Pervasive Application Development and the WYSIWYG Pitfall

Lawrence D. Bergman[1], Tatiana Kichkaylo[2], Guruduth Banavar[1], and Jeremy Sussman[1]

[1] IBM T.J. Watson Research Center, 30 Saw Mill River Road, Hawthorne, NY 10532
[2] Department of Computer Science, New York University
{bergmanl, banavar, jsussman}@us.ibm.com, kichkay@cs.nyu.edu

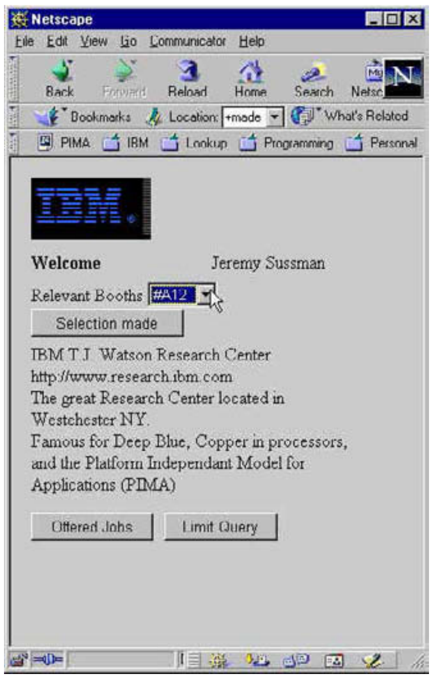
Abstract. Development of application front-ends that are designed for deployment on multiple devices requires facilities for specifying device-independent semantics. This paper focuses on the user-interface requirements for specifying device-independent layout constraints. We describe a device independent application model, and detail a set of high-level constraints that support automated layout on a wide variety of target platforms. We then focus on the problems that are inherent in any single-view direct-manipulation WYSIWYG interface for specifying such constraints. We propose a two-view interface designed to address those problems, and discuss how this interface effectively meets the requirements of abstract specification for pervasive applications.

1 Introduction

The challenge of writing applications has broadened in recent years because of a proliferation of portable devices, like personal digital assistants (PDA's) and programmable phones. These devices have varying physical resources, including differing display sizes and special I/O mechanisms. As a consequence, application front-ends, which include the user interface and some control logic such as event handlers, often must be written from scratch for each type of device on which that application is to run. This imposes an immense development and maintenance burden, particularly since the developer may not be aware of all the devices on which the application is to be deployed. Indeed, an application may be run on hardware platforms that were not even in existence when the application was written!

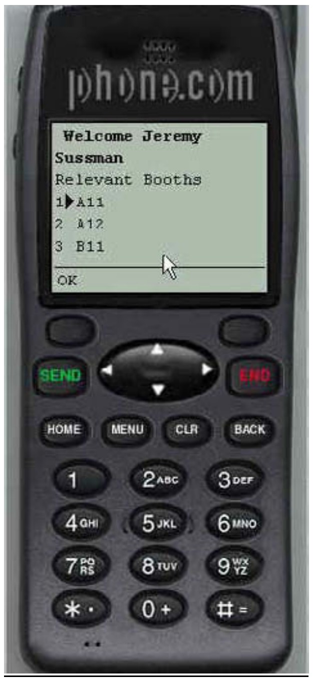
Figure 1 shows an example. The three parts of the figure show a portion of the same application (browsing of information at a job fair) on a PC running java, on a web-browser, and on a mobile phone. Notice the differences in the amount of information contained on a screen, as well as in the layout. The ideal is to have a single specification that produces all of these rendered applications.

There are two common approaches to solving the problem of specifying application front-ends that are intended to run on multiple devices. The first is to use a device- neutral specification language and library, with run-times for that language


Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netsc
Bookmarks Location: +mode What's Related
PIMA IBM Lookup Programming Personal
IBM

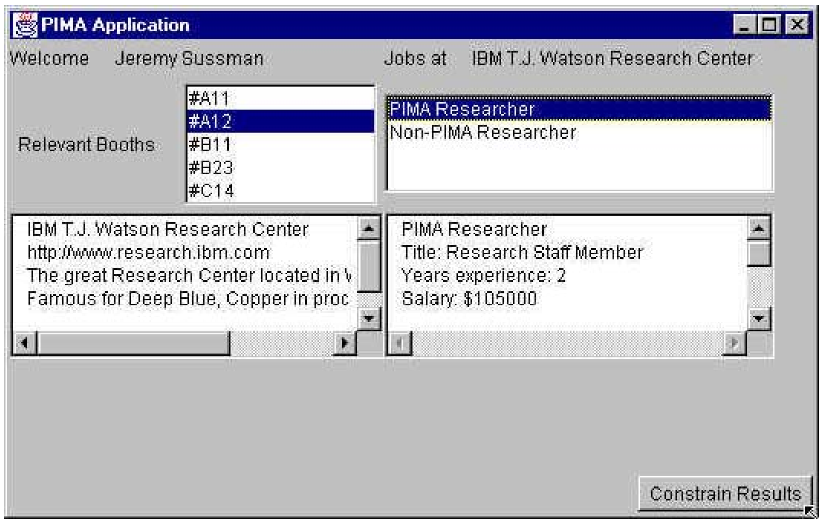
Welcome Jeremy Sussman
Relevant Booths #A12
Selection made
IBM T.J. Watson Research Center
http://www.research.ibm.com
The great Research Center located in Westchester NY.
Famous for Deep Blue, Copper in processors, and the Platform Independant Model for Applications (PIMA)
Offered Jobs Limit Query

(a) Browser (HTML)



Welcome Jeremy Sussman
Relevant Booths
1 A11
2 A12
3 B11
OK

(b) Mobile phone (WML)



Welcome Jeremy Sussman Jobs at IBM T.J. Watson Research Center

Relevant Booths #A11 #A12 #B11 #B23 #C14

PIMA Researcher
Non-PIMA Researcher

IBM T.J. Watson Research Center
http://www.research.ibm.com
The great Research Center located in \
Famous for Deep Blue, Copper in proc

PIMA Researcher
Title: Research Staff Member
Years experience: 2
Salary: $105000

Constrain Results

(c) Java

Fig. 1. Platform independent application (job fair) deployed on three platforms

deployed on all target devices. This is the approach taken by Java [10]. The problem here is that device characteristics can be radically different, so attempts to run a single user-interface design on multiple platforms is bound to give poor results. Indeed, on some devices, such as cellular phones, the UI assumptions underlying AWT do not hold at all.

The other approach is "style sheet" customization. This allows the designer to craft a device-specific set of layout rules for each individual device, using a style sheet transformation language such as XSLT [18]. The problem with this approach is that a style sheet is not application-specific; the same style sheet is typically applied to all applications to be run on a particular device. The style sheet overrides any application-specific characteristics.

An additional possibility, of course, is to develop a different style sheet for each application-device pair. This leads to the problem of both developing and managing a large number of style sheets – exactly the problem addressed by device-independent application development.

We are developing a framework that attempts to address the problem of creating application front-ends that can be developed once, and that will run reasonably on a wide variety of hardware platforms. Our goal is "write once, right anywhere," application development. Our approach is to provide an application model that allows the designer to create a device-independent specification of the front-end. Part of the specification is a set of device-independent *layout semantics*, which are specified as a set of *constraints*. Although the layout produced by the specification should be reasonable on all devices, the designer may wish to produce interfaces tailored for certain platforms. By providing layout constraints and overrides, the designer can customize the interface to particular sets of devices (e.g., all mobile phones with display screens). We call this process *specialization*. Specializations can be specified broadly, for example, *all* GUI devices, or narrowly, for example, a single device. When specializing for a single device, we allow the designer to use the convenience of a WYSIWYG-style interface. We call this style of specialization *tweaking*. The discussion in this paper will focus on specification of device-independent semantics for specialization, rather than on device-specific specialization or tweaking.

The main contributions of this work are:

- A constraint model for specifying layout semantics that is to be applied to generating user-interfaces for multiple target devices.
- A discussion of problems with single-view direct-manipulation "what you see is what you get" (WYSIWYG) interfaces for specifying device-independent layout constraints.
- A proposed two-view interface design and implementation that addresses these issues.

2 Related Work

The work described here builds on previous work in the areas of model-based user interface systems, constraint-based interface specification, and multi-view systems. We do not claim to advance any of these areas in this paper. Instead, we focus on the unique problems that arise when model-based systems with constraints are applied to the area of multi-device application development, and how a multi-view system can potentially solve these problems.

Model-based user interface systems [15], [16] allow interfaces to be specified by constructing a declarative model of how the interface should look and behave. A run-time engine executes the model and constructs the application's displays and interprets input according to the information in the model. Of particular relevance in this area are the various task models of user interaction [4], [11]. Our application development model described in the following section is an example of a model-based user-interface specification. In contrast to previous work, however, the design of our application model was driven specifically by the requirements of multi-device application development.

Model-based systems pose a class of problems having to do with developers' difficulty in maintaining a mental model that connects an abstract interface specification with multiple concrete realizations of it. One problem in this class, called the "mapping problem", was identified by Puerta and Eisenstein [13]. The mapping problem is the difficulty of mapping an abstract interface element to multiple concrete elements, thus reducing the usability of model-based systems. To solve this problem, the authors propose a new model component, called the *design model*, to support the inspection and setting of mappings via the use of intelligent tools. The problem that this paper focuses on – the difficulty of simultaneously viewing device-independent specifications and device-specific layouts derived from those specifications – is another instance in the class of problems mentioned above.

There has been some recent work in the area of "intent-based" markup languages [1], which are textual languages for declaratively specifying multi-device interfaces. However, this work is yet to address the issue of developing tools for creating applications using such languages.

Another related area of research is the creation of user-interfaces via demonstration. The Amulet project is one example of such work [12].

Much work has been done in the area of constraints for user interface development. Constraint-based windowing systems include SCWM [2] and Smalltalk [6]. There are several constraint-based drawing systems such as [14]. Constraints have been applied to a variety of user-interface problems, including database user interfaces [7], programming by demonstration [8], and data visualization [17]. Constraint solvers for user-interface systems have been developed [5], [9]. However, we are not aware of any work that has applied constraints to the problem of multi-device user interface development.

3 The Application Development Model

In this section we will give a very brief, high-level view of our application development model. Although the task model described is not particularly novel, we present it in order to lay the groundwork for the central discussion of this paper – mechanisms for specifying layout semantics.

An application developed using our framework consists of one or more *tasks*. A task is a unit of work to be performed by the user. Examples of tasks include registering for a subscription service, placing an online order, or browsing a catalog.

A task may be made up of *subtasks*, with the granularity of the lowest level tasks completely at the discretion of the designer.

Lowest level tasks or *leaf tasks* contain *interaction elements*. Interaction elements provide for user input and/or system output. Interaction elements provide abstract descriptions of entities that may be rendered as widgets on GUI devices, or voice elements in a speech-based interface. Examples of interaction elements include *SelectableList*, which presents the user with a list of data items from which the user is to choose (e.g., a pull-down menu), and *Input* which allows the user to enter information (e.g., a type-in field).

In addition to tasks and interaction elements, the application designer specifies variables and event handlers, navigation between tasks, and sets of layout constraints. The constraints are used to create to page layouts for each particular device, and for determining the number and contents of pages for the rendered application.

4 User-Interface Constraint Specification

In this section, we will describe the set of constraints that we wish to specify for our models. It is important to keep in mind that we are not trying to address the problem of laying out user interfaces targeted for particular devices; this ground has been well-covered by previous investigators. We are addressing the problem of devising sets of constraints to be used for generating interfaces for multiple target platforms. For this reason, we focus primarily on semantic constraints.

In this discussion, we distinguish between two types of user-interface constraints that can be specified for a device-independent application. Constraints can either be *generic*, applicable to any type of input or output device, or they can be *graphical*, applicable only to devices with traditional display screens.

4.1 Generic Constraints

Generic constraints are completely device-independent. We currently support two types of generic constraints: ordering constraints and grouping constraints.

Ordering constraints specify the sequencing of interaction elements in the interface. For example, if an address form has a name field, a street address field, and a city/state field, we want to specify that they be presented in that order. *First* and *last* constraints specify the relative positioning of individual interaction elements within a leaf task. In addition, ordering between pairs of interaction elements can be specified. This type of constraint, which we call *after*, operates on two interaction elements, specifying that one element is to be positioned anywhere after the other[1].

Grouping constraints specify that interaction elements are semantically related, and should be kept together (in a GUI, on the same screen and adjacent) in the rendering of the user-interface. An example is that all the elements in the address form discussed above are related, and would constitute a group. The single grouping

[1] For simplicity, we limited the interface to this small number of ordering constraints. It may be that others such as *immediately after* are sufficiently valuable to warrant inclusion.

constraint, *group*, can be applied to any number of interaction elements within a leaf task.

For purposes of constraint specification, groups are treated just like interaction elements. In other words, a group can be selected as an argument for any constraint operator, including the grouping operator (i.e., groups can be nested).

Group is a high-level semantic construct that can be interpreted in different ways by run-time implementations for different devices. We readily envision more device-specific grouping constraints such as *group by row*, or *group by column* – these particular examples only applying to GUI's.

Note that both generic ordering and grouping constraints are semantic specifications, applicable to any class of device, with device-specific presentation differing from device to device. *First*, for example, on a large-screen visual display would specify the interaction element is to be rendered as a widget positioned in the upper-left corner of the display. On a small-screen display (a mobile phone, for example), rendering of the interaction element might fill an entire screen, thus *first* would indicate the first screen. On a voice interface, on the other hand, *first* would indicate the first event in the voice interactions, either a voice-input or a voice-output item. Similarly *after* clearly specifies temporal ordering for a speech interface, but is subject to interpretation on a GUI – probably producing some sort of text-flow (e.g., down and/or to the left) ordering.

4.2 Graphical Constraints

Graphical constraints apply only to devices with visual displays. We currently support two types – sizing constraints and anchoring constraints.

Sizing constraints are used to ensure that members of a set of interaction elements are all rendered with the same width (using the *same width* constraint) or the same height (using the *same height* constraint). This facilitates the design of interfaces that conform to standard UI guidelines. Note that we specify sizing and grouping separately. This allows us to specify that all buttons are to be rendered the same size for a particular task, but without requiring that the buttons be adjacent.

Anchoring constraints are used to position interaction elements on a screen, giving a designer some control over element placement. The four anchoring constraints, *top*, *bottom*, *left*, and *right* allow a designer to place particular elements at the boundaries of the visual display – permitting a set of buttons to be positioned at the bottom of the screen, for example.

The set of constraints described here is by no means complete. This set does not come close to supporting the degree of control possible with a UI toolkit such as AWT or Motif. This is by intent. Our goal here is provide an easy-to-use, high-level set of constraints that can be used to specify interface characteristics to be applied across a wide range of possible devices. We anticipate that designers who want truly beautiful interfaces for particular target platforms will take the output of our specialization engine, and tweak it by hand, as mentioned in the introduction. Our goal is to produce a usable interface for any device, with provisions for a designer to improve the default interface for particular devices or sets of devices.

5 The WYSIWYG Interface Problem

In designing a constraint-specification interface for user-interface layout, it is tempting to develop a single-view, direct-manipulation, "what you see is what you get" (WYSIWYG) interface. Such an interface would provide the following functionality, required for any design environment of this sort:

1. ***Selecting items.*** A WYSIWYG view can be used for selecting user-interface components on which operations are to be performed (positioned, sized, etc). This is common practice, allowing the designer to quickly and effectively specify the items in context.
2. ***Viewing constraints.*** The WYSIWYG view can also be used to display the constraints by visually identifying the interface components and the constraints that apply to them. This is attractive, because it minimizes the cognitive load on the designer. S/he can see which interaction elements have constraints applied to them, and what those constraints are, while reviewing the visual appearance of the interface.
3. ***Viewing layout***. It is critical that any interface design system provide visual feedback to the designer. A WYSIWYG interface shows the effects of changes in the layout specification by presenting a representation of the sizes and positions of the user interface components as they will appear in the final application.

Although it seems desirable to provide all of this functionality in a single view, there are several serious problems in using a single WYSIWYG view as a direct-manipulation interface when specifying constraints for device-independent applications[2]. Some of these problems have to do with the fact that the application will, in general, be laid out across multiple screens, some of them have to do with the fact that the application is to be deployed on multiple platforms. These problems, in order of decreasing importance, are as follows:

1. The most serious drawback to a WYSIWYG interface is that the user is "led down the garden path." What the user sees is *not* what the user is going to get in general, since only a single device is emulated in the interface. The user is being tempted to customize the design for *one particular* device, rather than thinking about the general problem of constraints that are appropriate for *all* devices. Even though an interface may provide a capability for toggling between device emulations, thereby allowing a view of multiple layouts, this feature is easily overlooked. A possible solution is to simultaneously provide multiple views. This can easily lead to confusion between viewing and control (in the model/view/controller sense), however.
2. A WYSIWYG screen layout changes each time a constraint is specified. If the WYSIWYG view is also being used as a set of direct-manipulation controls, this has the effect of moving those controls after each operation, a highly undesirable characteristic. Furthermore, the constraint solver may move some of the interaction elements to different screens (i.e., pages), compounding the sense of dislocation experienced by the user, particularly if only one virtual screen is displayed at a time – interaction elements will disappear from view. This seriously

[2] Single-view WYSIWYG interfaces may be effective for customizing a design for a particular device, a process we call *tweaking*, but this is not the device-independent design we are discussing here.

reduces the usability of the interface. It may be possible to partially alleviate this problem by having the user explicitly specify when the interface is to be re-laid out, but this breaks the WYSIWYG model.

3. Constraints cannot always be satisfied. For example, the screen size may not be large enough to contain all interaction elements in a group. A single-view WYSIWYG interface may be unable to display the constraint in that case (e.g., displaying *group* as some form of visual containment is not possible). Switching the emulation to a larger device may allow that constraint to be included, and hence displayed. This is misleading and confusing behavior.
4. The user may wish to specify constraints between interaction elements on separate screens. For example, *after* or *group* constraints may involve interaction elements on more than one screen. This makes the specification a bit awkward, since the user must switch screens while selecting interaction elements. Either multiple screens must be displayed simultaneously, which can strain screen resources, or the designer will need to toggle between screens. Intuitive constraint displays such as arrows for *after* or visual containment for *group* (using a bounding box, for example) becomes problematic. The interface needs to rely more heavily on less obvious visual metaphors and/or user memory.
5. A WYSIWYG interface, because it presents *only* final appearance, lacks information about the structure of the task model, information that might be of value to the interface designer. In specifying layout constraints, it may be important to know with which task or subtask particular interaction elements are associated. Although it would be possible to provide some of this information – by labeling emulated screens with subtask names, for example – it is difficult to envision an interface that will provide all of this structural information in the context provided by a WYSIWYG view.

From this discussion, it should be clear that a better interface paradigm is required. In the next section, we discuss an alternative to the single-view, direct-manipulation WYSIWYG interface, and describe our implementation.

6 A Solution: The Two-View Constraint Editor

In this section we will discuss the desired characteristics of a two-view constraint editor, describe our implementation, and explain how the two-view interface solved the “WYSIWYG pitfall.”

6.1 Desired Characteristics

The fundamental problem with a single-view direct-manipulation WYSIWYG editor is that the single view is not adequate to display both the logical structure of the constraint set within the context of the task model, as well as an indication of the types of layouts produced by that constraint set. What is desired is a single logical view of the task structure and interaction elements that could be used both as an interface to specify constraints, and also as a conceptual view of these constraints. The logical view should be arranged to make it easy to see and think about the entire constraint set.

Additionally, the designer should be able to view the effects of the constraints on interfaces that are generated for various devices, but with minimal opportunity for the user to assume that the preview is a "true" representation of what will be produced for multiple devices. A clear separation of the logical view from the device preview should facilitate this. For these reasons, a two-view system is more likely to embody all the desired characteristics of a constraint-specification interface for pervasive application development than a single-view WYSIWYG editor.

6.2 Current Implementation

The constraint-specification interface is one component of an application development (AD) tool for specifying device independent applications. The AD tool is a part of our device-independent application framework, called PIMA (Platform-Independent Model for Applications) [3]. Other components of the AD tool include a task editor for managing task structure and navigation; and a task details editor for specifying interaction elements, variables, and events that comprise individual tasks.

Figure 2 shows the two-view constraint-specification interface with some constraints specified. On the left hand side is a graphical representation of the task structure and interaction elements. We call this the *logical view*. Tasks are represented as labeled, nested rectangles, with gray rectangular icons representing interaction elements contained within them. The logical view serves two purposes. It is used to add and remove constraints, and also to view the set of constraints that are currently specified.

Interaction elements are selected by clicking on them, and then operations are applied to individual interaction elements or groups of interaction elements. The user can select one or more icons using the mouse. Once a set of interaction element icons has been selected, pressing a button corresponding to the desired constraint type specifies a constraint. A visual representation of the constraint is added to the display. In a similar fashion, constraints can be removed.

The right-hand side of the two-view interface is a representation of a screen layout for a single screen on a particular device, with each widget (interaction elements will be presented as widgets in a GUI) represented as a named rectangle, sized and positioned as it would be in the final running application. We call this the *layout view*. In figure 2, the layout view displays an emulation of the first screen of our job fair application, configured for a PC running Java. Each change to the constraint specification triggers an update in the layout view. The user can navigate the screens for a particular device by selecting from a set of radio buttons, or select from a set of different devices by choosing from a selectable list. Each device is emulated using the task/interaction element/constraint description, and a device capabilities file that defines screen size and default widget sizes.

Figure 2 shows the interface after specification of several constraints. Dark circles in the upper left and lower right corners of interaction elements icons in the logical view, indicate first and last constraints, respectively. "Name" has been specified as first, and "URL" as last. Arrows connecting two interaction elements shows ordering. The arrow between "Name" and "Phone" indicates that "Name" is to precede "Phone."

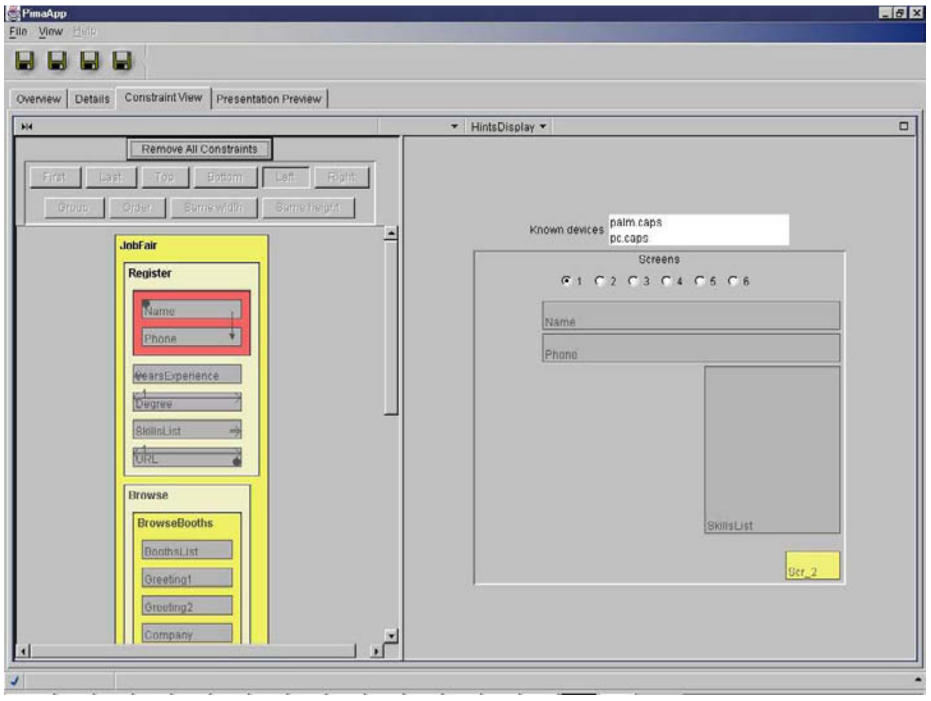


Fig. 2. Two-view layout constraint specification interface

To facilitate legibility, we ensure that all arrows point downward, rearranging icon placement if necessary. For obvious reasons, cycles are detected by the interface and disallowed. Small arrow icons point to the side of interaction element icons for which anchoring has been specified. "Skillslist" is anchored on the right, and "Years of experience" on the left. Double-headed arrows are used to indicate same size constraints – vertical arrows for height constraints, and horizontal arrows for width constraints. The arrows have numbers associated with them, the members of a same-size set all displaying the same number. "Degree" and "URL" have the same width in our example.

Groups are represented by repositioning all entities to be grouped so they are contiguous, and drawing a rectangle that encloses them. The figure shows a single group containing "Name" and "Phone."

Figure 3 shows a later version of the same interface. Notice that we have included two device-specific layout views on the right-hand side (layout for a PC on top, for a browser on the bottom). This allows a user to see the effects of constraints on multiple devices simultaneously. We expect this to further reduce the tendency towards "what you see is what you get" design.

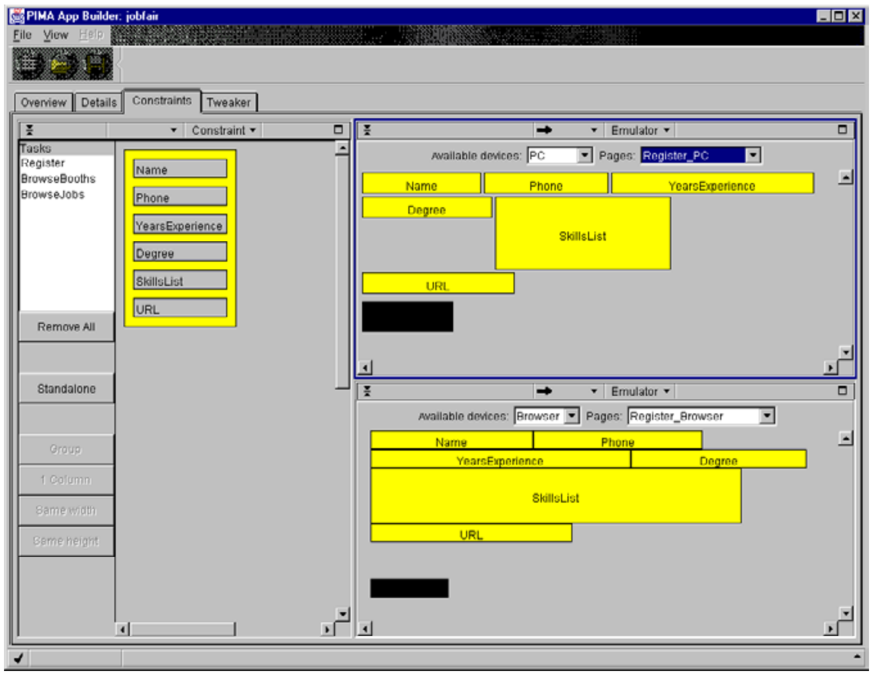


Fig. 3 A later version of the two-view constraint specification interface showing the constraint view on the left, and the device-specific layouts for two different devices on the right

6.3 Solutions to the WYSIWYG Interface Problems

We note that the two-view interface solves the previously described problems of the WYSIWYG interface as follows:

1. Since the logical (interface/constraint display) view is provided as a single, scrollable display, not separate virtual screens; no context switching is required to select multiple interaction elements. Display of ordering is readily achieved on the single panel.
2. Since the logical and layout views are separated, constraints can always be visualized, even if constraints cannot be satisfied for particular devices being emulated.
3. The task model is integrated into the logical view, facilitating high-level, device-independent thinking about the constraint set.
4. Although interface controls can move in this interface (since specification of ordering constraints can lead to re-ordering of the interaction element icons in the logical view), context is much more readily maintained than it would be with a single-view WYSIWYG interface. The task structure helps to retain orientation, and the problem of some of the interaction elements vanishing off-screen for certain operations has been solved. Note that allowing the user to specify when icons are to be reordered would be a viable possibility, and would not pose the problem discussed earlier of breaking a WYSIWYG model.

5. The user is much less tempted to think that the emulation view is “reality” than with a single-view WYSIWYG interface. The separate logical view encourages device-independent thinking.

7 Discussion

The main thrust of the two-view interface is to encourage a designer to think generically. When designing a device-independent application intended to run across a variety of pervasive devices, it is critical that design choices be made on general principles, not based on specific display characteristics. The interface should encourage thinking of the form, “This particular interaction element really belongs at the beginning of the task on any device.” If the designer is thinking, “I see a blank spot at the top of this screen, I’d like to move that widget to the top to fill it,” s/he has missed the point and trouble may ensue.

By separating the logical structure of the application from its presentation view, and integrating the constraint interface with that logical structure, we encourage device-independent design. It is rather difficult for a designer to make the connection between specific constraints and the generated layout. This is by intent. By *not* providing the easy connection, “this widget is at the top of the screen because a ‘top’ constraint was specified,” there will be far less temptation to do “appearance-guided design,” in our opinion the major potential pitfall for device-independent UI development.

Even though we have separated the logical view from the UI appearance, presentation of a single concrete representation of the layout could still pose a problem, leading the user to believe it is *the* view. We have recently added multiple device presentations to our interface. Since these presentations are for viewing only, not for interface control, providing multiple views is less problematic than for a WYSIWYG editor. Whether providing multiple views is an effective solution will require future user studies.

8 Future Work

Although we have presented a problem with interfaces for device-independent user interface development, and proposed a solution, user studies are clearly necessary.
Two questions need to be addressed. The first is, “does the device-independent application model, with specification of device-independent constraints provide enough control – allowing the designer to create applications that run reasonably well on all platforms, and that are easily tweaked to produce high-quality interfaces on selected platforms?”

The second question broadly is, “what is the ‘best’ interface for specifying device-independent interface constraints?” We propose a more focused question, namely, “is a two-view interface more usable for specifying device-independent interface constraints than a WYSIWYG interface?” A user study that pits our two-view interface against a single-view WYSIWYG interface, measuring the quality of the interfaces produced and/or the speed of production should provide valuable insight into the nature of the device-independent application development process, and the tools required to support it. Note that to tweak an interface for a specific platform, a single-view direct-manipulation WYSIWYG interface is clearly appropriate.

Another question that we plan to investigate is, "how does one provide a designer with tools for specifying interface characteristics for classes of devices?" We have discussed two points on a continuum – either creating a generic device-independent specification, or customizing an interface for a particular device type. Clearly there are points in between. We can envision classifying device characteristics – graphical vs. voice, large-screen vs. small screen, presence/absence of hardware inputs such as hard buttons, etc., and then providing different sets of constraints or hints for different classes. Essentially design becomes a process of specifying sets of rules – "if the application is running on a small-screened device, place this widget on its own screen," for example. The question then is, does such a design methodology produce usable interfaces with fewer burdens on the designer than producing an interface separately for each device to be supported? We believe that with a carefully developed methodology, the answer will be yes, both in terms of less work being required, and in providing support for devices not in existence when the application is developed.

9 Conclusions

In this paper we have explored some issues surrounding design of development tools for creating device-independent applications. In particular, we have pointed out particular pitfalls when designing for multiple devices that do not exist when designing an application for a single device, or a very small set of predetermined devices.

The central problem that we have identified is that a design tool must be carefully structured to not tempt a designer to believe that emulations of the interface represent what will actually be displayed on target devices; the targets cannot be known or adequately represented at design-time. We suggest the need for design environments that facilitate intent-based rather than graphical thinking.

We have proposed one solution to this problem – a two-view system for specifying interface characteristics. Although we have not proven the utility of this approach, we have clearly identified the problems that exist, and suggested how different design interface characteristics might alleviate or exacerbate the problem. We have provided a framework for future studies and interface development in this area.

Acknowledgments

We would like to thank Noi Sukaviriya and Rachel Bellamy for illuminating conversations, which helped fuel this work. John Richards and Jonathon Brezin identified the "single-view direct-manipulation WYSIWYG" problem, and were instrumental in helping us formulate a viable solution. John Turek provided valuable support and feedback during the course of this work.

References

1. M. Abrams, C. Phanouriou, A. Batongbacal, S. Williams, and J. Shuster, UIML: An Appliance-Independent XML User Interface Language, in Proceedings of the Eighth International World Wide Web Conference, May 1999, p. 617-630.
2. Greg J. Badros, Jeffrey Nichols, and Alan Borning, SCWM---an Intelligent Constraint-enabled Window Manager, in Proceedings of the AAAI Spring Symposium on Smart Graphics, March 2000.
3. PIMA project home page. http://www.research.ibm.com/PIMA
4. Larry Birnbaum, Ray Bareiss, Tom Hinrichs, and Christopher Johnson, Interface Design Based on Standardized Task Models, in Proceedings of the 1998 International Conference on Intelligent User Interfaces 1998, p.65-72. http://www.acm.org/pubs/articles/proceedings/uist/268389/p65-birnbaum/p65-birnbaum.pdf
5. Alan Borning, Kim Marriott, Peter Stuckey, and Yi Xiao, Solving Linear Arithmetic Constraints for User Interface Applications, in Proceedings of the 1997 ACM Symposium on User Interface Software and Technology, October 1997, p. 87-96. http://www.acm.org/pubs/articles/proceedings/uist/263407/p87-borning/p87-borning.pdf
6. Danny Epstein and Wilf LaLonde, A Smalltalk Window System Based on Constraints, in Proceedings of the 1988 ACM Conference on Object-Oriented Programming Systems, Languages and Applications, San Diego, September 1988, p. 83-94.
7. Phil Gray, Richard Cooper, Jessie Kennedy, Peter Barclay, and Tony Griffiths, Lightweight Presentation Model for Database User Interfaces, in Proceedings of the 4th ERCIM Workshop on “User Interfaces for All” 1998 n.16, p.14. http://www.ics.forth.gr/proj/at-hci/UI4ALL/UI4ALL-98/gray.pdf
8. Takashi Hattori, Programming Constraint System by Demonstration, in Proceedings of the 1999 International Conference on Intelligent User Interfaces 1999, p.202. http://www.acm.org/pubs/articles/proceedings/uist/291080/p202-hattori/p202-hattori.pdf
9. Scott Hudson and Ian Smith, Ultra-Lightweight Constraints, in Proceedings of the ACM Symposium on User Interface Software and Technology 1996, p.147-155. http://www.acm.org/pubs/articles/proceedings/uist/237091/p147-hudson/p147-hudson.pdf
10. http://www.javasoft.com/
11. David Maulsby, Inductive Task Modeling for User Interface Customization, in Proceedings of the 1997 International Conference on Intelligent User Interfaces 1997, p. 233-236. www.acm.org/pubs/articles/proceedings/uist/238218/p233-maulsby/p233-maulsby.pdf
12. Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alan S. Ferrency, Andrew Faulring, Bruce D. Kyle, Andrew Mickish, Alex Klimovitski and Patrick Doane. The Amulet Environment: New Models for Effective User Interface Software Development, IEEE Transactions on Software Engineering, Vol. 23, no. 6. June, 1997. pp. 347-365.
13. Angel Puerta and Jacob Eisenstein, Towards a General Computational Framework for Model-Based Interface Development Systems Model-Based Interfaces, Proceedings of the 1999 International Conference on Intelligent User Interfaces 1999, p.171-178. http://www.acm.org/pubs/articles/proceedings/uist/291080/p171-puerta/p171-puerta.pdf
14. Kathy Ryall, Joe Marks, and Stuart Shieber, An Interactive Constraint-based System for Drawing Graphs, in Proceedings of UIST 1997, Banff, Alberta Canada, October 1997, p. 97-104. http://www.acm.org/pubs/articles/proceedings/uist/263407/p97-ryall/p97-ryall.pdf
15. Piyawadee "Noi" Sukaviriya, James D. Foley, and Todd Griffith, A Second Generation User Interface Design Environment: The Model and the Runtime Architecture, in Proceedings of ACM INTERCHI'93 Conference on Human Factors in Computing Systems 1993, p.375-382. http://www.acm.org/pubs/articles/proceedings/chi/ 169059 /p375-sukaviriya/p375-sukaviriya.pdf
16. Pedro Szekely, Ping Luo, and Robert Neches, Beyond Interface Builders: Model-Based Interface Tools, in Proceedings of ACM INTERCHI'93 Conference on Human Factors in

Computing Systems 1993, p.383-390. http://www.acm.org/pubs/articles/proceedings/chi/169059/p383-szekely/p383-szekely.pdf
17. Allison Woodruff, James Landay, and Michael Stonebraker, Constant Density Visualizations of Non-Uniform Distributions of Data Visualization, Proceedings of the ACM Symposium on User Interface Software and Technology 1998, p.19-28. http://www.acm.org/pubs/articles/proceedings/uist/288392/p19-woodruff/p19-woodruff.pdf
18. XSL Transformations (XSLT) Version 1.0, W3C Recommendation 16, November 1999. http://www.w3.org/TR/xslt See also, www.xslt.com.

Discussion

L. Nigay: I would like to come back to the specialization mechanism. How do you go from abstract widgets to concrete widgets? Do you have a model to describe devices?
L. Bergman: Simple mapping. Only graphical modalities. Thinking of speech input.

K. Schneider: Does the multi-view editor support "tweaking" the user interface? Are the "tweaks" retained when a constraint is added or moved or changed? Can it "tweak" the constraints, such as order, for a particular concrete user interface?
L. Bergman: We are just beginning to address those issues. The editor supports "tweaking". The "tweaks" to the properties of an element are retained but the structural "tweaks" are not. And, yes, you can override the constraints when "tweaking" the user interface.

J. Höhle: I agree that people want/need a hands-on approach to explore/play with the design. But do you really think that tools like MS FrontPage will continue to be wanted? Couldn't your tool provide hands-on and not produce output that only works with MS-Explorer, only in 640x480, generates incorrect HTML, etc. ?
L. Bergman: We have a handful of feedback: these people really want to move around stuff and not learn a new model. We need more feedback, though.

N. Graham: What are the limits of this kind of approach? When designing for very different interfaces (eg electronic whiteboard vs palm pilot), the resulting interfaces may be completely different, not just different in choice of widgets.
L. Bergman: This approach is very much biased towards form-based approaches. This is a limitation when trying to get cross-platform design.

C. Roast: I'm interested in the user's response to a logical view and possible concrete views. There is evidence that the concrete view presists for designers. What plans do you have for your user studies?
L. Bergman: Because of the concrete bias we are interested in example driven uses of the editor. As for user studies, we are still planning.

M. Borup-Harning: Can your approach cope with situations where e.g. presenting editable information on a GUI vs. a HTML based platform might result in one form-based interface on the GUI platform, whereas the HTML based one will be divided into a presentation page with an edit button and one or more form-based pages for editing.
L. Bergman: I am not sure I understand the question? But the system was not meant to deal with arbitrarily long lists of information.

J. Williams: Have you considered providing a transition from the concrete interface view to the logical representation? This would allow developers to specify in the concrete, yet you retain reuse across devices. In addition, you can transform existing specifications.
L. Bergman: Yes, this is future work.