User Interface Design Assistance For Large-Scale Software Development

Gregory Alan Bolcer
University of Southern California
Center for Software Engineering[1]
Los Angeles, CA 90089
E-mail: gbolcer@sunset.usc.edu
+1.714.725-2704

ABSTRACT

The User Interface Design Assistant (UIDA) addresses the specific design problems of style and integration consistency throughout the user interface development process and aids in the automated feedback and evaluation of a system's graphical user interface according to knowledge-based rules and project-specific design examples. The UIDA system is able to quickly identify inconsistent style guide interpretations and UI design decisions resulting from distributed development of multiple UI sub-systems. This case arises when each subsystem conforms to the general style guide rules, but when integrated together, may appear inconsistent.

KEYWORDS: user interface design assistance, style guidelines, project integration, style inconsistency, large-scale software, distributed development.

OVERVIEW

A User interface *style guide* is a collection of heuristical knowledge forming general rules for the creation of an interface for a software application. Because this knowledge is accumulated over the course of tens or hundreds of projects, it may be too general for application to specific design contexts and real design problems. User interface designers often miss many critical concepts and details[12], and completeness and consistency are often overlooked when checking for adherence. This problem stems from the abundance of GUI-specific style guides [1][2][3], the difficulty the UI designer has interpreting these general guidelines [12], and the overwhelming difficulty the UI designer has referencing and applying these guidelines in their daily work[8].

The key objective of the User Interface Design Assistant is to assist project personnel in the automated use of expert knowledge in the design and implementation of toolkit-based graphical user interfaces throughout the lifecycle of large-scale software engineering projects. The UIDA approach is unique because the knowledge-base addresses specific concerns that arise from distributed development. By looking at a system's user interface design beyond one dialog box at a time, the UIDA system is able to use the knowledge-base to synthesize a consistent design when integrating disconnected elements of a system.

The knowledge that is used to enforce style guide adherence and consistency may be supplied from customers, human-factors engineers, and other project developers. Software system customers can provide knowledge about the workplace environment, projected usage, target platforms, display characteristics, or any other requirements that may effect the user interface design. User interface style and project-specific information can also be encoded into the knowledge-base to form project or company-wide guidelines allowing interactive design exploration using the UIDA system while maintaining adherence.

The format of the knowledge is both rule-based for capturing general constraints between graphical objects and example-based for allowing specification of design criteria (possibly by non-technical end-users of the system with the aid of a knowledge engineer). The UIDA knowledge-base does not form a mutually consistent set of guidelines, and in fact, a large portion of the rules are contradictory. The function of this system is to aid in the design of a graphical user interface that adheres to as many style rules as possible while making explicit any rule contradictions.

The UIDA system is integrated with Sun Microsystem's **DevGuide**[2] through the use of its Guide Interface Language (GIL), and allows a common reference point for evaluating and customizing user interface requirements throughout the course of a project or even across multiple projects or versions. This is accomplished by explic-

[1] This work was conducted jointly at the University of Southern California's Center for Software Engineering, and the Arcadia Project, University of California Irvine.

[2] A commercial direct manipulation user interface design and specification layout tool for building Open Look(tm) applications

itly stating project style guidelines in an OPS5-like[3][5] rule format that allows the *conditions* of the rule to identify inconsistent layout, sizing, color, and presentation, and the *actions* of the rule to generate alternative UI designs that are consistent with the evolving design specification or project specific design cases.

The automated use of the knowledge allows feedback to the developers at the various stages of the project development.

Requirements: The knowledge-base can be provided by the customer to constrain the GUI design space available to the contractor ensuring that conventions and appropriate usage models are consistent with what is currently being used within the customer's company at large.

Rapid Prototyping: System prototypes can be rapidly created and evaluated while conforming to overall UI requirements. This helps capture all of the benefits of rapid prototyping with less of a chance of incurring a "throw away" cost.

Design Review and Integration: Inconsistencies created by integration of project sub-systems can be easily identified, reconciled, and recorded.

Acceptance Testing: There is more solid communication between the customer and the contractor through the sharing of the knowledge-base resulting in less chance of the customer asking for a UI revision.

The principal means for addressing these issues is accomplished in the UIDA system through a combination of *analytical* critiquing[6], where the user interface is evaluated with respect to possible flaws, *differential* critiquing[8], where the system generates its own solution and compares it to the interface created by the designer by pointing out differences, and *advisory* critiquing, where the designer is notified of a change that couldn't be accomplished because of conflicts or constraints even though the change would improve the evaluation of the interface according to the style guide criteria. It is important to note that researchers have argued (Löwgren and Nordqvist[8]) that critiquing in the domain of user interface design is not eligible for *differential* treatment because, in general, there are many examples of alternative solutions with equal validity. The UIDA system addresses this issue by generating design alternatives that are limited to what can be found in other sub-system UI designs within the project or in specific graphical examples of desired UI designs. If no solution can be generated by the UIDA, an advisory notification is given to the designer.

[3] A common rule-based syntax for expressing knowledge.

UIDA APPROACH

The UIDA system uses knowledge and rules to satisfy design principles within a set of objects or consistency checking between several sets of objects. The UIDA knowledge-base contains representations for user decisions, rule application history, user interface objects and their attributes, and grouping information for grouping objects into a project or rules into a family. The rule-base utilizes this knowledge to initiate actions and make intelligent decisions about a user interface. Rules fall into three categories: interaction, identification, and resolution. Interaction rules serve to provide help, browsing, querying, and warning functions to the user; identification rules embody the style guide principles including layout critiquing and consistency checking; resolution rules handle user interface object manipulation and attribute assignment.

The level of abstraction that the UIDA reasons about consistency and layout is at the frame and panel level which correspond to a graphical window and the area inside the window. The UIDA takes an inductive approach to managing consistency (see figure 1). Each frame is first critiqued according to the knowledge-base as a standalone application. Once all the frames have been filtered through the style guide knowledge-base, consistency across frames can take place. The UIDA makes the assumption that objects within a frame or panel are semantically related to some user task and therefore are grouped together. Similarities between windows are measured by the intersection of the labels, the types of the objects, and the alignment and placement policies. When an intersecting set is found between two or more frames, inclusion, exclusion, and inconsistency information is generated about the UI. The resoultion rules are then applied to a single frame to determine what changes to make, if any, and only if the user approves them. The frame is then annotated with a set of inconsistencies, a set of rules applied to the frame, and a set of user decisions in response to the rules. After doing this on a frame-by-frame basis, the system will have a series of decisions recorded as explicit design decisions. As integration of the project proceeds, new frames can be added. When conflicting design decisions are identified, they are brought to the integrator's attention.

The UIDA knowledge-base also contains a collection of *meta-rules* for recommending rule applications to the user, providing conflict identification support, and customizing various levels of automated conflict resolution. Meta-rules also encode the knowledge that the successful application of one UI style guide rule may contradict the successful application of another.

Base Case (frames $1..N$):
Style Guide *Base Case*:
Recommend by context rules to apply: $Rules \subseteq AllRules$
$\forall\ \mathcal{F} \in Frames$ do:
$\forall\ \mathcal{R} \in Rules$ do:
$\mathcal{A}$: apply $\mathcal{R}$ to $\mathcal{F}$
Record $\mathcal{A}(\mathcal{R}, \mathcal{F})$ as
Approved: annotate $\mathcal{F}$ with ($\mathcal{R}$,+)
Rejected: annotate $\mathcal{F}$ with ($\mathcal{R}$,-)
Approved w/ Inconsistency:
annotate $\mathcal{F}$ with ($\mathcal{R}$,+)
annotate $\mathcal{F}$ with *Object Inconsistencies*

Inductive Case (frame $N+1$):
Inconsistent Objects:
If F_1 satisfies *Base Case*, $O_1 \subseteq F_1$,
$\exists\ F_2$ satisfies Base Case, $O_2 \subseteq F_2$,
Then $\forall (o_1 \in O_1, o_2 \in O_2\ st.\ o_1 \sim o_2)$ do:
Notify User of proposed changes to O_1:
Rejected: annotate Project with *Frame Inconsistencies*
Approved: Change O_1, Recheck F_1 satisfies *Base Case*

Inconsistent Styles:
If $\exists F_1, F_2$ st. satisfy *Base Case*,
F_1, F_2 annotated with rules R_1, R_2,
R_1, R_2 are incompatible
Notify User of conflict and propose reapplication of R_1:
Rejected: annotate Project with *Frame Inconsistencies*
Approved: Re-apply $\mathcal{A}$:(R_1, F_1) or $\mathcal{A}$:(R_2, F_2),
Recheck Inconsistent Styles

Figure 1: Inductive Integration

UIDA IMPLEMENTATION

Knowledge-Base

Style and layout knowledge-base rules are separated into families. A family of rules corresponds to a directory of files, each file containing 1-4 related rules. Because the rules are located in files, the knowledge-base is easily browsed using a standard graphical desktop file manager. In the case of the UIDA system, the Open Windows' file manager is provided for ease of use.

In addition to the style and layout rules, the UIDA provides consistency and integration rules that can be enabled when integrating sub-systems. Some rules provide a means for recognizing project inconsistencies even though the individual interfaces comply to the style guidelines or rules. A detailed example of this can be seen in figure 2. The three frames are taken from a system [14] where each component was built by a different designer for a different part of the overall system. For each frame, the designer has chosen to ignore one or more style guide principles. For example, all three frames don't include either a "Quit" or "Done" button. These decisions are then stored with each of the frames to be used later for inconsistency and conflict detection. Once these frames are integrated into the project, the designer can then check for system-wide inconsistencies. For the frames in the example, the following integration conflicts are brought to the designer's attention that could not be detected before integration (frames are referred to left-to-right by $Frame_1, Frame_2, and Frame_3$:

- The **Field Class** object in $Frame_1$ is implemented as a different type than $Frame_2 and Frame_3$.
- The layout algorithm for the text fields between the frames is left-aligned in $Frame_1$, and colon-aligned in the others.
- The frame sizes don't correspond well to the number of objects present within them. $Frame_2$ contains five objects but has a smaller area than $Frame_3$ which contains only four objects with the similar labels.
- The length of the text fields is different across all three frames.
- The ordering of the fields is switched between frames.
- $Frame_2$ and $Frame_3$ are missing a **Default Value** field, or else $Frame_1$ has included extraneous objects. Several warnings are also given concerning the inclusion and exclusion of the buttons **Apply**, **Insert in List**, **Reset**, and **Hide** across the frames.

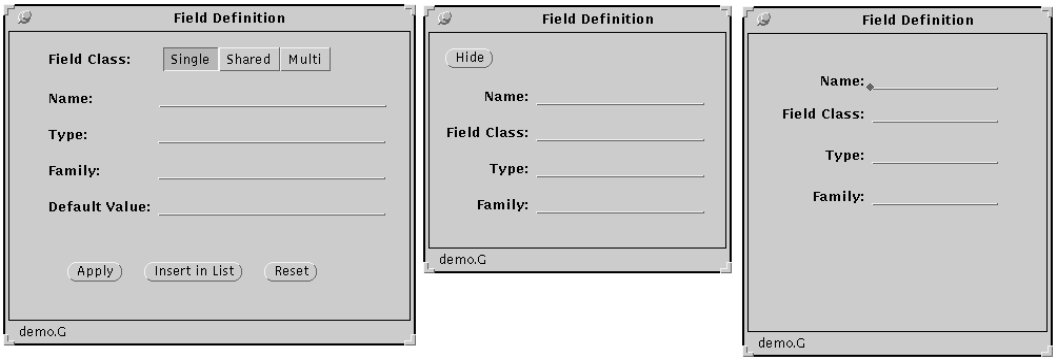

Figure 2: Example Inconsistencies

Architecture

The UIDA system is part of a larger collection of design and runtime user interface development tools[11], although its bottom-up approach and file-based integration allow it to be used as a stand alone system. The UIDA system adopted DevGuide's notion of graphical interface language files (GIL) and project files as means of specifying UI descriptions. Each project has a project description file ("projectname.P"), a list of interface description files ("filename.G") and an associated rule-action list for storing design choices and other information ("projectname.A"). These files are read-in and written-out as the result of meta-rules triggered by the inference engine whenever a current working file is selected. While GIL allows a straightforward integration with the inference engine and knowledge-base, it is not a general object description language. This causes some GIL-specific dependencies in the rule syntax as seen in figure 3.

Once UI designers save their interface files in the GIL format, these files can be read into the lisp environment and treated by the UIDA as a list of graphical objects. Changes are made to object attribute values in the working memory of the inference engine. To view the changes, the objects are written out to a file in the GIL format and then re-read into DevGuide to allow for side-by-side comparison. The inference engine is designed to execute OPS5-like rules and contains approximately 1200 lines of lisp code. It is interesting to note that because GIL is a relatively high level design language, various programming languages (other than lisp) and runtime architectures can be generated from the user interface designs that are evaluated and changed using the UIDA system. This is accomplished through DevGuide's separation of implementation and design as well as the loose file-based integration between DevGuide and the inference engine.

Families of rules correspond to specific directories containing several related files. Each file may contain several rules that share the same style or consistency guideline expressed as a combination of conditional pattern matching and procedural knowledge represented by a lisp function. A single rule file contains all the rule-based and procedural knowledge that is needed to find and correct the single specific style and integration inconsistency. The one exception to this is the shared meta-rules which are loaded in with all rule sets and provide mechanisms for reading in objects, identifying conflicts, handling the level of automation support, and handling of messaging information (i.e. which family or style guide the rule belongs to).

User Interaction

Because the integration between the knowledge-base, DevGuide, and the inferencing engine is file-based, users need to explicitly load in rules and interface files. Project and interface description (GIL) files are loaded through a call to 'set-working-file'. The designer determines the level of change the system is allowed to make to the interface after a conflict is found by setting the automation level to LOW for generating warnings, MED for asking the user to confirm the change, and HIGH for always making the change if a solution is possible. To aid the designer in remembering and choosing rules to apply, the UIDA system provides meta-rules. These rules (at the prompting of the user through a 'recommend' function) provide a list of families of style guide rules. The user can then easily browse and choose rules to apply from that family.

Figure 4 shows an excerpt of the output from the application of the "inconsistent-ordering" rules to an example project. The level of automation in this case defaults to HIGH. The working file is set to "order-project.P", a project file containing two completed interface files "order-demo.G" and "order-demo2.G" as seen in fig-

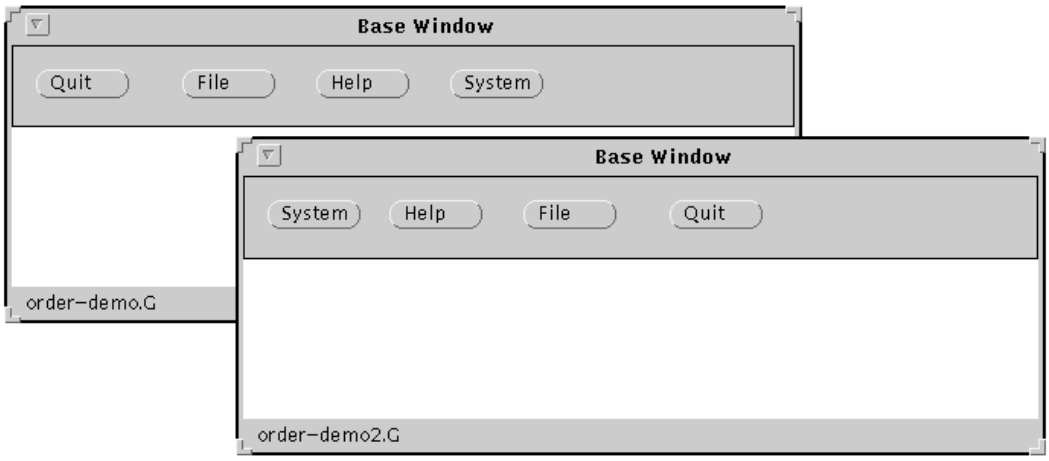

Figure 5: Before UIDA Changes, Inconsistent Ordering

ure 5. Because some of the graphical objects have labels that are the same, the recommended rule list includes the order family. Loading and running the order-switch rules from the order family then results in the following interaction. Once the changes are made, the user prints the file out, reads it into DevGuide and compares the before and after views of the interface side-by-side as seen in figures 5 and 6. If the change is satisfactory, then the designer initiates a *Save-Change* saving the changes to the working-file.

The method of rule application is incremental, allowing the designer to iteratively refine the design of the user interface through a series of small, consistent changes. This approach prevents the user from getting a design back that is "unrecognizable" by allowing each change to be vetoed. Also, conflicts are immediately identified, and various courses of action represent explicit design decisions that can be recorded and compared with other sub-components of the project. Referring again to figures 5 and 6, in addition to the consistent ordering rules, the designer can apply another rule to the project that causes the "Help" button to always be the rightmost button. Note that when these two rules are applied incrementally, the UIDA system brings to the attention of the designer that a conflict may arise if the second rule is allowed to continue with the suggested changes. This is because the rule that re-orders the buttons may not preserve the previous application of the help button positioning. To overcome this conflict, the designer can apply multiple rule sets by invoking *Add-Ruleset*. In this way, two rules that may conflict with each other can be considered simultaneously. If later in the development lifecycle of the project a new button is required as a result of a design change, previously matched rules can be easily re-checked using the UIDA system, and if a conflict is found, it is brought immediately to the attention of the designer.

EXPERIMENTAL RESULTS

Because DevGuide is a commercial tool, the UIDA system has the ability to critique "real-world" user interfaces. To evaluate the effectiveness of the UIDA knowledge-base, several medium-size software projects ([7],[10]) developed using DevGuide were critiqued using the Design Assistant (only one of which will be discussed in detail). In addition, to evaluate the claim that knowledge-based approaches that provide even partial solutions can have a large impact on quality and productivity[4], the UIDA system was empirically evaluated on issues of usability and ease of knowledge-application. Described in the balance of this section are the results of these experiments.

KBRA (*Knowledge Based Review Assistant*[7]) is a tool designed to assist a software project *review coordinator* in preparing a formal code review. This system encompasses about 350 graphical toolkit objects spread across 15 different windows built by two developers. The implementation of the design is compiled into 14-KLOC (thousand-Source-Lines-Of-Code) approximately 85% of which is generated user interface code. Of the 72 total rules, 15 rules were triggered one or more times. Of these 15 rules, 9 could be construed as legitimate design choices while the other 6 were side effects of the developer's choice to use graphical objects for their "look" rather than their functionality. For instance, the developers decided to use arrows in one of the panels that were made up of empty text-fields and message fields. This flagged several alignment, layout, and labeling rule violations. Overall, 14% of the graphical objects were either manually or automatically

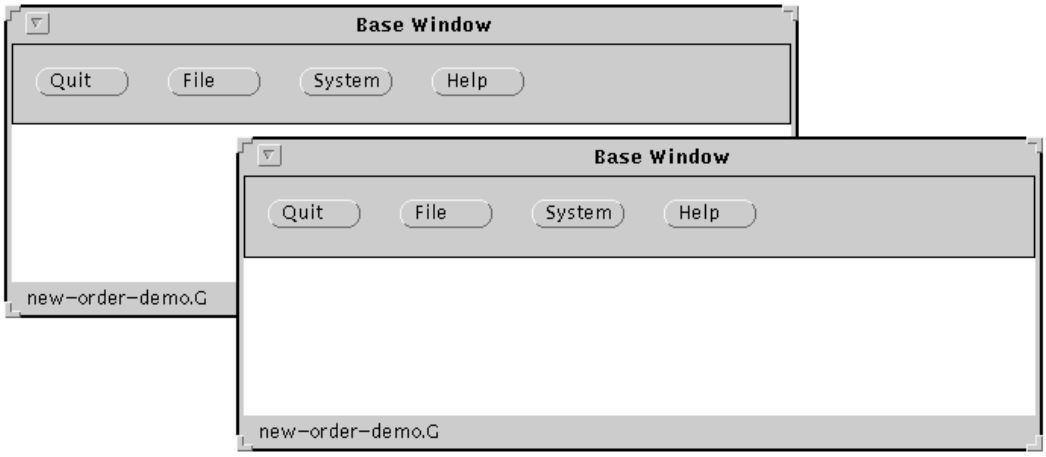

Figure 6: After UIDA Changes, Consistent Ordering

changed to conform to style and integration rules.

The second part of the experiment was to allow the use of the UIDA by several design groups to evaluate the robustness of the knowledge-base and the ease which the developer could apply it. The subjects for the experiment were undergraduate computer science students working on the design of a Traffic Control and Management System for a software engineering class. Two project teams chose to participate (4 members per group). Each team member was responsible for designing some portion of the user interface using DevGuide. Once all the members saved their UI designs into the project file, one member from each team sat down under supervision to use the UIDA system.

As mentioned before, because the UIDA system is loosely integrated, some of the methods of interaction are cumbersome. Uniformly, both students had some degree of the difficulty in using the system which could easily be remedied with the introduction of a graphical user interface for loading and running rules. Also, when a rule was applied, the recap of the style guidelines from which the rule originated often didn't provide enough information for the students. They often requested an explanation as to how the rule went about matching to their specific interface. It is important to note that although both subjects had a fair amount of programming expertise, neither had any previous experience with "principles of good user interface design". The issue of whether one needs to be a human factors expert in order to effectively apply human factors knowledge correctly should be addressed in a more controlled experiment (along the lines of [9][13]). As in the previous experiment, several rules could also be ignored because of design decisions on the part of the students. For instance, DevGuide allows the association of textual help for each panel area, thus the rule checking for the existence of help flagged several violations. Also, because of the small number of rules that actually triggered, the UIDA system wasn't able to identify any conflicts in the applications of these rules to the interface descriptions.

On the encouraging side, both students found the rule browsing and *meta-rules* useful for choosing rules for application. Both students found the rules for alignment and layout corrections useful for portions of the user interface where not enough attentiveness to detail was applied to the design. In addition, between the projects, two integration inconsistencies where found. One project had two buttons located on different windows (designed by different people) that shared the same label even though they represented different functionality. On the other project, the location and ordering of several "Apply" and "Done" buttons differed across windows.

SUMMARY AND CONCLUSIONS

The most important difference between current systems and the UIDA system is the emphasis on addressing problems that arise when scaling the technique of knowledge-based design assistance to large-scale and distributed development of user interfaces. Because *a priori* style and consistency rules are difficult to specify in this setting, the focus of the design assistance should be on conflict resolution and consistency. In addition, the incrementality of the knowledge-base allows the history of decisions made by the designer (through rule applications) to be explicitly recorded allowing for easy identification of user interface design decision conflicts. Meta-rules for recommending style guide rule applica-

```
(make-production
  :name 'x-reorder-objects
  :conditions '((order) (automation HIGH)
    ;; GIL Dependent
    (:type ?type  :name ?name
                  :owner ?owner
                  :help ?help
                  :x ?x
                  :y ?y
                  !other-values)
    (:type ?type2 :name ?name2
                  :owner ?owner
                  :help ?help2
                  :x ?x2
                  :y ?y
                  !other-values2)
    (:type ?type3 :name ?name3
                  :owner ?owner3
                  :help ?help3
                  :x ?x3
                  :y ?y3
                  !other-values3)
    (:type ?type4 :name ?name4
                  :owner ?owner3
                  :help ?help4
                  :x ?x4
                  :y ?y3
                  !other-values4)
    ;; end GIL Dependent
    (*is-pitem ?type)
    (*is-pitem ?type2)
    (*is-pitem ?type3)
    (*is-pitem ?type4)
    (*unequal ?name ?name2)
    (*unequal ?name3 ?name4)
    (*unequal ?owner ?owner3)
    (*< ?x ?x2)
    (*< ?x4 ?x3)
    (*label-equal ?other-values ?other-values3)
    (*label-equal ?other-values2 ?other-values4))
  :actions '(
    ($write-line* "Switching: ")
    ($write-line* "   " ?name " and " ?name2)
    ($write-line* "    to match order of ")
    ($write-line* "   " ?name3 " and "?name4)
    ($switch-x ?name ?name2))
  :number '3)
```

Figure 3: Example Rule Syntax

```
1.   begin

2.   get-all-objects
     Found Project file: order-project.P

3.   get-objects
     Reading Objects from order-demo2.G

4.   get-objects
     Reading Objects from order-demo.G

5.   x-reorder-objects
     Switching:
       button7 and  button6
       to match order of
       button3 and  button2

     • • •

     • • •

     • • •


10.  x-reorder-objects
     Switching:
       button7 and button5
       to match order of
       button3 and button1

11.  halted.
```

Figure 4: Example Output Messages

tions are helpful for non-experts. Also, by designing the knowledge-base with concern for integration, the problem of generating a solution through *differential* critiquing becomes more manageable by conforming to an explicitly provided design example or an evolving design specification in another UI sub-component. The UIDA system represents a methodology for addressing large-scale user interface issues for applying design knowledge, and the author encourages other researchers to expand on the approaches identified in this paper that address these areas of concern.

ACKNOWLEDGEMENTS

The author would like to thank Barry Boehm, Prasanta Bose, and Richard Taylor for their comments and guidance on both this paper and project, Jonas Löwgren for sharing the KRI/AG knowledge-base, Debra Richardson for use of her undergraduates as test subjects, and all those who donated design code for evaluation.

References

[1] *OSF/Motif Style Guide*. Cambridge, MA, 1.1 edition, 1988. Open Software Foundation.

[2] *OPEN LOOK Graphical User Interface Application Style Guidelines*. Sun Microsystems, Inc.; Addison-Wesley, 1990 edition, 1989.

[3] Defense information systems agency human computer interface style guide. Style guide, Center for Information Management, Feb. 1992. Version 1.0, 200 pages.

[4] B. Boehm. Notes on a knowledge based software architecture assistant. Notes, U.S.C. Center for Software Engineering, Jan. 1992. 18 pages.

[5] L. Brownstone and et al. *Programming Expert Systems in OPS5*, volume 1 of *Artificial Intelligence*. Addison-Wesley, Jan. 1986.

[6] G. Fischer, A. C. Lemke, and T. Mastaglio. Using critics to empower users. In *Proceedings of CHI '90*, pages 337–347, Seattle, Washington, Apr. 1990.

[7] J. Liao and J. Hsieh. Knowledge-based review assistant. Project, U.S.C. Center for Software Engineering, 1993. 15 pages.

[8] J. Lowgren and T. Nordqvist. Knowledge-based evaluation as design support for graphical user interfaces. In *Proceedings of CHI '92*, pages 181–188, Monterey, California, May 1992.

[9] J. Nielsen. Finding usability problems through heuristic evaluation. In *Proceedings of CHI '92*, pages 373–380, Monterey, California, May 1992.

[10] D. Richardson and et al. Developing and integrating prodag in the arcadia environment. In *ACM SIGSOFT*, pages 109–119, Tyson's Corner, Virginia, USA, Dec. 1992.

[11] R. N. Taylor and G. F. Johnson. Separations of concerns in the chiron-1 user interface development and management system. In *Proceedings of INTERCHI '93*, pages 367–374, Amsterdam, The Netherlands, Apr. 1993.

[12] L. Tetzlaff and D. R. Schwartz. The use of guidelines in interface design. In *Proceedings of CHI '91*, pages 329–333, New Orleans, Louisiana, Apr. 1991.

[13] H. Thovtrup and J. Nielsen. Assessing the usability of a user interface standard. In *Proceedings of CHI '91*, pages 335–341, New Orleans, Louisiana, Apr. 1991.

[14] P. S. Young and R. N. Taylor. Human-executed operations in the teamwaree process programming system. In *Proceedings of the Ninth International Software Process Workshop*, Jan. 1992.