Toward a Design-Oriented User Interface Specification That Runs

Jacob W. Jespersen
IT University of Copenhagen
Rued Langgaards Vej 7, DK-2300 Copenhagen S
jwj@itu.dk
phone: +45 72 18 51 72 fax: +45 72 18 50 01

ABSTRACT

For the purpose of developing and conveying designs, people designing user interfaces create descriptions in the form of text, drawings, diagrams, etc. We consider these forms of expression constituents of *design-oriented* user interface specifications. In a complementary way, programmers building user interfaces create descriptions (e.g. code) that control user interface technology. We take interest in information systems and in styles of user interface descriptions that are design-oriented *and* able to control technology. This paper presents preliminary results from our research, specifically the capabilities of an experimental prototype that stages human-computer interaction by specification. We introduce a specification style that allows designers to directly craft user interface appearance and behavior at a detailed level without involving low-level programming concepts, and we discuss the rationale behind our notion of *design orientation*.

Author Keywords

Information Systems, User Interface Development, Design-Oriented Specification, Model-Based User Interface Development

ACM Classification Keywords

H.5.2. [Information Interfaces and Presentation]: User Interfaces; D.2.2. [Software Engineering]: Design Tools and Techniques

INTRODUCTION

User interface (UI) development is reckoned a multi-disciplinary effort. Among the combined skills of the two dominant groupings in the field (HCI & SE) we find psychology, ethnography, task analysis, usability, graphics design, and engineering, in no particular order.

In the information system development community that we consider, the practices of development teams increasingly blur the conventional groupings; particularly web development seems to promote this intentionally. To a large extent our research is motivated by this trend when addressing development of user interfaces to information systems in a way that we deem *design-oriented*. We believe design practitioners in the field – regardless of HCI/SE disposition – benefit from a shared and open user interface specification. Furthermore, we find it desirable to support a UI development process without prescribing a design methodology or a theory for design in the area.

In order to study the prerequisites and implications of a design-oriented user interface specification *that runs* – i.e. controls UI technology directly – we have implemented an experimental prototype. In the following we will outline the principles behind its design and present some of the results that we find encouraging.

WHAT IS MEANT BY DESIGN ORIENTATION

Faced with the task of developing a user interface, we take *design orientation* to mean:

A perspective that prioritizes the ability to amalgamate appearance and behavior using concepts from the use domain.

To exemplify this definition, consider first a paper prototype, i.e. a set of drawings that show what will appear on screen when a prospective information system runs. Paper prototyping is recognized as an efficient design-oriented technique to evaluate design ideas before writing code [1]. Usually a paper prototype session[1] mimics a possible concrete use of the future system, and showing realistic data.

Clearly, paper prototyping has a strong design orientation according to the previous definition. The designer is free to imagine any kind of presentation and interaction; it only has to come onto paper with a pen. Some intention will drive

[1] In a paper prototype session a presenter shows a prepared paper prototype to a *test person* (usually one page at a time in some sequence) in order to get feedback on the design.

the design process, and design knowledge from various sources inform it, yet the strokes on paper that form the prototype express concretely what a prospective user will experience by way of concepts from the use domain.

Now consider a set of program source code files to compile and run a user interface to the same system. Clearly, such source code will look differently dependent on the interface paradigm (e.g. WIMP or command line) and the computing platform it is written for, e.g. Java or C#. But if we focus on the conceptual level in the coded specifications (the kind of UI primitives that exist, e.g. WIMP's check-boxes, pop-ups, edit fields, etc.) there are only marginal differences within equal paradigms.

Code vs. paper

Can we sensibly compare such two separate UI specifications, one cast in paper and one cast in code? We note that both forms usually exist in a UI development process, where the paper prototype generally precedes the coded one, and for that reason may encapsulate more knowledge about the domain than was available when the paper one was done.

We find that we *can* compare code and paper in this situation – even as there are a lot of obvious differences – if we consider them responses to the question: *how is data from the system presented in a way that makes sense to people that use it and how do they interact with the system?*

It is possible to answer this question either with pen and paper (and enough time) or by writing code. Thus, in this respect we can't satisfactorily distinguish the two forms of specification by a criterion of completeness, but clearly only the coded specification is suitable for computation. A complete specification by pen and paper is neither practical nor desirable in common UI development, but it *is* possible.

We argue that traditional programming languages (which are dominating commercial UI development [2]) are not design-oriented with regards to UI development, primarily because they require the designer to deal with concepts outside the use domain, e.g. object-oriented programming in the case of Java and C#. By the same definition, paper prototyping *is* design-oriented as the designer stays comfortably within concepts from the use domain when specifying a design.

In our research agenda, crafting a UI specification style is properly a question of how to maintain computability while optimizing the design orientation.

AN EXAMPLE UI SPECIFICATION

Designers articulate data to become information to the users of their designs. We are not prescribing a certain way to do the articulation; rather we seek to provide the means through a certain style of specification and a platform able to turn the specifications into a live user interface.

To demonstrate the principle and introduce the expressive power in our style of specification, we present a simple case based on a micro CRM (Customer Relations Management) system. In essence, such system holds data on a number of customers, their contact information and records of *tickets*, i.e. the pending issues for each customer. It is beyond the scope of this paper to introduce the case detailed enough to discuss domain analysis (e.g. task and data model). Instead, we focus on the possible design space, in particular the articulation of the customer details and their respective tickets.

Design is articulation

Intuitively, every UI specification dictates how data is articulated – or articulates itself, if you prefer – in relation to the context it appears in. What differs substantially between current approaches are the means to control the articulation [3]. Due to our preference for design orientation, as we defined it earlier, we strive for a UI specification closely tied to the use domain. The basic idea in our approach is to let the UI specification bridge the gap between the dataset and the *articulation primitives* as directly as possible. In the WIMP style, which we adopt for the time being, articulation primitives are the well-known UI widgets.

In the following we introduce our specification style in an informal way by example. We do not provide tool support for creating the specifications, so they must be prepared in textual form. A simple statement that says that a Customer must appear as a Form (a.k.a. window) looks like this:

```
do Customer as Form
```

This one-liner is a quite trivial, yet complete UI specification of what should happen when a Customer (or rather the data set that describes one) appears to the user: It should show as a Form, i.e. a rectangular shape with some contents. We still need to specify how the customer details are to appear inside it, though. This is done in brackets immediately after the Form primitive:

```
do Customer as Form {
    do 'First Name' as Choice
    do Surname as Choice
    do Address as Choice
    do Phone as Choice
}
```

This specification achieves the following appearance when run in our prototype:

Figure 1. Name, address, and phone fields are editable.

Several questions are immediately pressing: how do the designer control color, placement, size? What controls the dialog? And why aren't the aforementioned tickets showing?

The last question is easily answered: The customer's tickets are not appearing because the specification does not detail them. This is fixed by inserting the last two statements here:

```
do Customer as Form {
    do 'First Name' as Choice
    do Surname as Choice
    do Address as Choice
    do Phone as Choice
    do Ticket* as List {
        do Ticket as Element
}   }
```

The above specification realizes the UI depicted in Figure 2. The asterisk appearing in the 6th statement plays a significant role that we will return to later on.

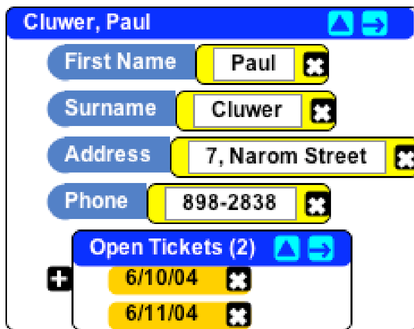

Figure 2. Information on open tickets (titled by date) now appears

The question of dialog control is also discussed later on, and there are facilities to control colors, size, and layout generally, however they are omitted here for brevity (the example shows what turns up when not exercising control). Instead, we will detail the general principle behind this kind of specification, and explain some of the facilities that go beyond plain form building.

Rules of articulation

The articulation primitives in the example so far are Form, Choice, Element, and List; several others exist. Each statement in the specification (*do* X *as* Y) comprises a *rule* that tells what primitive should be used to express some particular data. Data in the underlying data set is referred to by name; primitives have names that we have chosen, and which reflects their appearance and behavior.

Thus, a UI specification in our style consists of a set of rules that determine how to articulate data; in other words, they control the presentation and interactive features of the resulting UI. Rules nest, i.e. a rule may contain other rules surrounded by curly brackets as shown in the example above. Rules inside a rule are only in effect if the enclosing rule is. So, when is a rule in effect? To answer this question we have to look at when and how the rules are consulted.

Running the UI

The experimental prototype we have built is a *runtime environment* (RE), which in our case is merely a piece of software that is able to read a UI specification and stage a user interface accordingly. Besides a UI specification the RE also needs access to the data set to work on; how this is done and the technical architecture involved is beyond the scope here. For the examples in this paper we use a fixed (static) data set, but as you would expect the RE can work with dynamic data sets, e.g. from an application or a database.

Initially, the RE shows a blank screen, even if it has access to a data set. Only when some piece of data from the data set is *put in focus* explicitly (we'll mention how later on) does the RE consult the UI specification to learn how to articulate that data. Since every piece of data has a *type* (e.g. Paul Cluwer is a customer, thus the data about him is of the Customer type), the RE may find a rule that tells how Customer data should be articulated. If a suitable rule exists – as in the example – the named primitive (in casu 'Form') is used to articulate the Customer data. But that only draws the rectangular shape and the title of Figure 2; in order to learn how to articulate the customer details (name, address, and phone) the RE therefore consults the rules that follow in brackets. These rules must tell how the customer details are articulated. Again, each rule states that a particular articulation primitive handles a particular named type. For each type of customer detail data, the RE seeks to find a governing rule.

THE ONE AND THE MANY

We will now return to the role of the asterisk in the specification that realized Figure 2. The asterisk signifies that the rule to articulate 'Ticket' as a 'List' applies only to a *collection of* Tickets, i.e. many individual Tickets taken together. The example demonstrates a common use of this facility: A customer often has several open tickets, thus it is important that the UI designer can declare how to articulate the ticket collection, e.g. in a List, Table, or Graph.

We have found this individual/collection dichotomy to be a central theme in our UI specification. It is obviously closely related to the issue of cardinality among entities in a data set (e.g. as expressed in a UML diagram [4]), yet there does not seem to be a trivial correlation between cardinality and the appropriate articulation. In fact, our experiments seem to indicate that it is partly in the balance between articulation of the collection on its own, and its members individually, that the designer exercises control to create an attractive and usable UI. It is not sufficient to always show collections in lists with the option to see the members. A designer must be allowed to choose an articulation dependent on the context; in some cases the collection must be highly visible with a frame and scrollbars, in others it is invisible, and only the members show. In our style of specification, the designer can articulate collections as freely as individual data entities; the asterisk is used to indicate if the rule concerns the one or the many.

To show the facility in practice, we may decide to improve further on the example. Say, it is not satisfactory that we only get to see the date associated with each of the open tickets. As the underlying data set also holds data on a ticket's category (which is either *Dispute* or *Request*) and a free text, this data would fit well on the lines that show the tickets. So we change the specification. Specifically, we add rules to the articulation of the ticket collection to make each individual ticket into a Row, and further rules to have the ticket details (its category and a text) appear as Choices:

```
do Customer as Form {
    do 'First Name' as Choice
    do Surname as Choice
    do Address as Choice
    do Phone as Choice
    do Ticket* as List {
        do Ticket as Row {
            do Category as Choice
            do Text as Choice
}   }   }
```

With these changes the UI now looks as shown in Figure 3.

Figure 3. Additional information now details the open tickets

Conditional articulation rules

The last facility introduced here is the ability to let articulation depend on context, e.g. the value of some data entity, or a user's click with the mouse. It is this *conditional articulation* that allows the designer to control dialog as well as what data gets *in focus,* as previously mentioned.

As a simple example, consider the following conditionals to render different Forms dependent on customers' location:

```
do Customer where (Country = US) as Form { ... }
do Customer where (Country = CA) as Form { ... }
```

When multiple rules concern the same data type, the first matching rule – i.e. where data satisfies the conditions – is chosen for that data, and its sub-rules will determine further articulation in the manner described earlier. Note that rule evaluation takes place at runtime; for example, if a customer changes location (Country data) the articulation of that customer will immediately change accordingly.

Similarly, when the user clicks an articulation primitive, the click transfers to the data behind it, which may then change articulation due to conditional articulation rules:

```
do Ticket as Row { ... }
do Ticket where (clicked = Yes) as SelectedRow { ... }
```

Lastly, articulation may depend on aggregations and reflections on collections, e.g. in order to treat large collections in a separate way.

RELATED WORK

Our approach share traits with work done in model-based user interface development (MB-UID) [5], specifically we share the ambition of devising a specification of a user interface at a higher level than that usually seen when writing code that uses UI *toolkits* [3]. In the terminology of the MB-UID tradition our work is specification-based, provides only textual modeling tools, and does *not* automate design.

Compared to most known MB-UID approaches [6], a distinct facet in our work is that we do not generate code in order to run the UI; our UI specifications run directly in the RE. Furthermore, we strive for a unified specification, and do not use several separate specifications (models) of data, tasks, dialog, and presentation, nor do we explicate the notion of abstract and concrete articulation as in [7]. The implications of these differences are beyond the scope here.

CONCLUDING REMARKS

We have argued that what separates approaches to user interface specification are the concepts used to express how the user interface looks and behaves. Based on our definition of *design-orientation* we have outlined an approach that prioritizes this perspective when developing user interfaces. UI designers are able to express appearance and behavior based on data entities (with named types) in an underlying, possibly dynamic, data set.

In our experience a contextual rule-based UI specification aligns well with designers' practices; we look forward to validating this empirically in future work.

REFERENCES

1. Rettig, M., *Prototyping for tiny fingers.* Commun. ACM, 1994. **37**(4): p. 21--27.
2. Szekely, P., *Retrospective and Challenges for Model-Based Interface Development*, in *Computer-Aided Design of User Interfaces*. 1996, Namur University Press.
3. Myers, B.A., *User Interface Software Tools.* ACM Trans. Comput.-Hum. Interact., 1995. **2**(1): p. 64--103.
4. Booch, G., J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*. 1999, Reading, MA: Addison-Wesley Longman, Inc.
5. Szekely, P., P. Luo, and R. Neches. *Beyond interface builders: Model-based interface tools*. in *CHI '93*. 1993. Amsterdam, The Netherlands.
6. da Silva, P.P. *User interface declarative models and development environments: A survey*. in *DSV-IS2000*. 2000. Limerick, Ireland: Springer-Verlag.
7. Vanderdonckt, J., et al. *UsiXML: A user interface description language for specifying multimodal user interfaces*. in *W3C Workshop on Multimodal Interaction*. 2004. Sophia Antipolis.