

Presentation Models by Example

Pablo Castells
E.T.S.I. Informatica
Universidad Autonoma de Madrid
Ctra. de Colmenar Viejo km. 17
28049 Madrid, Spain
pablo.castells@ii.uam.es

Pedro Szekely
Information Sciences Institute
University of Southern California
4676 Admiralty Way, #1001
Marina del Rey, CA 90292
szekely@isi.edu

Abstract Interface builders and multi-media authoring tools only support the construction of static displays where the components of the display are known at design time (e.g., buttons, menus). High-level UIMSs and automated designers support more sophisticated displays but are not easy to use as they require dealing explicitly with elaborate abstract concepts. This paper describes a GUI development environment, HandsOn, where complex displays of dynamically changing data can be constructed by direct manipulation. HandsOn integrates principles of graphic design, supports constraint-based layout, and has facilities for easily specifying the layout of collections of data. The system incorporates Programming By Example techniques to relieve the designer from having to deal with abstractions, and relies on a model-based language for the representation of the displays being constructed and as a means to provide information for the tool to reason about.

Keywords User interface development tools, model-based user interfaces, direct manipulation, programming by example, data visualization, graphic design.

1. Introduction

Visual tools for GUI development have greatly contributed to alleviate the effort involved in interface construction [8], and one can hardly conceive GUI development nowadays without the assistance of a graphical editor of some sort. Visual builders save time, require very little knowledge from the developer, and help improve the quality of displays. However, the tools we know today are confined to the construction of the static portion of presentations and provide very little or no support for the dynamic aspects of interface displays. The main reason for this is that the level of abstraction of the visual languages these tools provide is very low, which on the one hand favors their ease of use, but on the other makes it very hard to specify procedural information.

Research in the field of Programming By Example (PBE) has shown that it is possible to overcome these limitations by including inference capabilities and domain knowledge to make it possible to build abstractions by manipulating concrete objects [6, 9]. However the results achieved to date tend to lack the reliability required for a wide implantation in GUI technology. Inference entails unpredictability and lack of control when the user is not provided with all the relevant information about the state of the application and the steps taken by the system. The difficulty resides in finding the appropriate form to convey this information.

On the other end, certain high-level systems like UIMSs and model-based tools support sophisticated interface features [2, 3, 13, 14, 15] but they are hard to use as they require learning a particular specification language and understanding non-trivial abstract concepts. While some of these tools have been complemented with graphical editors, the interaction with the developer tends to be based on menus, property-sheets, and the like. Typically, direct manipulation is supported for the customization of displays after they are generated, but the designer is not provided with adequate control over the generation process itself. Carrying the kind of abstract and complex underlying concepts and mechanisms as these systems use to an intuitive visual environment is a difficult problem in general.

Even for the static part of displays, interface builders and multimedia authoring tools do not provide adequate support. While graphic design has become an essential part of the development of GUI products, few if any tools support it, or they do in a very limited way. The layout facilities are patterned after the layout facilities of drawing editors where groups of elements can be left-aligned, right-aligned, etc. Graphic designers often work by defining guides and grids to organize page layouts [1, 16].

Our research aims at extending the expressive power of existing visual tools for the construction of a significant range of dynamic displays while retaining the ease of use of direct manipulation, providing facilities for constructing well structured and visually appealing screen layouts. Our approach consists of a) using the model-based paradigm for the internal representation of the constructed displays, with models that support dynamic presentation functionalities, b) developing an extended visual language with the appropriate level of abstraction, that incorporates PBE techniques for the interactive specification of dynamic presentations by manipulating interface presentation objects in a visual tool, c) allowing the developer to create examples of application data at design-time, and to use them to construct concrete presentations that are generalized by the system, and d) incorporating high-level graphic design facilities that help improve interface quality and simplify the construction of complex layouts. These ideas have been carried to a GUI development tool, HandsOn (Human-Amiable tool for building Neat Display Structures by working ON examples), for the interactive construction of presentation models [5].

Figures 1 through 3 show examples of displays built in HandsOn. Figure 1 is the famous Minard chart showing Napoleon's march to Moscow. The thickness of the line encodes the number of troops in Napoleon's army, the line darkness encodes the temperature (darker is hotter). The squares and labels indicate places where battles took place. The input data consists of two lists of records. One containing information about latitude, longitude, number of troops, and temperature, and the other list containing records of the time and places where battles took place. This figure is an example of a custom designed display that cannot be produced by any charting program. Sage [11, 12] can automatically generate this chart from the relational data, but it requires that each tuple provide the two end-points of each line. In HandsOn this display can be modeled independently of the format in which the data comes in (list of points or list of intervals).

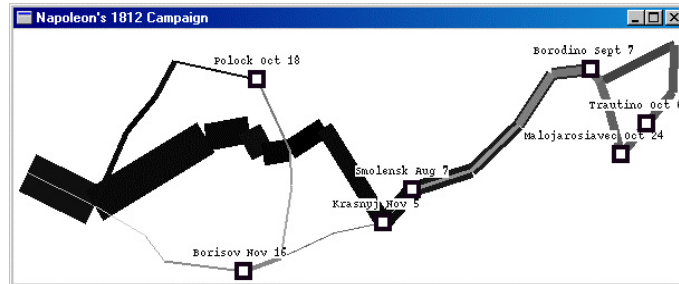


Fig. 1. Napoleon's march to Moscow

Figure 2 shows a composite bar-chart. The input data is a list of three records one for each person. The record for each person itself contains a list of records about the activities that the person is managing. This chart cannot be produced by charting programs, but can be produced by Sage. The difficulty in generating this display is that it consists of two charts put side by side in a coordinated way, and each chart itself is a hierarchical composition of an outline display (the data for each person) with a chart (the activities managed by each person).

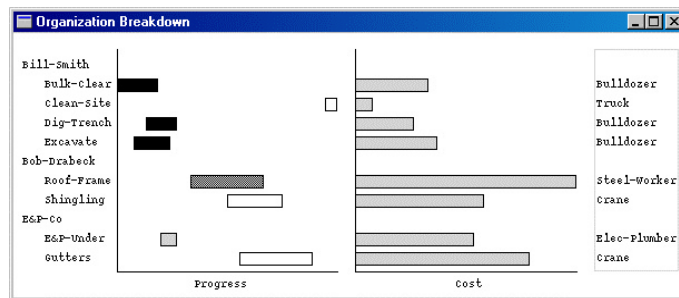


Fig. 2. Complex bar-chart

Figure 3 shows a tree structure where the width of each node is dependent on the information contained in that node. This figure is interesting because it shows that HandsOn supports recursively defined models.

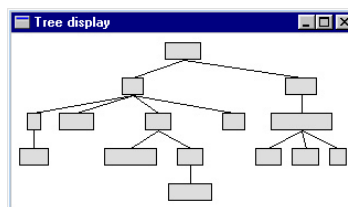


Fig. 3. Tree display

The rest of the paper is organized as follows. The next section gives a quick description of the main components of the system. Section 3 gives an overview of the underlying interface model used for the construction of displays, and how models are visualized graphically. Section 4 describes the PBE techniques used in our system to define constraints and control structures, which are illustrated in section 5 by showing a complete example.

2. System overview

HandsOn takes its presentation model from previous work in Mastermind (see [4]). The model language has been conceived to be easily amenable to interactive specification and bridges the gap between the declarative descriptions obtained from a graphical tool and the procedural information needed to execute the interfaces described in the model. Visual languages are well suited for the description of static shapes, i.e. declarative information, but it is hard to obtain procedural information this way. The model-based approach helps by providing a declarative representation to model dynamic display behavior. Declarative models are constructed in a graphical environment, and executable presentations are generated from the information collected in the model. Figure 4 shows the architecture of the HandsOn presentation generation system.

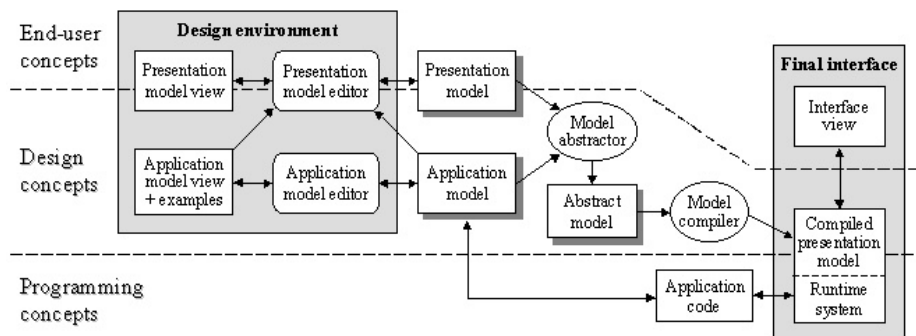


Fig. 4. HandsOn architecture

The HandsOn development environment consists of a graphical presentation editor and an application data builder (see fig. 6). The latter is used to create examples of application data, and the former to construct presentations. The developer uses data examples to provide values at design-time for the construction of interface presentations that display application data at run-time. The designer can browse the classes that are defined in the application, select classes, create instances of the classes in the application data builder and edit their contents to construct application object examples.

The data builder displays data as a tree where composite data (objects and lists) can be expanded or collapsed. HandsOn provides direct manipulation facilities for connecting data from the application examples built by the designer to graphical com-

ponents of the presentation. Rather than building a generic display, the designer constructs specific displays using specific data and the system generates abstract constructs by generalizing the examples. Data examples provide the designer with concrete objects to refer to, and they provide the system with information that the system uses to infer the designer's intent.

Tightly integrated in both the visual language and the presentation model, HandsOn provides graphic design tools like movable guides and dynamic grids for global screen space organization. These tools are not provided merely as passive visual aids, but in the form of design objects that interact actively with other objects of the display, playing a central role in defining the dynamic behavior of presentations.

The output that results from the presentation editor is a model of the interface presentation containing references to application data examples. From this model a generic model is generated by generalizing from the examples, inferring data characterizations to parameterize presentations. The abstract model is translated into an executable representation that is given to the run-time system to generate and manage displays at run-time. The run-time system maps the compiled abstract model to actual application data to generate the final display that is presented to the end-user.

3. The model

The presentation model specifies the structure and graphical components of displays, how the components are connected to application data, the visual appearance of each component, and how the components are laid out.

3.1 Basic constructs

The model consists of a part-whole hierarchy of graphical objects that is defined by assembling graphical primitives and predefined widgets from a widget library. Each presentation component has three kinds of parameters that determine different properties: visual parameters, which include guides and other magnitudes like width and height, style parameters like color and font, and data parameters that store data from the application. Components can serve as prototypes to create instances, i.e. copies that dynamically inherit properties from the prototype.

Predefined components have their own set of primitive parameters that control their basic graphic properties, but new parameters can be added by the designer. All components have at least four natural guides (the bounding box) that define components' size and position. Other guides can be added, like middle guides, a baseline for text labels, or other arbitrary guides. Two of these guides define the origin of coordinates for all the other guides and sub-parts of the component.

The model supports one-directional constraints on parameters with respect to other parameters and/or application data. Constraints are themselves model objects with their own parameters that can in turn be constrained, allowing for the incremental construction of complex constraints. Predefined components usually have default constraints on some of their parameters and guides (e.g. the width parameter of graphic primitives depends on left and right).

Other dynamic constructs supported by the model include iterative displays that show variable amounts of application data. Iterative constructs are described by a presentation component and a collection of application values. They are created at run-time by creating one replica of the component per value in the sequence of data. Typically, certain settings of the copies depend on the corresponding values of the list. If the list of values changes, the replication is automatically updated accordingly. The presentation model also allows the specification of conditional displays whose structure, layout and visual appearance depends dynamically on the data to be presented or the presentation context (e.g. available screen space).

The application model includes standard types, sequences, object class definitions, and specific example objects created by the developer. Application examples are the keystone for constructing abstract control structures without using an abstract (visual or textual) language. Our application model requires the application to be written or wrapped in a programming framework that supports dynamic object creation, object browsing, dynamic method invocation and a mechanism for the notification of changes in object attribute values. In our prototype implementation we have assumed that the part of the application that is relevant for the interface is written in Amulet [7], but a more general platform like CORBA would meet our requirements as well, and would impose less restrictions on the application programmer.

3.2 Visualizing the model

Presentation models are visualized in the graphical editor under a representation that is very similar to the final resulting interface. When abstract constructs are visualized that depend on application data, data examples are used to provide the designer with a concrete representation.

Certain elements appear in the editor that are not part of the final interface. It is the case of guides, grids and parameters. Guides and grids are represented by vertical or horizontal lines and sets of lines respectively. Non-guide parameters are visualized as a box showing the name of the parameter. When selected, their value is shown and can be edited if it is not constrained. To avoid clutter, these auxiliary objects are hidden most of the time, but there are (implicit and explicit) commands to bring them up for selected objects. Figure 5 shows a portion of the presentation editor work area where a text label, a rectangle, and a line have been created. The parameters of all three objects are being visualized; label *WIDTH* and line *THICKNESS* are selected. The bounding-box guides of the rectangle are visible, and other guides are hidden.

Guides and parameter icons are shown in blue when they are not constrained, and in red otherwise (this cannot be seen in the grayscale figure). Free guides can be moved and free parameters' value can be edited, so that dependent guides and parameters, if any, are updated accordingly. Constrained guides and parameters cannot be altered directly unless the designer wants to customize the constraint by demonstration, as we will describe in section 4.1.

Constraints involving selected parameters can be visualized in the editor as red arrows that connect an output variable to one or more input variables. In figure 5, the top guide of the rectangle is constrained to be placed a fixed offset above the

horizontal guide near the bottom of the figure. The designer can free parameters and guides, change input variables or constrain other parameters by moving constraint-arrow ends to different objects. Selected constraints can be copied and pasted in the editor, so that new unconnected arrows appear that can be linked to other objects, defining new constraint instances that take their settings after the original constraint.

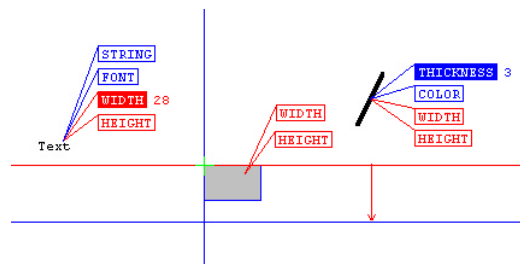


Fig. 5. Visualizing model objects in the presentation editor.

4. Building presentations by example

The designer creates display components by selecting graphical primitives and predefined components from a widget library. The graphical editor allows grouping components, adding guides, grids or new parts to aggregate components, visualizing component parameters and editing their value, and other common usual facilities of graphical editors. The designer can define constraints visually and customize them by demonstrating the desired effects. By manipulating application data examples the designer can define data dependencies and abstract control structures.

4.1 Defining constraints

HandsOn supports one-directional constraints that consist of the composition of a set of standard functions like linear functions and max/min, application object attribute access expressions (data constraints), and invocation of object methods. The designer can set constraints visually on presentation parameters by dragging or pointing at parameters, guides, and application data, within or across the presentation and application models. We use the generic term *link operation* to refer to all gestures that indicate a direct association between two presentation elements, like dragging a guide onto another guide, which sets an equality constraint on the moved guide with respect to the other guide when *guide-snapping-mode* is on. A more general way to perform link operations is to press the *link* command button, point at a guide, a parameter or a presentation component, and drag the mouse onto a guide, a presentation parameter or a value being displayed in the application area. A rubber-band red arrow from the object to the mouse pointer provides interim feedback for the operation.

The meaning of these gestures and the kind of constraints that result depend on the nature and characteristics of the variables involved, and a few global environment settings like *guide-snapping-mode* and *constraint-demonstration-mode*. The action to

be taken in response to the designer's manipulations often involve other operations besides creating constraints, and is determined by a set of heuristics that examine the presentation and application objects involved each time the designer performs a link operation. Also certain restrictions apply that prevent the creation of illegal constraints like the ones that would cause circularities.

For instance in the construction of the Minard chart shown in figure 1, if the designer links the *THICKNESS* parameter of a line segment to the *TROOPS* attribute of an object of type *Stage* constructed in the data area, the system defines an equality constraint from the value of the attribute to the thickness of the line, if constraint-demonstration-mode is off. This means that the thickness of the line will be equal to the troop size at the corresponding march stage, which will probably range over several hundred thousands units. In this case, this is probably not what the designer wants. Rather, the constraint should include a scaling factor to adjust the application data to the desired dimensions of the chart on the screen, as is usually the case when displays take numeric magnitudes from application data. This is why the system automatically defines a linear transformation constraint when the designer links numeric values if demonstration mode is on. The linear transformation is defined to conform to the current values of the variables being connected, giving default values to underdetermined coefficients. In the previous example, in demonstration mode the system sets $THICKNESS = a \cdot TROOPS + b$ where b is assigned the default value 0 and a is assigned the quotient of the current values of *THICKNESS* and *TROOPS*.

The designer can modify the coefficients of linear constraints by demonstration by editing the constrained values. This mechanism is activated when the constraint demonstration mode is on. A constraint is customized by creating a copy of the constraint and connecting it to a new set of variables. The new set of values plus the original one provide two equations involving the coefficients of the constraint. By repeating the operation, more equations are obtained. Linear constraints keep track of the maximum consistent set of most recently provided equations obtained in this way so that the coefficients of the linear transformation can be determined by solving the system of equations (if there are not enough equations, the appropriate number of coefficients are taken as constants). Once the equations are solved, the result is applied to all instances of the constraint. By editing values involved in one of the constraint instances, the corresponding equation is modified accordingly and the coefficients are recalculated. Value sets can also be provided without copying constraints by detaching a constraint from its variables, modifying the value of the variables and reattaching the constraint.

When HandsOn is not in constraint demonstration mode, constraint output variables cannot be changed, and when input values are modified, the constrained parameters are automatically adjusted to enforce the constraint.

4.2 Control structures

Each time a presentation parameter is mapped to a data value, HandsOn examines the data structure containing the value, all the way up to the top-level of the structure, looking for structural properties like recursivity, iteration, pre-existing links to presentation objects, and other characteristics of the data. This analysis provides the

system with hints to infer control structures to display the data, like replications, recursive presentations and conditionals.

Replications are collections of variable amounts of components that correspond to sequences of application data. To build a replication the designer needs to describe a generic component and how it maps to application data, and then specify how the remaining copies should be created using a list of data, and how they should be laid out. In HandsOn the designer builds replications for specific examples of data sequences, and the system generalizes the constructed presentation for lists of values of the same type.

The simplest way to define a replication is to link a guide or a parameter of a presentation component to a value displayed in the application model area that belongs to a list or to a data structure contained in a list. When the designer does so, upon confirmation the system assumes that a replication is being defined. The component to which the guide or parameter belongs is the component to replicate, and the data sequence is the list that contains the value to which the presentation was linked. Each replication component will have a constraint to its associated application data as defined by the link operation between the original component and the data. HandsOn is able to generate nested replications for nested data lists, but does not handle more than two nesting levels at a time. Treating more than two nesting levels at once would produce a cluttered construct almost impossible to handle for the designer.

By using example values the designer can see at design-time how the replication will look like. By manipulating the generated replicas, the designer defines how the replication should be laid out. The designer can also refine the presentation of replicas by editing the first one, changing its visual settings and defining more constraints to data, so that the system automatically propagates changes to the remaining replicas. How actions on individual replicas are translated to remaining replicas is determined by a set of heuristics that select iteration variables and constants among the elements involved.

For example, to build the chart shown in figure 1, the designer creates a list of objects of type *Stage* in the data area (assuming the data is organized as a list of intervals rather than a list of points), and draws a line segment in the presentation area. Then the designer links the *THICKNESS* parameter of the line to the *TROOPS* attribute of the first *Stage* (see fig. 6). Because the stage object appears inside a list, HandsOn infers a replication of line segments with respect to stages, with line thickness constrained to stage troops. Now the designer links the guides that determine the x and y coordinates of the start point of the first replicated line to the attributes *LATITUDE* and *LONGITUDE* of the *START_POINT* of the first stage. Automatically every replicated line's start point takes its coordinates from the corresponding stage. Line end points are constrained to stage *END_POINTS* coordinates in a similar way. Were the data given as a list of points, the designer would attach the end point of the first segment to the second point in the data sequence, so that replicated segments would be associated to two instead of one element of the sequence.

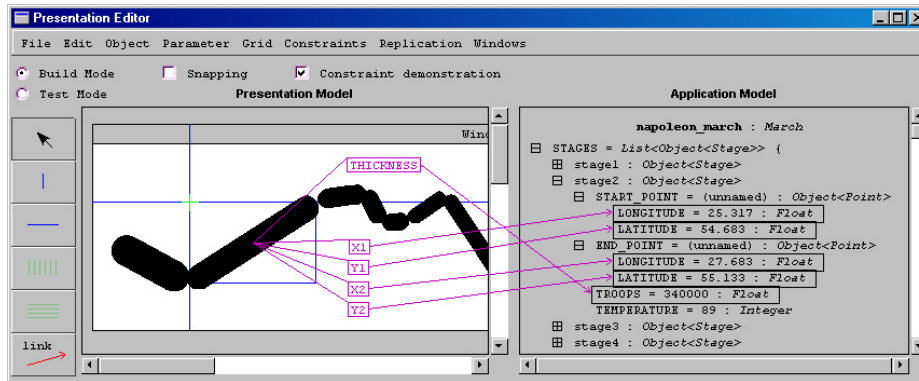


Fig. 6. The HandsOn environment: data constraints and replication of line segment

HandsOn is also able to infer presentations for recursive data structures like trees and networks. The system detects a recursive presentation when the designer links two presentation objects of the same type to two respective objects in the application area that have a containment relationship, i.e. one object can be accessed from the other through a traversal of successive object attributes and list members. After asking for the designer's confirmation, the system builds a recursive presentation by defining an abstract component prototype that consists of grouping the two graphical objects plus other related components, and taking the whole abstract prototype as the model for the second graphical object involved in the recursion. The recursion prototype has an input parameter that is assigned successive values from the recursive data structure until an empty list or a NULL object is reached, or until an object is reached that has already been displayed in the recursive presentation. Related components to be included in the recursion prototype, like arcs between nodes of a tree, are determined by heuristics that check for constraint relationships with the two initial objects used to define the recursion. These additional components can be added before or after the recursion is defined.

If the containment relationship in the recursive data structure involves list membership, recursion is combined with replication, as is the case in the tree display shown in figure 3 (recursion over tree levels and iteration over node children). HandsOn does not currently handle the case where recursion involves nested lists.

A limited form of conditional presentation is also currently supported in the visual environment. When the designer links two variables that have different types, the system first tries to find an appropriate transformation that is compatible with the types (see section 4.4 below). If no defined transformation function is applicable, HandsOn creates a conditional definition for the constrained value based on a switch-like statement that is defined by tuples of value correspondences. A more general mechanism for the specification of conditional presentation constructs is currently under development.

4.3 Layout

The MASTERMIND layout facilities are based on the grid design techniques used in graphic design. By manipulating guides and grids the designer determines the size and positioning of presentation objects. Guides are horizontal or vertical reference lines, and grids are sets of horizontal or vertical lines. Grids are characterized by four quantities: the number of lines, the separation between the lines, the start and the end positions. A grid is defined by specifying three out of the four quantities. Other presentation parameters are edited by entering values from the keyboard or by defining constraints to other values. By setting constraints on guides, grids and parameters, self-adjusting layouts are constructed.

For example, a horizontal grid can be used in the display shown in figure 2 for the layout of labels and bars. The grid spacing can be defined in terms of the font size of the text being displayed, and the top and bottom of the grid can be defined to match horizontal guides at the top and bottom of the window. Interface builders do not provide this kind of facilities for good page design. Some interface builders provide grids, but they are used just for initial placement of the items. It is not possible, for instance to specify the grid size based on font size, so that if the font size is changed at run-time, the design doesn't break apart.

Guides and grids are also crucial for defining the layout of replicated parts. In the simplest case, a replication can be assigned to a grid, meaning that consecutive replicas are placed in consecutive grid-lines. The designer attaches the primary replica to one or more grid lines and the system infers the positioning of the rest of the objects across the grid. HandsOn includes heuristics for the adjustment of the layout strategy by manipulating individual replicas: it is possible to specify the grid-line index for the first replica, to specify the number of grid-lines to be occupied by each replica, to specify that each replica should go to the next free grid-line, etc. Nested replications can be assigned to a common grid, so the elements are placed sequentially on the common grid (see fig. 7). Recursive presentations can also be laid out with respect to a single grid, as in the tree display (fig. 3).

4.4 Data descriptions

To infer the appropriate effects from the designer's actions, HandsOn uses information obtained by examining model properties and design context information. The system analyzes value types, visual properties and geometric relationships among the objects being manipulated, structural patterns of the data being used, existing mappings from data to presentations, and data visualization context. This information is used to generate automatically presentation constructs and to produce generalizations from the examples provided by the designer. The latter involves generating abstract references to data values and presentation objects, and substituting concrete values by presentation variables whose value is computed at run-time according to the way data references are described.

Data reference descriptions involve describing the data structure traversal and the transformation functions that have to be applied in order to compute values. These descriptions are generated when the designer assigns data examples to presentation

parameters. Transformation functions can be specified by the designer or inferred by the system by examining the types of involved variables. When the designer links two variables that have different types, HandsOn tries to construct a compatible constraint by considering object method calls (if an application object is involved) and standard conversion functions (like string to color or number to string) that agree with the types. The designer is asked for a choice if more than one function is possible (if none is found by the system or provided by the designer, a conditional is defined as explained in 4.3). This is how, for example, progress bars in figure 2 can have their left and right (integers) attached to task start and end dates respectively, if *Date* objects have a *date_to_int* method. In demonstration mode, if the return value of the constraint is a number, a linear transformation is added for scaling as described in 4.1.

Structure traversal specifications are generated to match the structure of the data being displayed in the application area. When a value being assigned to a presentation is nested in a higher-level data structure, HandsOn infers a data description for the value that involves accessing the nested value from the top-level data structure displayed at this time. References to application examples are used to parameterize presentations. Some of the data examples give rise to presentation input arguments and others become data access expressions. Whether a data reference is generalized in one way or the other depends on the context in which the data is displayed at design-time. An input parameter is created for objects that are displayed at the top-level in the application example builder at the time when the designer connects presentations to data, and all other values become internal parameters that take their value from input parameters, which states a natural rule that should be easy to understand for designers.

5. An example

The bar-chart display (fig. 2) is a good example of the use of replications, grids and guides. The display consists of two nested replications. The outer replication corresponds to the list of workers that perform a work like repairing a building, and the inner replication corresponds to the list of tasks that each worker performs. For each task the display shows its name, a bar that indicates the start and end dates of the task and its status (different colors for finished, on course, cancelled), a bar that corresponds to task cost, and a label showing the resource used for the task.

5.1 Top-level replication

The designer starts by creating the labels that display worker names. S/he creates a window in the presentation design area and adds a label to the window client area. Then the designer creates an object of type *Works* in the application area, gives it the name *repair_building* and fills it in with example values down to the innermost levels, creating a complex data structure of nested objects and lists (see fig. 7).

At the top-level, *repair_building* has a single attribute whose value is a list of objects of type *Worker*. The designer links the *STRING* parameter of the label to the attribute *NAME* of the first worker, *bill_smith*. The object *bill_smith* appearing inside a list of

objects, HandsOn, upon confirmation, automatically replicates the original label with respect to the list of workers, attaching labels' text to worker names.

The original label's position did not have any constraint for the system to propagate, so the replicated labels are given a default arbitrary positioning close to the first label. The designer adds a horizontal grid to the window, and adjusts grid line spacing as desired. S/he activates guide snapping and drags the first label's top guide onto the first grid line, defining a constraint that is automatically propagated to the remaining replicas, attaching each label's top to successive grid lines.

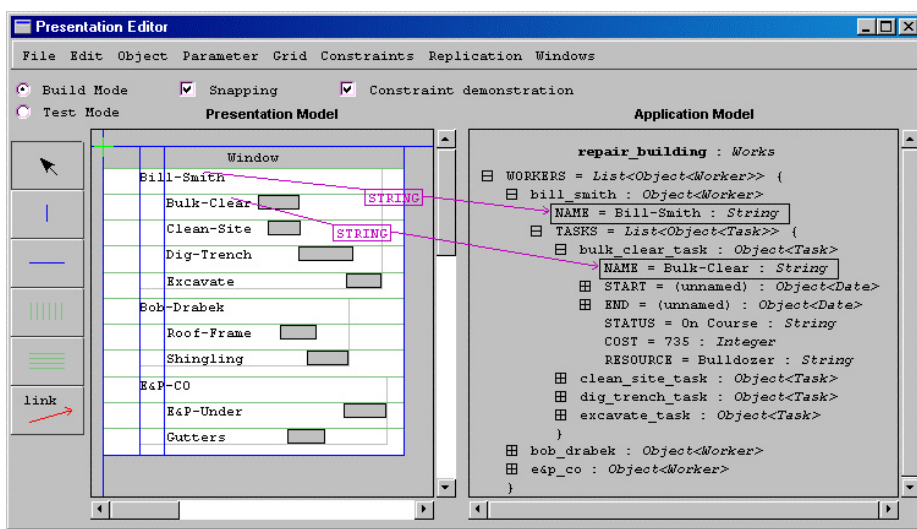


Fig. 7. Building and laying out nested replications in the HandsOn environment

In order to make room for the worker's tasks display, the designer moves the second label down by two grid lines. This modifies the replication layout rule so that replicas are placed every three grid lines. The designer left-aligns the labels by adding a vertical guide to the window and linking the first label's left guide to it, so that all labels take this constraint.

5.2 Editing replications

Now the designer adds labels for the tasks performed by each worker: the designer adds a label next to the first worker name label, and groups both labels. HandsOn propagates this operation so now the replication is made of groups of two labels. Groups are visualized as gray-border rectangles that comprise their components.

Worker objects have a *TASKS* attribute that is a list of the tasks the worker performs. The designer links the new label's *STRING* parameter to the attribute *NAME* of the first task of *bill_smith*, *excavate_task*. The system notices that *excavate_task* belongs to a list, so a replication of labels is created showing the name of tasks performed by *bill_smith*. Because the replicated label belongs to a replica associated to *bill_smith*, similar replications are generated in the remaining replicas of the top-level iteration,

substituting *bill_smith* by the value (the worker) that corresponds to each replica.

The designer now attaches the first label's top to the grid line immediately below the *bill_smith* label. Task labels of all workers are laid out vertically in response, taking one grid line per label starting after each worker's name label grid line. To avoid overlapping and wasted space between worker groups, the designer moves down the second group to the grid line immediately below the first group. As a consequence, the worker replication is laid out according to the rule "take as many grid lines as needed". The designer left-aligns task name labels by adding a vertical guide and linking the first label's left to it (see fig. 7).

5.3 Constraint customization

Bars for task progress and task cost, and labels for task resources are added for all task replicas just by adding them to the first replica. The left and right position of progress bars should correspond to the start and end dates of the corresponding task. The designer links the left guide of the *bulk_clear_task* progress bar to the *START* attribute of the *bulk_clear_task* object. Types do not match so upon designer's acceptance HandsOn creates a constraint using the *date_to_int* method of the *Date* object class, followed by a linear transformation that is given default coefficients, as described in 4.1. The constraint is tuned under demonstration mode by copying and attaching it to the bar's right and the task end date, and by manually manipulating date values and bar ends to obtain the desired effect. As before, settings are transmitted to the corresponding replicated parts.

Cost bars are left aligned using a vertical guide, and their width is linked to task costs, customizing the linear constraint for scaling. Progress bar color is defined as a conditional property by linking different color values to different corresponding status string values.

When the running presentation is generated, the example object *repair_building* turns into an input parameter that is added to the chart top-level widget (i.e. the window). This parameter has to be supplied at run-time with an object of type *Works*, and all the parameters that were linked to data inside *repair_building* are constrained to compute their values from the object stored in the input parameter.

The construction process described is only one of many possible ways to build the display. For instance, it is possible to create all the components that correspond to one particular task (name label, progress bar, cost bar, resource label), group them, replicate the group for all tasks of one particular worker, and then replicate again for all workers. The way we have shown the construction of the example, the display is constructed by columns, paying little attention to the nested structure, whereas in the latter approach the chart is seen as a set of rows and the design is a more direct mapping to the nested data structure.

6. Related work

HandsOn's presentation model is based mostly on Mastermind [4]. One can think of HandsOn as a graphical editor built on top of the Mastermind presentation system in which the designer can construct presentation models by direct manipulation of

display objects without being concerned with how the underlying model is being represented. The main improvement with respect to the Mastermind model is the incorporation of application data examples into the model at design-time, from which generic models are abstracted by the system.

Our model of presentation is similar to Humanoid's templates [11], which also have constructs for replication and conditionals, though Humanoid does not support layout based on grids and guides, and the way they are used in our system for the layout of replications. Humanoid includes a graphical environment for the construction of interfaces, but the designer works mainly on an abstract representation of the model. The designer is provided at design-time with a view of the resulting display, though the direct manipulation facilities supported on this view are intended for browsing rather than for editing the model.

Peridot [9] supports creating sets of widget copies by using examples of data sequences at design-time. Unlike HandsOn, Peridot requires the designer to specify explicitly input parameters for presentations. Peridot uses a very simple data model consisting basically of lists and simple types, whereas HandsOn supports more elaborate relationships involving complex data structures. Peridot automatically infers the layout for sets of widgets by requiring very little information from the designer, but it does not support the construction of complex layouts.

In many respects, HandsOn is similar to automatic presentation planners such as Sage [11, 12] and Gold [10]. These systems allow the designer to associate specific data values to presentation settings, and they analyze characteristics of the data in order to infer properties of the display. However, Gold and Sage can only produce presentations from relational data, whereas HandsOn can model a much larger variety of displays.

HandsOn borrows the idea of using constraints for propagating values in the component tree from Amulet [7]. HandsOn is implemented using Amulet, and the generated models are translated into Amulet objects, making extensive use of the Amulet constraint system.

7. Conclusions

HandsOn combines the expressive power of a model-based system with the ease of use of a programming by example tool. Demonstrational techniques benefit from the model-based paradigm because the model provides an explicit declarative representation of the interface the tool can reason about. HandsOn infers presentation constructs by analyzing model information describing value types, structural properties of data, and spatial relationships in the presentation. The system also uses information about how the designer visualizes and handles data examples.

The direct manipulation techniques supported are qualitatively more powerful than those provided by interface builders and multi-media authoring tools in that they allow the specification of abstract constructs, which are used to create both static and dynamic displays. Balance between expressive power and ease of use is achieved by a) keeping limited the amount and complexity of model abstraction the developer has

to deal with and b) mapping underlying model abstractions to visible presentation objects, reducing the mental transformation effort required from the developer between different representations of the interface.

HandsOn is currently under development, though most of the features described here have been essentially completed. The system has been implemented in C++ using Amulet [7], both for the design environment and for running the generated interfaces. The generated interfaces and the interface constructs visualized in the environment at design-time share a great part of their functionality —the designer works on presentations that have the same dynamic behavior as the executable display that results. However the generation of interface code from the model is still only partially implemented.

We are also improving the conditional presentation mechanism described in this paper to support displays whose structure and appearance depends on arbitrary conditions on data values, presentation parameters, and platform characteristics. The action side of the conditionals will consist of alternative presentations that can be specified by the designer either by constructing each one from scratch or by successively applying the desired modifications for each condition to a single component. Mastermind provides a model for this kind of constructs [4, 14]. The main difficulty now resides in specifying the model by demonstrating the conditions that determine the presentation to be applied.

Our work so far has focussed on the visual part of interface design and does not currently address aspects related to the dialog with the end-user. Our plans for the near future include the extension of our work to a more comprehensive environment that provides support for interactive aspects based on user task modeling.

Acknowledgements

Our thanks to the anonymous reviewers for their detailed feedback and helpful comments. The work reported in this paper was partially supported by the *Plan Nacional de Investigación*, Spain, Project Number TIC96-0723-C02-02.

References

1. R. Ballinger. *Layout and Graphic Design*. Van Nostrand Reinhold, New York, 1970.
2. Bauer, B. *Generating User Interfaces from Formal Specifications of the Application*. Proceedings of 2nd International Workshop on Computer-Aided Design of User Interfaces (CADUI'96). Presses Universitaires de Namur, 1996.
3. F. Bodart, A.-M. Hennebert, J.-M. Leheureux, I. Provot, B. Sacre, J. Vanderdonckt. *Towards a Systematic Building of Software Architectures: the Trident Methodological Guide*. Proceedings of 2nd Eurographics Workshop on Design, Specification, Verification of Interactive Systems (DSV-IS'95). Springer-Verlag, 1995.

4. P. Castells, P. Szekely and E. Salcher. *Declarative Models of Presentation*. Proceedings of International Conference on Intelligent Interfaces (IUI'96). Orlando, Florida, 1997.
5. P. Castells and P. Szekely. *HandsOn: Dynamic Interface Presentations by Example*. To appear in Proceedings of 8th International Conference on Human-Computer Interaction (HCI International '99). Munich, Germany, 1999.
6. A. Cypher (ed.). *Watch What I Do: Programming by Demonstration*. The MIT Press, 1993.
7. B. A. Myers et al. *The Amulet 2.0 Reference Manual*. Carnegie Mellon University Tech. Report, 1996.
8. B. A. Myers. *User Interface Software Tools*. ACM Transactions on Computer Human Interaction, v2, n1, March 1995, pp. 64-103.
9. B. A. Myers. *Creating User Interfaces by Demonstration*. Academic Press, San Diego, 1988.
10. B. A. Myers, J. Goldstein, M. Goldberg. *Creating Charts by Demonstration*. Proceedings of the CHI'94 Conference. ACM Press, Boston, April 1994.
11. S. F. Roth and J. Mattis *Data Characterization for Intelligent Graphics Presentation*. Proceedings of SIGCHI'90 Human Factors in Computing Systems. ACM press, Seattle ,WA, April 1990, pp. 193-200.
12. S.F. Roth, J. Kolojejchick, J. Mattis, and J. Goldstein. *Interactive Graphic Design Using Automatic Presentation Knowledge*. Proceedings of the CHI'94 Conference. ACM Press, Boston, April 1994, pp. 112-117
13. P. Szekely, P. Luo, and R. Neches. *Beyond Interface Builders: Model-Based Interface Tools*. Proceedings of INTERCHI'93, April 1993.
14. P. Szekely, P. Sukaviriya, P. Castells, J. Muthukumarasamy and E. Salcher. *Declarative Interface Models for User Interface Construction: The Mastermind Approach*. In Engineering for Human-Computer Interaction, L. Bass and C. Unger (eds.). Chapman & Hall, 1996.
15. A. Wiecha, W. Bennett, S. Boies, J. Gould and S. Greene. *ITS: A Tool For Rapidly Developing Interactive Applications*. ACM Transactions on Information Systems 8(3), July 1990, pp. 204-236.
16. R. Williams. *The Non-Designer Design Book*. Peachpit Press Inc., Berkeley, California, 1994.