

Template-Based Mapping of Application Data to Interactive Displays

Pedro Szekely

USC/Information Sciences Institute
4676 Admiralty Way, Marina del Rey, CA 90292
(213) 822-1511
szekely@isi.edu

Abstract

This paper describes a template-based method for constructing interactive displays with the building-blocks (widgets) provided in a user interface toolkit. Templates specify how to break down complex application objects into smaller pieces, specify the graphical components (widgets) to be used for displaying each piece, and specify their layout. Complex interfaces are constructed by recursively applying templates, thus constructing a tree of widgets to display a complex application object. The template-based method is more general than the interactive, WYSIWYG interface builders in that it can specify dynamic displays for application data that changes at run time. The template-based method also leads to more consistent, extendable and modifiable interfaces.

1.0 Introduction

User interface toolkits such as the X Toolkit [10] and the Macintosh Tool-Box [5] provide abstractions that make the construction of user interfaces significantly easier than programming using graphics primitives. Unfortunately the toolkits do not make the construction of user interfaces easy enough. The tasks of assembling the widgets to construct complex displays, and of tying the widgets to application data structures remains difficult and time consuming.

Interactive user interface builder systems such as Prototyper [14] provide interactive what-you-see-is-what-you-get interfaces to assemble tool-box widgets into more complex

interfaces. These tools are excellent for a restricted class of interfaces, which typically includes only menus and dialogue boxes. However, these tools do not help with the construction of the "main windows" of an application, which display application objects that typically change at run time.

This paper describes a template-based method for assembling widgets into complex interfaces and tying them to application objects. Templates specify how to break down complex application objects into smaller pieces, specify the widgets to be used for displaying each piece, and specify their layout. Complex interfaces are constructed by recursively applying templates, thus constructing a tree of widgets to display a composite application object. The template-based method supports the construction of dynamic displays, and also encourages the design of consistent, extendable and modifiable interfaces.

The paper is organized as follows. Section 2.0 presents an overview of a template-based UIMS named Humanoid, the High-level UIMS for Manufacturing Applications Needing Organized Iterative Development. Section 3.0 compares the template-based method with other methods, sections 4.0 and 5.0 describe the template-based method in detail, and finally section 6.0 discusses our experience with Humanoid.

2.0 Overview of Humanoid

At a high level, Humanoid can be viewed as a layer of software on top of traditional user interface toolkits. Humanoid provides abstractions to facilitate the specification of the connections between application data structures and widgets provided by toolkits. Humanoid can also be viewed as a user interface design environment where interface designers can construct interfaces by establishing a mapping between the data structures of an application and the widgets of a user interface toolkit.

FIGURE 1. shows the role Humanoid plays in the specification of a user interface. The information above the grey line is information specified at application construction time. The application types and commands specify the content or functionality of an application. They specify the kinds of objects that the application supports, and the commands to manipulate those objects. For instance, in an agenda management application the objects include *Agenda-Items*, which specify tasks that the user wants to manage, and the operations include *Open*, to view the contents of a task, and *Mark-As-Done*, to mark a task as done so that it is taken off the agenda. Humanoid assumes that applications are written using a frame-based knowledge representation system [8]. The application objects are represented as data structures with slots whose values can be either other objects, or primitive values such as numbers and strings. Hence, the run time state of an application is represented as a network of objects.

The widget set contains the set of user interface building blocks available to the interface designer from a widget library, plus the ones the designer defines in the course of building the interface. Widgets include panels, which group together other widgets, as well as check-boxes, radio-buttons, type-in buffers and buttons. At run time, the state of the interface is represented by a tree of widget instances, each of which defines a small portion of a complex display. For instance, a dialogue box is constructed as a tree of widget instances. The top node of the tree is a widget instance that groups together children widget instances that represent the check-boxes, radio buttons, text labels and other items that appear in the dialogue box. Each check box is itself a tree of widget instances, one widget instance for the label and another one for the selection box. Widget instances have slots that determine their visual characteristics (e.g. font, labels, line-style, visibility), slots that point to other widget instances, and slots that point to application objects, thus establishing a correspondence between application objects and their representation on the display.

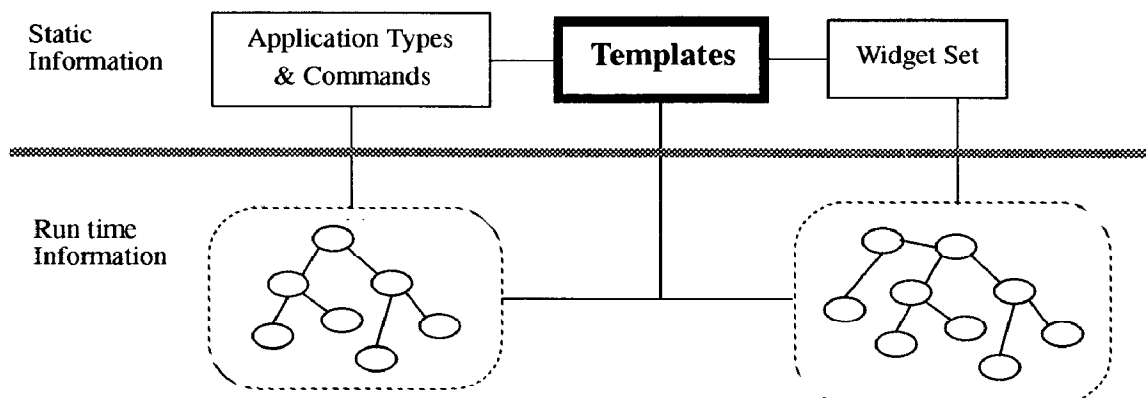
The templates are specifications of how the application objects created at run time must be connected to instances of widgets, also created at run time. The templates specify for each application type the widgets available to display objects of that type, and for each available widget, the conditions under which it is appropriate to use it, and instructions on how to fill in the slots of widget instances.

There are several factors that complicate the definition of the mapping between application objects and widget instances:

- The application and widget networks do not have corresponding nodes. Many nodes in the application graph are for internal uses and should not be presented to the user. Also, many nodes in the widget tree represent labels and background graphics with no counterpart in the application graph.
- The application and widget graphs do not have the same structure. The structure of the application graph is usually determined to make programming convenient. The widget tree structure typically depends on the layout and the subset of the information to be presented to the user in a particular view.
- The difference in structure between the two graphs requires increased flexibility in the mapping specification language, because it must be able to define mappings between potentially very different structures.
- The contents of the application tree are determined at run time. Hence, the assignment of widgets to application objects has to be conditional on the features of the application graph that change at run time.

Templates address these issues by providing flexible mechanisms to select the widget kind to be used for each node in the widget tree, and by providing flexible ways for specifying the correspondence between nodes of the application and widget trees.

FIGURE 1. Humanoid provides an abstraction called Templates to specify the mapping between application objects and widget instances.



3.0 Related Work

There are several methods for specifying the mapping between application objects and widgets. This section reviews the more important ones and compares them with Humanoid's template-based approach.

The most primitive method for specifying the mapping between application objects and widget instances is to write a program in a conventional programming language such as C or Lisp. The program calls toolkit procedures to create the appropriate widget instances and passes in as parameters the values to fill in the widget slots. This is the method that many applications of the X toolkit [10] and the Macintosh toolbox [5] use. Even though the use of the toolkit saves a tremendous amount of effort, the job is sufficiently complex that the remaining tasks are still time-consuming and cumbersome. Since programming is involved, human factors experts are isolated from the iterative development cycle needed to construct good interfaces. The main advantage of the programming approach is its great flexibility in constructing any interface feature desired.

Interactive interface builder systems such as Prototyper [14], Cardelli's interface builder [4] and Action [1] are tools whose goal is to eliminate programming from the interface construction process. Interface designers draw the widgets on the screen, interactively specifying their layout and other display properties such as fonts, line styles and labels. Some interface builders, such as Prototyper also allow the designer to interactively specify the application procedures to be called when the widget is activated. Interface builders have many advantages over the programming approach. Interface builders are easy to use, the interfaces built with them are easy to change, and new displays can be built very quickly, often in a matter of minutes.

The main shortcoming of interface builders is their inability to specify interfaces for objects that change at run time. Interface builders are excellent for constructing dialogue boxes containing menus, buttons, sliders and other such widgets, provided that all the widget instances to be displayed are known at design time. For instance, the number of radio buttons in a group, and their labels, must be known at design time so that the designer can draw them on the screen as they are going to look to the user. However, if the set of choices is determined at run time, it is impossible at design time to specify the labels and the position of the buttons in a what-you-see-is-what-you-get fashion. In this case it is necessary to specify a *method* for computing the labels and their layout. The inability of interface builders to specify *methods* of presentation rather than the presentation itself makes interface builders of little use for building the "main windows" of an application, which typically display objects that change at run time.

Systems like APT [9] and II [2] are representative of systems that automatically generate displays. APT produces displays of quantitative data, and can produce results of quality comparable or better than the results produced by humans. However, few automatic systems have been built to construct fully interactive displays of data that can change at run time.

Systems like MIKE [13] and Chisel [15] are representative of systems that generate interactive interfaces automatically from a description of an application's objects and commands. The main shortcoming of these systems is that the interfaces generated are not of high enough quality. Thus these systems are used to generate a prototype of the interface, which is then refined by editing the generated specification of the interface. This is like editing the object code produced by a compiler, which creates the maintenance problem of carrying over the changes when the sources themselves change and the original interface must be re-generated. The shortcomings of these systems arise because the algorithms they use to design the interface are built-in into the code of the tool, and cannot be changed by designers.

Few automatic or semi-automatic systems have been built that make aspects of their interface design algorithm explicit so that designers can change them to produce the interfaces they desire. The II system encodes the design algorithm as rules so that designers can add new rules to change the behavior of the system. The shortcoming of II is that it only addresses the output part of interface generation, its rule language is complicated and hard to use, and the displays produced are not updated when the presented data changes.

The ITS system [3] is another system that uses rules. In ITS, interactive programs are defined using three kinds of abstractions: data definitions, to define the data manipulated by the program, content trees, to define the dialogues in an abstract way, and style rules, to map the content trees to graphical objects. ITS constructs interfaces by first refining the content tree until it represents all the data to be presented to the user, and then applying the style rules to the refined content tree to obtain the graphical representation of the data. The main difference between ITS and Humanoid is that Humanoid uses the templates both to decide what information to present (content tree refinement) and to format the data (style rule application). Thus, Humanoid uses rules to design both the content and formatting of the interface.

Humanoid is a semi-automatic system. Humanoid's template-based method solves the main shortcomings of interface builders and automatic interface generators. Humanoid can specify interfaces for data that changes at run time, and makes the interface design algorithms explicit as templates that designers can change.

4.0 Templates

Templates specify how to map application data structures to widgets, how to break down complex data structures into sub-structures, and how to assign a widget to display each sub-structure. FIGURE 2. shows an example (drawn from an agenda management application) of the job that templates help Humanoid perform. FIGURE 2.a. shows a tree of application objects used in an agenda management application. The ovals represent objects, and the small black rectangles represent slots. The figure shows an agenda object named Agenda-1, with a commands and a tasks slot. The commands slot has values Close and Open, which are commands to open and close detailed views of the agenda items. The tasks slot holds agenda items, which are the representations of activities a user wants to be reminded about. Each task object has relevant-object and sub-tasks slots. The relevant-object slot holds an object in the knowledge base that the task is related to. The sub-tasks slot holds a set of objects that represent the steps to perform a task.

FIGURE 2.c. shows the display of a task. The top part of the display presents the relevant-object, which in this case represents an object in an inventory management system. The middle part of the display shows a button labelled Close, which presents the close command mentioned above. The bottom part of the displays presents the sub-tasks.

FIGURE 2.b. shows a section of the widget tree that produces the display. The structure of the widget tree closely matches the structure of the display. Agenda-Item-Widget-1 is the head of the tree, and represents the whole display. Agenda-Item-Widget-1 has three children: Inventory-Object-Widget-1 represents the part of the display that presents the relevant object, Panel-Widget-1 represents the close command button, and Task-Widget-1 represents the four sub-tasks. Each of the children of these widgets represent smaller portions of the display. The leaves of the tree represent the graphical objects that appear on the display. For example, Icon-Widget-1 represents the small black rectangle in FIGURE 2.c., and Text-Widget-1 represents its label (“Examine LSA Anomalies”).

The arrows pointing from boxes in the widget tree to ovals in the application objects tree represent the connections between widget instances and objects in the application. For example, the pointer from the top of the widget tree (Agenda-Item-Widget-1) to Task-2 means that the widget tree constructs the display for Task-2. The child widgets of Agenda-Item-Widget-1, and their children contain pointers to objects related to Task-2. For example, the slot labelled object-data in Inventory-Object-Widget-1 points to Inventory-Object-1, which is the relevant-object of Task-2.

FIGURE 2. illustrates the difficult issues that must be addressed when mapping application objects to widget trees:

- The widget to present the relevant-object is dependent on the type of relevant object. Since different tasks can have different types of relevant object, Humanoid must determine at run time the widget to display it.
- The widget tree can pull information from anywhere in the application tree. For example, Button-Widget-1 pulls the information from the commands slot of Agenda-1, and not from a child of Task-2.
- The number of sub-tasks of a task is determined at run time, so Humanoid needs to be able to instantiate as many Sub-Task-Widgets as there are sub-tasks in a task.

The job of templates is to guide Humanoid in constructing the widget tree and connecting it to the application tree. The specification of a template consists of four parts:

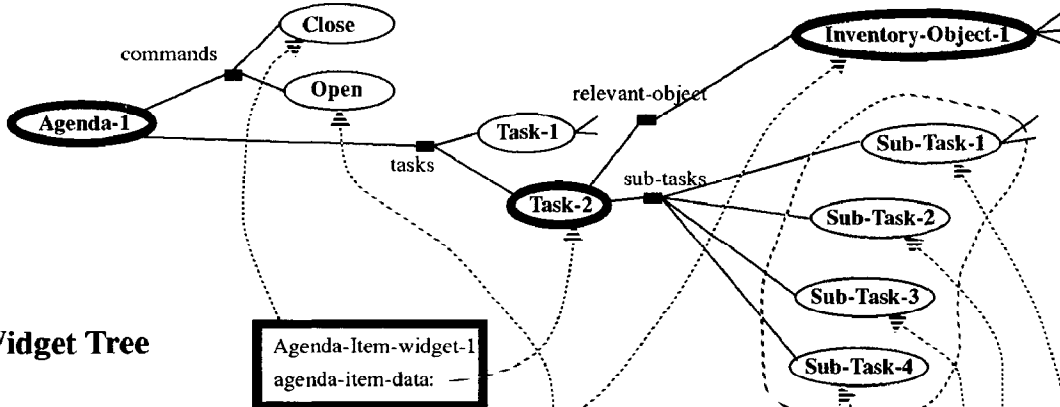
- **Widget:** the widget used for displaying information (e.g. Text, Circle, Radio Button, Panel). Layout is specified in composite widgets (e.g. Panel), which specify the layout of their child widgets.
- **Data slots:** descriptions of the objects that can be used as fillers of the widget’s slots. These descriptions range from specifications of the type of object that can legally fill a slot, to specifications of how to compute the slot values in terms of the values of other data slots.
- **Parts:** descriptions of the child widgets. Humanoid uses these descriptions at run time to select an appropriate child widget based on the characteristics of the application object to be displayed.
- **Parent:** a template from where a template inherits widget, data slot and part information. The parent relation organizes templates in an inheritance hierarchy, facilitating the definition of new templates as elaborations of existing templates.

Humanoid’s algorithm to construct a widget tree for an object has four steps. The input to that algorithm is an object, or a set of objects, and a template. The result is a tree of widgets to display the given object. The steps are the following:

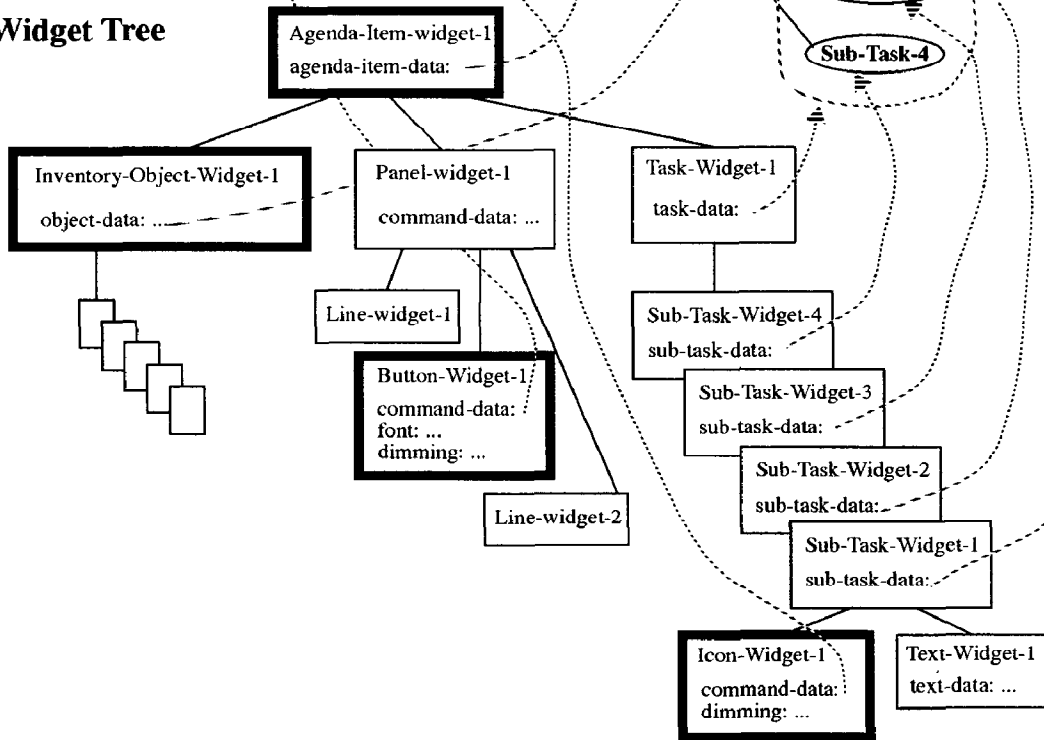
1. Select from the template library an elaboration of the given template that is more specifically suited to display the given object. The result is a template whose data slots are of a type that more closely matches the type of the given objects. The selection algorithm is described in detail in section 5.0.
2. Make an instance of the widget specified in the template found in the previous step, and fill in the appropriate data slot with the given object.

FIGURE 2. A tree of application objects for an agenda management application, the tree of widgets to display it, and the resulting display. The heavy borders on ovals and rectangles highlight elements referenced in the text so that they are easy to find. The ... represent arrows omitted from the figure in order to enhance readability.

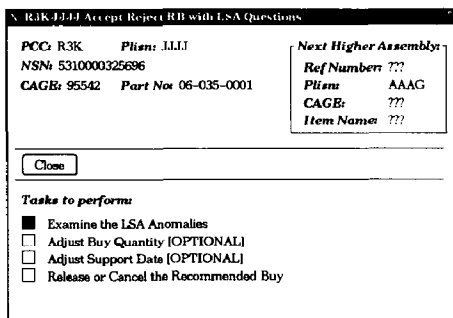
a. Application Objects



b. Widget Tree



c. Display



} relevant object displayed with Inventory-Object-Widget-1

} commands displayed with Panel-Widget-1

} sub-tasks displayed with Task-Widget-1

In the example shown in FIGURE 2. Humanoid creates an instance of the Agenda-Item-Widget named Agenda-Item-Widget-1, and fills in the slot agenda-item-data with Task-2, the agenda object to be displayed.

3. Construct child widgets for the widget created in step 2. Child widgets are constructed according to the parts specification of the template selected in step 1. Each part specifies two things. First, it gives Humanoid instructions for computing the data objects to be presented by the child widget, as a function of the data objects presented by ancestor widgets (this is discussed in detail below). Second, the part specifies a template that Humanoid should use to recursively call this algorithm via step 1. By entering this algorithm recursively, Humanoid will select a more specifically suited template to display the data for the part. When this algorithm bottoms out, Humanoid will have constructed a complete tree of widgets.

In the example shown in FIGURE 2. Humanoid constructs three child widgets for Agenda-Item-Widget-1, corresponding to each of the parts specified in its template. Suppose that the template specifies that the part should display the relevant object of the task using Generic-Object-Template¹. According to the part specification, the object to be displayed by the part is Inventory-Object-1. The template to use is Generic-Object-Template. Suppose that entering step 1 recursively instructs Humanoid to use a template called Inventory-Object-Template rather than Generic-Object-Template. Humanoid then makes an instance of Inventory-Object-Widget, and stores inventory-object-1 in its object-data slot (see FIGURE 2).

4. Solve the layout constraints of the widget instance tree and display it.

In the example shown in FIGURE 2. Humanoid produces the display shown in the bottom part of the figure.

The aspects of this algorithm that give Humanoid its power are the methods to pass data down a widget tree, and the template selection methods used in step 1. The rest of this section discusses the data passing methods, and section 5.0 discusses the template selection methods.

Humanoid provides several methods to specify how the data slots of child widgets should be filled in. The methods supported provide the flexibility to pull information from arbitrary nodes in the application tree, and also provide the flexibility to instruct Humanoid to make multiple instances of a widget when multiple values are to be presented. The methods range from simple ones where the value to be

stored in a widget slot is provided explicitly, to methods that retrieve the value from other widgets in the widget tree, to methods that go through a level of indirection in order to let the application objects compute, at run time, the values to be presented.

FIGURE 2. illustrates the use of several of these methods.

- The template that generates Inventory-Object-Widget-1 uses the method that retrieves the value for the object-data slot from another object. The method instructs Humanoid to go to Agenda-Item-Widget-1, get the value of slot agenda-item-data (Task-2), and then use the value of relevant-object slot in Task-2.
- The template that generates the four widgets labelled Sub-Task-Widget-1 to Sub-Task-Widget-4 uses the method that instructs humanoid to create one widget instance for each of the sub-tasks of Task-2.
- The dimming slot of the template that generates Icon-Widget-1 uses a method that tests whether the sub-task stored in the command-data is already open. If the sub-task is open, the dimming slot is set to true, and the little box in the sub-tasks display is grayed out to indicate that the open command is not available.

The indirect method is not used in FIGURE 2., but it is used in Generic-Object-Template, which is a template to display the values of any object in the system. It works as follows: all objects in the system have a slot called presentable-slots, whose value is the set of all slots defined for the object. The Generic-Object-Template uses the indirect method using presentable-slots to first compute the set of all slots that have values, and then creates a widget instance for each slot/value pair. This allows Generic-Object-Template to display any object in the system by displaying all the slots and values of the object.

Humanoid uses the data declarations not only to construct the displays of objects, but to automatically maintain the displays up to date when the slots of the objects change. Whenever a slot of an object displayed by Humanoid changes, the knowledge representation systems informs Humanoid. Humanoid collects the changes, and when asked to update the screen, processes all the changes, reconstructing the relevant portions of the widget instance tree. The automatic update procedure works even for structural changes to the application objects. When necessary, whole sub-trees of the widget instance tree might be destroyed or created in the process. The interface designer is completely isolated from the complexities involved in keeping the screen up to date.

1. Generic-Object-Template is a template that can display any object in the system by showing a table of all the slot/value pairs of the object.

5.0 Template Selection

Humanoid's template selection mechanism is designed to enable Humanoid to select from the template library the most specifically suited template to display an object. The selection is performed at run time, enabling Humanoid to construct displays adapted to the run time characteristics of the data.

Template selection uses the information stored in the data slots specification of templates. As was discussed in section 4.0, the data slots specification of a template declares the types of the objects that can be used to fill in the slots of a widget. The essence of the template selection algorithm is to find, given a set of objects to display, a template whose data slot types most closely match the types of the objects to be displayed.

The selection algorithm makes use of *predicate subsumption*, a feature of the Loom knowledge representation language [8]. The basic idea is that for a restricted set of predicates, Loom can automatically determine whether one predicate subsumes another. Predicate subsumption is similar to the sub-type relation in the type hierarchies of object-oriented programming languages, except that the types can be fairly general predicates. Loom computes predicate subsumption when the predicates are defined, so its use at run time is efficient (the rest of this discussion uses the terms *predicate* and *type* as synonymous).

The following is an English translation of the predicate (type) *Single-Command-Input-With-Alternatives*, used in the data slots specification of the template to display command inputs² as scrollable menus:

Single-Command-Input-With-Alternatives is a
 Command-Input such that
 the *input-max-number* is 1
 the *input-type* is a sub-predicate of *Set*.

This predicate is true of *Command-Inputs* that can have only one input value, and where the value is of a type that is subsumed by *Set*, that is, the value is one of a set of values. These conditions make the input a good candidate to be displayed as a scrollable menu.

The selection algorithm makes use of the template inheritance hierarchy. As was mentioned in section 4.0, new templates are defined as elaborations of existing templates. When a template elaborates another template, it inherits all

2. *Command-Inputs* are objects that specify the information to request in a dialogue box. The slots *input-min-number* and *input-max-number* specify the number of values to request, and the slot *input-type* holds a predicate that specifies the kind of object to request.

its parent's attributes, the widget, the data slot definitions, and the parts. The elaborations can add or eliminate parts, redefine data slots so that they reference objects in different ways, and override the widget so that the visual appearance is different.

The selection algorithm works as follows. Given a set of objects to be displayed, and a template, Humanoid first selects among all elaborations of the template those whose data predicates match the objects to be displayed, and then selects the best one by comparing the selected templates based on predicate subsumption of their data slots.

For example, suppose that when Humanoid displays the relevant-object depicted in FIGURE 2., the template selection algorithm finds two elaborations of *Generic-Object-Template* specialized to inventory items. Humanoid chooses between these two templates by testing for subsumption the type of inventory item they can present, and choosing the template with the most specific type (if the types cannot be compared, then Humanoid chooses randomly between the two templates). Hence the display of the relevant object at the bottom of the figure has a special formatting designed for inventory items, different from the simple tabular display that *Generic-Object-Template* would have produced.

Humanoid's template selection algorithm can search hierarchies of objects other than the elaboration hierarchy. For example, templates can be organized in a *replacement* hierarchy. When a template *A* replaces a template *B*, it means that *A* can be used instead of *B*, as long as the object to be displayed matches the data predicates of *A*. The replacement hierarchy is useful when one wants template *A* to be used instead of *B*, but *B* and *A* are so different that *A* cannot be defined as an elaboration of *B* (recall that templates inherit all their attributes through the elaboration hierarchy).

For example, the dialogue box template specifies that all its input parts are to be displayed using the *Type-In-Template*. Hence, by default, all inputs are prompted for in type-in buffers. The template to display inputs as scrollable menus is so different from the *Type-In-Template*, that it does not elaborate it. In order to allow Humanoid to consider *Scrollable-Menu-Template* when displaying the inputs, it is defined as a *replacement* for *Type-In-Template*. So, if an input satisfies *Single-Command-Input-With-Alternatives* (defined above), it will be displayed as a scrollable menu.

The combination of the two template selection methods has the following desirable benefits, which are not available in systems that cannot determine at run time how to display an object:

- Humanoid's ability to search for templates enables designers to write compact and reusable specifications of hierarchical displays whose parts can be of varying types. The template to display agenda items is an example of this: it is used to present agenda items, no matter what kind of relevant object they have. The different display formats for different kinds of relevant-objects are obtained through template selection. Thus, Humanoid eliminates the need to write code to select the appropriate display routine based on the data to be displayed.
- The ability to search for templates also facilitates extension of applications. For example, there is no need to change the specification of how to display agenda items when a new kind of agenda item with a new kind of relevant object is added to the system. Only the way to display the new kind of relevant object needs to be added to the system. The template selection mechanisms will select the appropriate template to display the relevant-object. Thus, Humanoid minimizes the need to change existing interface specifications when new object types are added to an application.

Loom's automatic predicate subsumption provides an additional useful service to Humanoid: it makes the template hierarchy easier to maintain. For example, suppose a user interface designer wants to add a template to display radio buttons in dialogue boxes. The designer defines the relevant template and uses the following predicate for the template's data slot:

```
Single-Command-Input-With-Few-Alternatives is a
  Command-Input such that
    the input-max-number is 1
    the input-type is a sub-predicate of Small-Set.
```

Loom automatically determines that *Single-Command-Input-With-Alternatives* subsumes *Single-Command-Input-With-Few-Alternatives*. So, Humanoid's selection algorithm selects the radio buttons template when given an input object whose input-type is a *Small-Set*. If the automatic subsumption detection was not there, the designer would have to compare the *Single-Command-Input-With-Alternatives* with the predicates attached to all templates in the library, and explicitly state which one subsumes the other. Otherwise, it would be impossible for Humanoid to know that radio-buttons should be preferred to scrollable menus when the set of alternatives is a *Small-Set*. This feature of Loom, in conjunction with Humanoid's selection algorithm makes the template library easier to maintain.

6.0 Implementation and Experience with Humanoid

Humanoid is implemented in CommonLisp using the Loom knowledge representation language [8], and the Garnet user interface toolkit [12]. Humanoid has been used to implement the interface for three applications: an inventory control system, an agenda management tool, and a knowledge base browser. An interface to Humanoid itself is under development. The experience with the system has uncovered several problems:

- **Efficiency:** considerable effort has been made to implement template selection efficiently. In the current implementation about 30% of the time to construct a display is spent on template selection and the rest in creating widget instances and solving their layout constraints. However, the system is designing the interface at run time, and as the template library becomes larger and templates more complicated and expressive, Humanoid will need to save portions of design results so that they do not have to be recalculated.
- **Expressiveness of the templates:** for instance, the current templates do not take into account the amount of space that a display occupies, or the shape of the display, making it impossible to do template selection based on how much screen space is available. We plan to address this problem by annotating widgets with hints about their space requirements.
- **Ease of use:** Humanoid is not yet as easy to use as interactive interface builders. Interface definition is indirect, and resembles programming. It is often difficult to understand why a display looks the way it does because it is difficult to know what templates are available in the library, and it is difficult to understand how they interact.
Ease of use can be increased by providing a development environment that borrows features from interactive interface builders, and knowledge-base development environments [6]. For example, template definition can be done interactively [7], examples of displays can be shown after each modification to a template, an interactive constraint editor such as Lapidary [12] can be used to define layout constraints.

The experience with Humanoid supports the expected advantages of rule-based interface specification, and specifically of template-based systems:

- **Template-based systems** make it easier to enforce and maintain consistency because the conditions of rules capture the situations when particular interface features should be used. Thus, the same interface features are used under the same conditions. Furthermore, when a

rule is changed, the effects are automatically propagated everywhere the rule is used.

- Template-based systems provide increased automation. Even though interface builders make it easy to build the displays, a human being has to build them. If an application uses many displays, and a design change is required, someone has to interactively edit each of the displays.
- Template-based systems provide increased portability between different displays and interaction devices. For example, to port an interface to a higher resolution screen it is not necessary to go into the interface builder and adjust the sizes of every widget in every display. Instead, changing a few rules to choose larger fonts and increase the space between widgets propagates the changes to every display in the system.
- Template-based systems facilitate maintaining the interface when the underlying application changes. For example, suppose new commands are added to the agenda management application. If the interface is built with an interactive interface builder, the designer would have to draw the button widget for the new command in *all* the displays where the command should appear. If the interface is built with the template-based method, the command widgets would automatically appear in the command panel because the template for the command panel selects all commands associated with tasks. There would be no need to change the interface specification.

Acknowledgements

I wish to thank Bob Neches, Peter Aberg, David Benjamin, Len Friedman, Brian Harp, Bob MacGregor and Bill Swartout for useful comments on drafts of this paper. I thank Peter Aberg, the first user of Humanoid, for his patience and suggestions. This work was supported by the Defense Logistics Agency and DARPA under contract #DACA76-89-D-0002. Contents represent the opinions of the author, and do not reflect official positions of DLA, DARPA, or any other government agency.

References

- [1] Action. ExperTelligence, 5638 Hollister Avenue, #302, Goleta, CA 93117.
- [2] Y. Arens, L. Miller, S. Shapiro and N. Sondheimer. Automatic Construction of User Interface Displays. In *AAAI 88*, pages 808-813, 1988.
- [3] W. Bennett, S. Boies, J. Gould, S. Greene and C. Wiecha. Transformations on a Dialog Tree: Rule-Based Mapping of Content to Style. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 67-75, November 1989.
- [4] L. Cardelli. Building User Interfaces by Direct Manipulation. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software*, Banff, Alberta, pages 152-166, October 1988.
- [5] Inside Macintosh. Addison-Wesley, Reading, Massachusetts, 1985.
- [6] KEE Software Development System User's Manual. KEE 3.0. 1986.
- [7] K. Lai and T. Malone. Object Lens: A Spreadsheet for Cooperative Work. In *Proceedings of the Conference on Computer-Supported Cooperative Work*, pages 115-124, September 1988.
- [8] R. MacGregor. A Deductive Pattern Matcher. In *Proceedings of AAAI' 88, The National Conference on Artificial Intelligence*, August 1988.
- [9] J. Mackinlay. Automating the Design of Graphical Presentations of Relational Information. *ACM Transactions on Graphics*, pages 110-141, April 1986.
- [10] J. McCormack and P. Asente. An overview of the X toolkit. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software*, Banff, Alberta, pages 46-55, October 1988.
- [11] B. Myers *et. al.* The Garnet Toolkit Reference Manuals: Support for Highly-Interactive, Graphical User Interfaces in Lisp. Technical Report CMU-CS-89-196, School of Computer Science, Carnegie Mellon University, November 1989.
- [12] B. Myers, B. Vander Zanden and R. Dannenberg. Creating Graphical Interactive Application Objects by Demonstration. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 95-104, November 1989.
- [13] D. Olsen. MIKE: The Menu Interaction Kontrol Environment. *ACM Transactions on Graphics*, vol 17, no 3, pages 43-50, 1986.
- [14] Prototyper Interface builder for the Macintosh. User's Manual, 1989.
- [15] G. Singh and M. Green. Chisel: A System for Creating Highly Interactive Screen Layouts. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 86-94, November 1989.