When Selected	Number of States	Choice List Type
Pushbutton	procedural (action)	One	Fixed
Toggle Button	State Change	Two	Fixed
Basic Menu	procedural (action)	Few (< 15)	Fixed
Option Menu	State Change	Few (< 15)	Fixed
Selection List	State Change	Many - (> 15)	Variable/ Dynamic

The Grouping Concept

In the "toggle button" selection object, there is an emergent phenomenon which occurs when the objects are grouped: they become mutually exclusive. That is, selecting one toggle button deselects all others in the group, as with radio buttons. The behavior of one toggle button affects the behavior of, potentially, all others in the group. The toggle buttons may be grouped organizationally (putting them all in the same window) without these semantics; it is possible to have multiple selection of toggle buttons. However, in these circumstances they are considered to be individual buttons and the grouping is purely presentation information. Grouping toggle buttons to make them mutually exclusive affects their semantics, not just their presentation.

Text Data Entry and Output

Text entry allows the user to input character strings to the application. Text data that is validated by comparing against a list of values is considered a selection mechanism (as described in the "selection list" abstract object). Text data that is not validated, however, such as in entering a name or address that is passed directly to the application, is considered to be pure data. Text data that is validated by checking against a regular expression (checked for pattern correctness, such as in entering a date, which would be checked for a particular date format and legal values) is also considered text data entry. Text output is also included here, since messages from the system can be output read-only and need to be detected in legacy code. There are two attributes that are important in choosing a Text Data Object:

- Modifiability - indicates whether the field can be edited (input), or if it is read-only (output).
- Length - determines the number of characters that can be entered into the textfield or displayed by the textfield.
- Validation allowed - can there be validation of the input, such as error checking, or pattern matching?

Following is the attribute table for text data, which was adapted from [DEB92]:

Table 50:

Modifiability	Length	Validation Allowed	Interaction Object
Editable	Short (< 80 characters)	yes	Single Line Text Field
Editable	Long (> 80 characters)	no	Multiline Text Field
Read Only	Short (< 80 characters)	no	Read-only message

Table 50:

Modifiability	Length	Validation Allowed	Interaction Object
Read Only	Long (> 80 characters)	no	Read-only multiline message

Quantify

The Quantify interaction task allows the user to enter a numeric value in a prescribed range [FOL90]. In a text-based application, this is usually accomplished by the user typing in a number. In rare cases, quantify could be implemented by terminal control semantics by moving the cursor up and down a scale, similar to a numeric slider. Numeric input has the following attributes:

- Modifiability - Indicates whether the numeric field can be edited, or is it display only (read-only).
- Upper and Lower Bounds - Determines whether the minimum and maximum values for the numeric field are known or unknown.
- Range Size - Taking into account the upper and lower bounds, how many values are in the range of the numeric field?
- Precision - Whether an approximate relative value is required (low), or a precise value (high).

The following tables were adapted from [DEB92]:

Table 51:

Modifiability	Upper and Lower Bounds	Range Size	Precision	Interaction Object
Editable	unknown	large	don't care	Textbox (numeric field)
Editable	known	small	low	Slider
Editable	known	small	high	Textbox (numeric field)
Editable	known	large	high	Slider with numeric field
Read-only	unknown	don't care	don't care	Read-only message
Read-only	known	don't care	low	Read-only Gauge
Read-only	known	don't care	high	Read-only message

Similarly, we can define attributes for numeric values with real data types:

Table 52:

Modifiability	Upper and Lower Bounds	Range Size	Precision	Interaction Object
Editable	unknown	any	any	Textbox (numeric field)
Editable	known	small	low	Slider
Editable	known	small	high	Textbox (numeric field)
Editable	known	large	any	Textbox (numeric field)
Read-only	any	any	any	Read-only textfield

Position

The Position interaction task allows the user to specify a coordinate position (x,y) on the screen for interpretation by the application program [FOL90]. In a GUI, this is typically accomplished by moving the mouse to the desired screen location and clicking, or otherwise moving a cursor and using a selection mechanism (such as a pushbutton) to indicate the choice has been made. Another method that can be employed in both GUI and text-based interfaces is typing the screen coordinates in a textfield. Text-based systems can also use terminal control semantics to move the cursor to a particular area of the screen - although care must be taken to ensure that selection is not mistaken for position. Position tasks in text-based systems are always discrete, never continuous, because of the nature of the keyboard input device. Attributes for the Position interaction task are:

- Resolution - how important the actual specified values are to the application.
- Coordinate system - which values need to be specified for the coordinates, either a 2-D coordinate (x,y) or a 3-D coordinate (x,y,z).

Table 53:

Resolution	Coordinate System	Interaction Object(s)
High	x,y,z	Textbox (numeric input)
High	x,y	Textbox (numeric input)
Low	x,y,z	Textbox (numeric input)
Low	x,y	point to drawing window

13.1.4 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.

13.1.5 MORPH Representation

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 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)
       )
)
```

13.1.6 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.

13.1.7 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.

13.2 The MORPH Pattern Set for User Interfaces

(patterns paper contents here)

13.3 The MORPH Toolset

The MORPH tools implement the MORPH process. This section describes the overall system dataflow and each component of the toolset.

13.3.1 Conceptual Dataflow

As shown in figure 2, legacy code is processed first by a recognizer, which identifies the pieces of the legacy code that implement the user interface. When the recognizer detects a user interaction object, it reports this to a modeler, which represents the abstract design of the user interface. Human input from the analyst/designer can modify and refine the user interface model, and when it is completed, it is fed into a transformer. The transformer maps the user interface abstractions into a specific GUI toolkit or widget set, and produces a new user interface. Details of each component are described below.

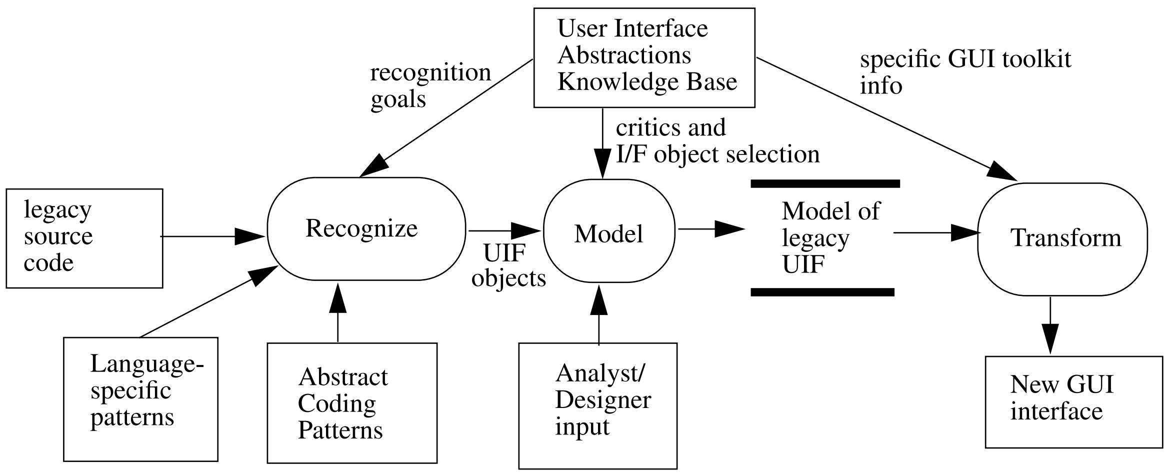

Figure 19 : MORPH Dataflow

13.3.2 System Components

Knowledge Base

The Knowledge Base is a central component of the MORPH toolset. It contains information about user interaction task abstractions, user interface object abstractions, and also specific GUI toolkit information. The Knowledge Base is used in all phases of the MORPH process as well, from detection to modeling to transformation.

- Catalog of User Interface Abstractions - User interface objects are described according to the basic interaction task they implement, plus defining characteristics and attributes.
- User Interface Object selection heuristics - Design critic rules are incorporated into the knowledge base in order to use inferencing to select the best interaction object for a particular task.
- Specific GUI toolkit descriptions - in the transformation step, information about specific GUI toolkit objects and widgets can be used to map from the abstract model to the concrete implementation of the GUI.

Recognizer

- Abstract Pattern Detection Rules - these non-language-specific rules describe patterns that implement basic interaction tasks and attributes of interaction objects in text-based code. The recognizer uses these rules to detect user interface components from the code.
- Language-specific patterns - A set of language-specific constructs are provided to allow MORPH to look for particular patterns in the legacy code (for example, a definition of the I/O routines for a specific language).
- Knowledge base interface for recognition goals - The knowledge base partially drives the detection process by narrowing the list of possible patterns that can be recognized by context. For example, if a selection mechanism has been detected, the knowledge base can infer that further selection mechanisms may indicate a menu hierarchy.

Modeler

- Representation of the generated model - Once the interaction objects and the dialogue control pieces have been detected by the recognizer, the modeler stores and displays the model as it is being built.
- Human judgement input - model editing allows the analyst or designer to make changes to the model as it is being developed.
- Knowledge base critics and object selection - heuristics for choosing an appropriate interface object can be used by the modeler to produce a higher-quality user interface.

Transformer

- Specific target GUI toolkit information is used to transform the abstract model into a concrete implementation in a particular GUI technology. For example, the abstract interaction objects could be mapped to the closest match in Motif widgets by using inferencing if the Motif widgets were also described in the knowledge base.

13.3.3 MORPH Toolset Architecture

The MORPH system is built using Refine and CLASSIC. Following is an architectural description:

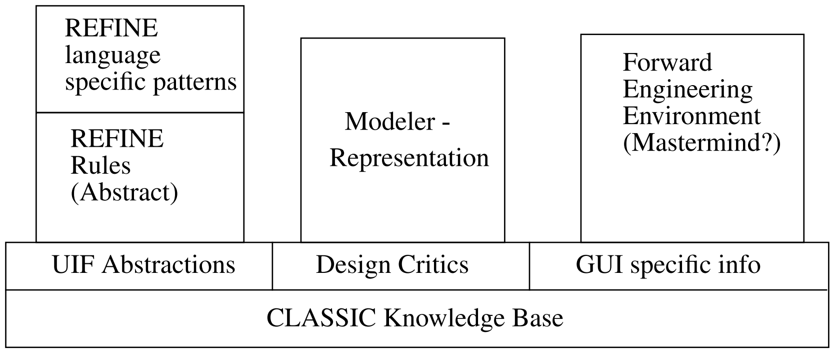

Figure 20 : MORPH Architecture

The cornerstone of the MORPH toolset is a knowledge base implemented in CLASSIC, which is an object-oriented knowledge representation language [RES93]. The Software Refinery [REA??] is used to implement the abstract pattern recognition rules and the language-specific patterns for the detection step. The Modeler is an editor that allows the analyst to view and modify the model that is being built. The Forward Engineering step will be tied into an existing GUI-building toolset such as Mastermind or Amulet.

13.4 Input formats

There are several important inputs in the MORPH technique:

- The Knowledge Base, which contains descriptions of user interface abstractions
- The set of code patterns that implement the user interface abstractions for the detection step

13.5 Output formats

13.6 Steps

13.7 Evaluation

14. Glossary

analyst - an individual who performs program understanding tasks on an existing system.

basic interaction task - (see interaction task) an indivisible interaction task - if the interaction task is decomposed, it is no longer meaningful to an application. [FOL90]

composite interaction task - Aggregate basic interaction tasks - combinations of basic interaction tasks into combined tasks that have a new meaning to an application. [FOL90]

data entry - input from the user to the computer via an input device (keyboard, mouse, etc.) that is not interpreted by the application program, but treated as information to be stored. For example, entering a typed command is not data entry, whereas entering a user's address is data entry.

designer - the software engineer responsible for designing and implementing a new application.

gesture - (as used in [HEN90]) - A sequence of user interactions with an application that can be recognized as belonging to a set of symbols that have meaning to the application. For example, textual handwriting recognition can be accomplished by a gesture recognizer that analyzes the input and matches it to known alphanumeric symbols.

input device - a piece of computer hardware that allows a user to interact with a computer. The concept of logical devices [FOL90] groups input devices into five equivalence classes: locator (position), pick (selection), valuator (numeric), keyboard (strings), and choice (actions).

interaction objects - as defined in [DEB92], are "controls such as buttons, sliders, or menus that can be manipulated directly by the user. Every interaction object in the user interface is associated with an action or attribute in the application data model". As defined in [HIX93], interaction objects are precoded objects in toolkits that can be used in implementing a user interface.

interaction styles defined in [HIX93] are described as "a collection of interface objects and associated techniques from which an interaction designer can choose when designing the user interaction component of an interface. They provide a behavioral view of how the user communicates with the system." In [SHN93], primary interaction styles are menu selection, form fill-in, command language, natural language, and direct manipulation.

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 entered with the interaction techniques. Interaction tasks are defined by what the user accomplishes, not how it is accomplished. The four basic interaction tasks are:
- Selection - allows the user to choose from a set of choices.
- Text Entry - provides unvalidated data entry and output messages to the user.
- Quantify - allows the user to specify a numeric value.
- Position - rarely used in text-based interfaces, allows the user to specify a position in screen or world coordinates.

interaction technique as defined in [FOL90] is a construct that "implements the hardware binding portion of a user-computer interface design". An earlier informal definition adds that interaction techniques are "ways to use input devices to enter information into the computer". Interaction techniques are also defined as primitive building blocks from which a UIF is crafted.

interaction technique toolkits [FOL90] are subroutine libraries of interaction techniques, mechanisms for making a collection of techniques available for use by application programmers (for consistent look and feel).

interactor - (Amulet) In a GUI presentation, interactors are invisible objects that attach to graphical objects in order to allow them to respond to input. [

legacy system - 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.

reengineering - The use of design recovery techniques to understand the code of a legacy system in order to modify an existing system. Typically reengineering is performed to correct errors, restructure a system to improve maintenance, to add functionality, or to port the system to another platform. Also called renovation or reclamation [CHI90].

reverse engineering - The process of analyzing a software program in an effort to create a representation of the program at a higher level of abstraction than source code. Also called design recovery, the process can consist of data extraction, architectural specification, control flow and data flow understanding. [PRE92]

terminal control semantics - in some text-based applications, the cursor can be moved around the screen using keyboard commands or keystrokes. If the application ascribes meaning to the location of the cursor, then the terminal control has semantics to the application. For example, if the screen is split into two areas, and moving the cursor from one area to another determines the meaning of a data value that the user enters, then the application has terminal control semantics.

text-based application (interface) - An application that accepts user input only from a keyboard. Typically a text-based application only displays textual data as well, however this is not a requirement. Alphanumeric characters, as well as function keys and arrow keys can be used to interact with a text-based application.

text entry - Any alphanumeric characters or strings entered by the user. Can be used as data entry (see definition) or as a selection mechanism, as in typing the name of a command.

user - the individual that the software application was designed and implemented for - the "end user" of the system.

user interface elements - [Mastermind term] Pedro's presentation model paper

user interface software - [HIX93] The means for implementing the code that instantiates the user interaction component. This is the "system" or "application" view of a user interface.

user interaction component - [HIX93] The interaction component of an application is its "look and feel", the application's response to what a user sees and hears and does while interacting with the computer. This is the "user view" of a system.

widget - [HIX93] another name for user interaction objects, specifically used to describe X Windows interaction objects, but sometimes used as a general term to describe objects in user interface toolkits. Examples of widgets are scrolling lists, pushbuttons, and cascade menus.

15. Annotated References to the Literature

(How this method relates to other methods, discriminators, links to further information)

Merlo Macroscope

Van Sickle

Commercial methods

16. References

[CHI90] Chikofsky, Elliott, and Cross, James. "Reverse Engineering and Design Recovery: A Taxonomy", *IEEE Software,* January 1990.

[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.

[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.

[FRO86] Frost, Richard. *Knowledge Base Systems,* MacMillan Publishing Company, New York, NY, 1986.

[HEN90] Henry, Tyson R., Hudson, Scott E., and Newell, Gary L. "Integrating Gesture and Snapping into a User Interface Toolkit", *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology (UIST)*, Snowbird, Utah, Oct 3-5, 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.

[MOO96a] Moore, Melody. "A Taxonomy of User Interface Abstractions", PhD Working Paper, College of Computing, Georgia Institute of Technology, November 1996.

[MOO96b] Moore, Melody. "Rules to Detect Interaction Tasks and Presentation Attributes in Character-Oriented Legacy Code", PhD Working Paper, College of Computing, Georgia Institute of Technology, Oct 22, 1996.

[MYE96] Myers, Brad, Ferrency, Alan, McDaniel, Rich, Miller, Robert, Doane, Patrick, Mickish, Andy, and Klimovitski, Alex. *The Amulet V2.0 Reference Manual*, School of Computer Science, Carnegie Mellon University, 1996.

[PRE92] Pressman, Roger S. *Software Engineering - A Practitioner's Approach, Third Edition*, McGraw-Hill, 1992.

[RES93] Resnick, Lori Alperin et al. CLASSIC Description and Reference Manual for the COMMON LISP Implementation Version 2.1, AT&T Bell Laboratoris, 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.