ABSTRACT
Adaptation to the available screen size is a recurrent problem in graphical user interface design. On the one hand it is not trivial for a designer to specify how a Graphical User Interface (GUI) should adapt; on the other hand it is very hard to construct a predictable adaptation algorithm that never leads to undesired user interface presentations. In this paper, we present a technique to design by example user interfaces that automatically adapt to the available screen size while the designer and eventually the end user still have full control over the adaptation during runtime. This design by example technique could lead to novel tools that lower the threshold for designers as well as developers to design adaptive user interfaces.

Categories and Subject Descriptors
H5.m. Information interfaces and presentation (e.g., HCI): Miscellaneous.

Keywords
Design by example, adaptation, toolglass, design space.

1. INTRODUCTION
An increasing diversity of computing devices such as mobile devices, tablet PCs or smartboards has overtaken the consumer market. This situation has led to a renewal in HCI research about Graphical User Interfaces (GUI) that adapt automatically according to device characteristics such as CPU power, memory or available screen size [2],[5],[7]. However, automatic adaptation of User Interfaces (UI) is in many cases unpredictable and may lead to undesired user interface presentation on the one hand and behavior that no longer fits the designers vision on the other hand. Among others, the most important reason for this is the difficulty for designers to specify how a user interface should adapt according to changes in device characteristics. In turn, this makes automatic UI adaptation hard to control for a designer.

Adapting GUIs according to the available screen size is one of the most important problems in the field of UI adaptation. The reason for this is twofold. At first the screen characteristics (physical length and width, number of pixels) can differ among the existing computing devices. Secondly, even on a given computing device, users may want to resize the GUI accordingly to the importance they attribute to it or in order to display multiple GUIs. Notice that resizing a GUI may imply deep changes, such as showed in Figure 1 for the audio player WinAMP.

![Figure 1. Different modes of the same WinAMP skin.](image)

Numerous algorithms are available to manage the layout of the GUI in order to fit the screen size. These algorithms can be based on nested boxes, constraint solvers or simply scaling. However, the complexity of most of these techniques implies that they are only usable for programmers instead of designers [5]. In order to build adaptive GUIs such as the one illustrated in Figure 1, it should be possible to show different GUI elements at different screen sizes (i.e. in some cases GUI elements need to be substituted by others if the screen size changes) However, the substitution of GUI elements is not supported by one of the aforementioned approaches.

In this paper, we propose a design by example approach, similar to the one presented in Artistic Resizing [5], to allow the designer and eventually the end user to specify how the GUI should be adapted to the available screen size. Design by example, as pointed out by Frank in [6], has two main advantages. First, the designer manipulates a concrete object rather than an abstract representation. Second, providing examples is less complex than programming the corresponding algorithm, which puts the control in the hand of the designer instead of the developer.

In order to provide the example designs and to validate our approach, we built a tool allowing designers and end-users to specify example designs directly on the running UI. This way we blur the distinction between runtime and design time and thus facilitate a trial and error approach by making the effect of an example immediately perceivable.

The main contributions we present in this paper are:
- A **design by example approach** enabling designers to demonstrate how a user interface should adapt according to the available screen size by providing example UIs (Section 3).
- A **tool** to specify example UIs at runtime and to manage the inter- and extrapolation algorithm (Section 5).

### 2. Related work

Artistic resizing [5] was a source of inspiration for this work. It allows designers to build visual variants of graphical objects by using their traditional tools (e.g. Adobe Illustrator). These variants are used as examples which are interpolated and extrapolated afterwards. Given our objectives, the two main restrictions of this work are (1) it is not possible to substitute a graphical object by another; and (2) the authors oriented themselves towards providing a tool for design time. That is, the designer has to switch between tools such as Adobe Illustrator and the Artistic resizing runtime environment to view the behavior.

Others works on programming by example mostly focus on inferring the mappings between application data and its visual representation [8],[11],[13],[14],[15]. The Chimera system [9] allows to express visually a wide range of behavior. However, as pointed out by [5], it requires the users to draw a lot of examples in order to define the right behavior and can infer over-constrained or unintended rules. Collignon et al. [3] propose an intelligent editor for multipresentation UIs. This editor allows to specify several versions of the UI, each adapted to a certain screen size. To build each UI version, another editor is used (GraphiXML in this case). The selection of the right version is done at runtime, depending on the available screen size. However, the designer cannot edit the behavior of the UI when it is already in operation. Therefore, with respect to our objectives, [3] only applies at design time. Moreover, the different versions of the UI are built separately which makes it difficult to keep them consistent. On the contrary, in our approach, all examples are related to the others since they are just a different view of the same UI. Therefore, every example is semantically equivalent to the others by construction.

Gajos et al. [7] propose SUPPLE as a system for automatic adaptation of the user interface with respect to the available screen size and a user model that is constituted by traces (i.e. the actions the user has really done). While the optimization algorithm used here is generally considered as one of the best developed so far, results are still not comparable to what a human designer can produce with custom widgets. SUPPLE [7] also suffers from the classical drawbacks of automatic generation such as unexpected behaviors when resizing or using the UI.

Li et al. [10] propose a system to create UI prototypes by graphical demonstration. This system does not aim to design adaptive user interfaces but instead provides interesting insights on algorithms that could be used to construct examples. In particular, the authors demonstrate how to detect pivot points and make use of movement paths. This confirms that it is possible to specify more complex behaviors using design by example. We will explore this in the future.

Meskens et al. presented Gummy [12], a GUI builder to create UIs for multiple platforms and screen sizes. It uses a design by example approach to generate initial designs for a certain platform from a set of previously designed user interfaces. This initial design can then be refined by the designer to reach an appropriate UI for a selected target platform. Contrary to our approach, they do not target runtime adaptations but only consider design time.

Finally, the possibility for an end-user to adapt UIs at runtime by copying and pasting UI elements into “Façades” is presented in [17]. This work also demonstrates the possibility to dynamically substitute interactors of legacy UIs. The philosophy is clearly close to the one that underlies our work: giving the designer (or even end-user) the ability to fully control the adaptation process by directly manipulating the UI. However, “Façades” are quite static; it is currently not possible to resize them. Moreover, it is not possible to build different version of the UI that will be dynamically chosen while resizing.

### 3. Design by example

We define a design space as a set of user interfaces that have similar behavior and goals and support the same set of interaction tasks. Each UI in this design space is appropriate for a certain range of screen sizes. Figure 2 illustrates the design space of a UI that supports the user task of selecting a slide number from a running presentation (e.g. PowerPoint). Different UIs can be defined to support this task and populate the design space; In Figure 2, UIs of examples a and b are adapted according to the available screen width, while they are not influenced according to the height. Example c is adapted according to primarily the increase in screen height, while example d is adapted to both an increase in width and height. This example design space shows that the presentation for the same task can differ significantly: the structure, style and layout of the user interface are tailored according the screen size but the functionality that is offered remains unchanged.

![Figure 2. Illustration of a design space of UIs that supports the user task of selecting a slide.](image-url)
In our approach, the designer can create these example UI designs for different screen sizes. These UI designs are interpolated and extrapolated to generate user interfaces for all other sizes the designer did not take into account explicitly. Interpolation and extrapolation means that, given some examples for certain screen sizes, the system will propose UIs for all intermediate window sizes (interpolation) and even smaller or bigger ones (extrapolation). In this sense, interpolation can be considered as a specialization of the design space, while extrapolation extends the coverage of the design space.

The design space can be changed at design time as well as at run time which makes our approach unique. The designer can specify examples (Figure 3-1) and view the results of interpolation or extrapolation (Figure 3-2) directly on the final running UI, and manipulate the design space this way. The designer can use the UI the same way the end-user will use it and edit the design space if the behavior is not what is expected (Figure 3-3). In order to be really effective and usable by designers, this approach requires the underlying mechanism of interpolation and extrapolation to be easily understandable but yet remain powerful enough to allow an end-user to change the UI according to her or his preferences. Artistic resizing [5] demonstrates this for the graphical aspect of a user interface.

During the adaptation process presented above we aim at maintaining a semantic equivalence. Therefore, each modification of the UI should preserve the semantics (i.e. preserving the user task). These modifications encompass several operations on the UI such as moving, resizing, rotating and substituting UI elements. Because only the substitution of interactors may alter the semantics our tool should ensure that interactors can only be substituted with compatible ones (compatible with respect to the original semantics/task).

4. Example inter- and extrapolation

The algorithm we use to interpolate examples is based on the orthogonal interpolation technique used in Artistic resizing [5]. This algorithm is simple and easily understandable even by non-programmers but yet allows for interesting results. However, the algorithm focuses on the graphical representation only, and does not deal with semantical equivalence when resizing. We generalize artistic resizing by providing a common infrastructure to express dependencies between any combination of UI variables.

4.1 Definition

To compute UIs by means of interpolation and extrapolation we divided the screen size space into zones. Figure 4 illustrate how examples of the Figure 2 can be structured in two zones. Each zone defines a subspace in the screen size space, i.e. a set of points <window width, window height> for which a set of examples is defined. These examples will be used to compute user interfaces for other screen sizes through the interpolation and extrapolation mechanism whenever the window is resized within the zone boundaries (e.g. the window stays larger than (100,100) and is smaller than (200,200)).

In order to compute interpolated user interfaces, the underlying algorithm relies on the data structure presented in Figure 5. Resizing the UI induces our algorithm to look up the zone the current UI is located in. Given the zone, every reference variable (as named in Figure 5, e.g. window height or window width) that influences properties of the UI’s elements can be retrieved. These influenced properties are called related variables in Figure 5 and can be for example the position of a button, the size of a slider, etc. For each of these related variables, an interpolation function (e.g. linear interpolation) computes an appropriate value according to the new size of the UI. This interpolation function relies on a set of example. Every example contains two values: the reference value of the reference variable when the example was provided (e.g. 320 pixels for the width of the window) and the value of a UI element’s property (e.g. 100 pixels for the position of a button within the window when the example was specified) which we call a related value.

For instance, the x position of the button “Last” in Figure 4 is defined by two examples (in the lower zone) relating the window width to the x position of the button : <320, 280> and <480, 430>. These two examples are linked to a related variable (Figure
5), the x position of the button which embeds a linear interpolation/extrapolation function. This related variable is linked to a reference variable: the window width. This reference variable is linked to the lower zone, it will be used only when the window size belong to this zone.

<table>
<thead>
<tr>
<th>Zone</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subspace of screen size’s space</td>
<td>Reference value Related value</td>
</tr>
</tbody>
</table>

Figure 5. UML class diagram representing the interpolation data mechanism.

4.2 Illustration
To better understand these concepts, Figure 6 illustrates how the x position of a button can be related to the width of a window given three examples. Each example associate a x position of the button to a particular width of the window. When no example exists for a certain window width, an interpolation/extrapolation function can be used. This interpolation function is not necessarily linear (thin line in Figure 6) but could also be discontinuous (thick dashed line in Figure 6) or anything else, depending on the needs of the designer.

| Reference variable | Interpolation/extrapolation function |

Figure 6. Possible interpolated functions that are compliant with the given examples (rounds). The thin line represent a linear interpolation, the thick dashed line represent a discontinuous interpolation.

Using this technique, the designer has the possibility to construct fine grained UI behavior. This behavior is not only restricted to interactors’ position, size or rotation, but can be applied to its type, thus making it possible to substitute interactors by others.

The key point for success is how to give the designer the right tools to edit these zones and examples. This is the subject of the next section.

5. Designer Tool Support
In this section we describe the tools a designer can use to design adaptable UIs by example. Designers can specify examples and associate them to zones. She can visualize, navigate and edit the UI design space (i.e. zones, examples and the interpolation function). Since these tasks are applied on the final UI, the designer can view the result by manipulating the UI at any time. We expect that blurring this distinction between design and runtime provides designers with better insights about how the end-user can interact with the user interface. This also enables the end-user to re-design as she or he sees suited.

5.1 Toolglass metaphor
In order to facilitate a designer and/or end user to work on a running UI, we provide them with a toolglass [1]. From our point of view, the toolglass is a more convenient way to interact with a running UI than “modifier keys” or right mouse click (which could be already used by the running UI). It acts like an extra layer on top of the UI.

As depicted in Figure 7, the toolglass we currently offer is composed of two main parts: a border which allows to drag it over the final UI and a central part used to provide examples and previews. The border of the toolglass also contains some commands (implemented as “mini-toolglasses”) the designer can use for fulfilling some specific subtasks (Figure 7).

5.2 How to specify examples
Editing examples on a running UI on the fly is a typical example of manipulating the design space at run-time. The toolglass acts upon the concrete graphical representation, but modifies the design space accordingly.

Since each example contains as well a reference variable as a related variable, the toolglass specifies examples by storing both variables in our data structure (as discussed in Section 4.1). The commands located on the top of Figure 7 are used to manage the reference variables. Once the reference variables have been chosen, the designer can click on a UI element (for example a slider which is part of a slide controller, in Figure 8) through the central part of the toolglass in order to modify this element (Figure 8-1). A modifier is then placed on top of the clicked element, providing a mean for direct manipulation of the elements’ presentation and behavior. This modifier decorates the underlying interactor with new graphical elements (comparable to the handles in traditional GUI builders) that allow the underlying element to be moved, rotated and scaled by the designer. The modifier acts as a proxy for the element underneath and all modifications applied to the modifier are also applied to the slider (Figure 8-2). When the slider is positioned, rotated and scaled as the designer wants, a simple click on the modifier through the toolglass suffices to save the example. More specific, the related variables of the selected interactor (e.g. its size and/or position) are stored in the data structure together with the previously selected reference variable (e.g. screen width or rotation).
In some situations, an interactor is not directly represented in the UI. For example, the slide controller presented in Figure 8 contains a slider and buttons but the slide controller itself is not represented directly as a graphical element but as a set of graphical elements (i.e., the four buttons and the slider). Concretely, if the user wants to modify the slide controller, she can proceed with clicking on one element (either on the slider or a button) through the toolglass, making a modifier appear. Subsequently she clicks on the modifier through the commands on the left side of the toolglass (see Figure 7) until the modifier is applied to the desired slide controller (Figure 8-3).

![Figure 7. Toolglass used to specify examples.](image)

By providing some examples, the designer can specify relatively complex layouts. However, layout is not sufficient for providing a UI adapted to each screen size. For example, when one reshapes the UI shown in Figure 8 to a small square, it is impossible to reach a usable result with only a layout mechanism. There is a need to be able to substitute GUI elements by other ones.

### 5.3 Using a semantic network to retrieve new presentations

We provide a command on the toolglass that can be used to substitute an element by a semantically equivalent one. This semantic equivalent element is retrieved from a semantic network as presented by Demeure et al. [4]. This semantic network can be compared as a kind of “UML diagram” based on concepts identified in [2] (abstract UI, concretization...). This semantic network can be queried at run-time, just as a service.

Figure 9 illustrates the part of the semantic network which is related to the slide controller. The nodes contain models of the slide controllers. Arks represent the links between these nodes and indicate the ongoing relationships between them such as an extension, specialization, concretization, etc. Leaf nodes correspond to several alternative implementations of the slide controller presentations. These implementations are the ones that the designer can retrieve automatically by using the toolglass in combination with the semantic network.
When the designer clicks on an interactor through the toolglass (Figure 10-1, the interactor corresponds to the “SC Generic PM” in Figure 9), the clicked interactor and its ancestors are displayed in a list (Figure 10-3). When clicking on an item of this interactors list, a query is automatically send to the semantic network, which acts as an external service, to retrieve all semantically equivalent interactors for the interactor type that is being modified. That is, all interactors supporting the same user task for which the current interactor is being used. The result of this query is currently displayed as a list of equivalent interactors, ordered according to their proximity (with respect to the semantic network) they have with the interactor being modified (Figure 10-3). When the designer clicks on an element of this list, the interactor is substituted by the interactor represented by this element. For instance, the slide controller composed of Figure 8 can be substituted by a custom one (Figure 10-4, the interactor corresponds to the “SC Skinable OpenGL” node in Figure 9). Instead of a list of interactor descriptions that can be hard to read for a designer, the same list could also be presented as a set of graphical previews of alternatives.

**5.4 How to navigate and edit design space**

As shown in , the design space is represented as a 2D graph where the vertical axis indicates the height of the UI (which is usually a window) while the horizontal axis represents its width. This graph contains zones. Each zone is associated to a set of examples. These examples are materialized by points and do not necessarily belong to the zone. When the designer clicks on a zone, the corresponding examples appear in the graph.
The size of the current user interface is represented by a special “cross hair”. Each time the designer resizes the user interface, the dot is updated to the new size. On the other hand, if a user drags the cursor to another place in the example space, the user interface is recalculated for this size. Thus, both the 2D example space and the current user interface are always synchronized. At any time, the users can define new examples (Figure 12 steps 2 and 3) and move into the design space by resizing the window to see the results of the examples (Figure 12 step 3).

While the users navigates through the example space, they can interactively edit the UI, for instance to substitute elements by some others (Figure 12-4). The users can add new zones where new sets of examples apply (Figure 12 steps 5 and 6). Examples can also be modified or deleted. Each time the user clicks on such a dot, the example that was given for that space will be visualized.

6. Discussion
As pointed out by Frank in [6], there are some well-known drawbacks of using example-based user interface design. First, there are theoretical limits: one can not specify all possible behaviors. For instance, in our case it is not possible to specify flow-layout or treemaps. These can be partially overcome by making the designer able to switch between layout functions in the same way as she can switch between interactors types (e.g. using a flow layout for wide screens and treemaps for small screens). The semantic network could be used to retrieve such layout functions, as it is to retrieve semantically equivalent interactors.

The second drawback discussed by Frank [6] is that one cannot have complete confidence in what has been inferred without inspecting a static representation of the behavior. Again, we agree with that. In order to alleviate this drawback, we provided the designer with an interactive representation of the design space which can be considered as a kind of “debugger” that helps the designer to reach the desired result.

The third and the last important issue is that demonstrating all of the behavior for a large design can be frustratingly tedious. In particular, the designer cannot easily define a common behavior once and then parameterize it for future reuse. We haven’t addressed this issue so far, but we certainly have plans to explore different solutions in the future. For instance, we plan to give designers the possibility to (1) define template behaviors that they could apply to objects and (2) use constraints when defining examples (e.g. specify four buttons to have the same size…).

7. Conclusion
In this paper, we presented a design by example approach to create adaptive user interfaces. Using this approach, the designer can achieve adaptive user interfaces of high aesthetic quality while keeping full control over the design process. This design process consists of three main steps: providing examples, editing the examples in the design space and viewing the behavior of the adaptive user interface immediately and continuously. We provided some tools based on the toolglass metaphor to support this approach directly on the running user interface. This blurs the distinction between design time and runtime, which encourages trial and error and thus lowers the threshold for developing adaptive user interfaces by example.

In future work, we will explore the possibility to dynamically switch – depending on the available screen space – between complex layout algorithms, such as treemaps, flow layout, etc. This would only require small modifications to the toolglass. In addition, we will explore ways to give the designer the possibility to generalize the behavior of an element to a set of elements. We will also explore ways to trigger graphical transitions when changing zone. Those transitions should help the user to understand how the UI was reconfigured (e.g. by using morphing). Finally, we will conduct evaluations in cooperation with designers.
Videos of the system can be found at: http://iihm.imag.fr/demeure

8. REFERENCES


