Authoring Support for Post-WIMP Applications

Katharina Gerken[1], Sven Frechenhäuser[1], Ralf Dörner[1], and Johannes Luderschmidt[1]

[1] RheinMain University of Applied Sciences, Wiesbaden, Germany

Katharina.B.Gerken@student.hs-rm.de, {Sven.Frechenhaeuser, Ralf.Doerner, Johannes.Luderschmidt}@hs-rm.de

Abstract. Employing post-WIMP interfaces, i.e. user interfaces going beyond the traditional WIMP (Windows, Icons, Menu, Pointer) paradigm, often implies a more complex authoring process for applications. We present a novel authoring method and a corresponding tool that aims to enable developers to cope with the added level of complexity. Regarding the development as a process conducted on different layers, we introduce a specific layer for post-WIMP in addition to layers addressing implementation or traditional GUI elements. We discuss the concept of cross layer authoring that supports different author groups in the collaborative creation of post-WIMP applications permitting them working independently on their respective layer and contributing their specific skills. The concept comprises interactive visualization techniques that highlight connections between code, GUI and post-WIMP functionality. It allows for graphical inspection while transitioning smoothly between layers. A cross layer authoring tool has been implemented and was well received by UI developers during evaluation.

Keywords: authoring processes; authoring tools; post-WIMP interfaces; cross layer authoring; collaborative user interface development; combined post-WIMP interactions; visual validation

1 Introduction

Authoring processes and tools for creating classical GUIs (Graphical User Interfaces) that adhere to the WIMP (Windows, Icons, Menus, Pointer) paradigm have been explored and refined over the years and developers have been honing their skills in creating these types of user interfaces. With post-WIMP [14] becoming more commonplace [11], the authoring process for user interfaces is becoming more complex, as the diversity of interaction methods rises [1, 2]. Novel tasks need to be carried out during authoring and new skill sets are required, for example expertise in reality-based interaction [5], in order to develop post-WIMP user interfaces. As single authors seldom possess the whole range of skills required, collaboration of multiple author groups with specific expert knowledge is often needed [4, 12]. We identified another factor that distinguishes authoring of post-WIMP user interfaces: many post-WIMP interface elements (e.g. gestures, speech input) have no graphical representation in the user interface and therefore make it more difficult to graphically represent them in author-

ing tools. In addition, authors may not understand from visually inspecting the user interface which interactions are implemented and how they work together. As a consequence, existing tools for GUI development may not be well suited for post-WIMP. Dedicated authoring tools are necessary to create these types of user interfaces efficiently, to ensure high quality, to facilitate meaningful inspection and tests, to enable authors to contribute their expertise to the development and to reduce overhead due to several authors having to cooperate in the development.

This paper presents an authoring process and an authoring tool for post-WIMP user interfaces that take the peculiarities of post-WIMP methods into account and foster collaboration between different author groups. In our concept, we distinguish between four author roles: the programmer writing code, the GUI designer building the application screens, the post-WIMP designer who adds further input channels to the screens and the client who wishes to customize the application. In contrast to using several different tools for the respective development tasks, we expect the authors to better understand the connections between code, GUI and post-WIMP interaction when working with only one tool [17]. This may lead to a more comprehensible development process. It also supports the refactoring and iterative refinement of applications. We call this cross layer authoring, as we regard the development of a post-WIMP application as a process that is conducted on several layers denoting the contributions of the distinct authors. Each layer corresponds to a different author role with different task areas. The authors should be able to work on their respective layers independently, although certain dependencies do exist: The code layer can be regarded as the fundamental contribution to the application being developed. On top of that, the GUI designer can build the application's appearance, which can then be enhanced with interaction by the post-WIMP designer. Each layer offers abstract connection hooks for the other layers, allowing the authors to be able to work on their particular layer only. Thus we expect them to be able to focus and immerse themselves in their respective task better, which might lead to an increase in quality of the resulting applications. In order to inspect the connections between layers, specific interactive visualization techniques are introduced. Transitions between layers are designed to be seamless in order to reduce the loss of orientation when switching layers. In our approach, we aim at preserving traditional GUI authoring as many authors are familiar with this and have already acquired a profound experience. Thus, post-WIMP is added as additional layer and is separated from developing the WIMP aspects of a user interface. Our paper offers the following contributions:

- We introduce the concept of a single cross layer authoring dedicated for the development of post-WIMP user interfaces.
- We identify four layers as one authoring dimension and four editing modes as an orthogonal dimension resulting in a 4 x 4 state matrix of the authoring tool.
- We conceive visualizations for interactive transitions between layers and for the creation and inspection of post-WIMP interactions in a cross layer authoring tool.
- We present a prototype implementation demonstrating the feasibility of our concept.

The paper is organized as follows: First, related work is presented to illustrate the state of the art in authoring for post-WIMP applications. Then our cross layer authoring concept and CLAY (Cross Layer Authoring Tool) are being introduced and illustrated with a small use case. After explaining the basic navigation elements of the tool, we focus on the creation and configuration of post-WIMP functionality. Finally, we discuss our concept and the CLAY editor by evaluating user feedback to our prototype and stimulating possible future research topics.

2 Related Work

There exist several frameworks and libraries which enable programmers to add post-WIMP features to existing applications or to develop applications with post-WIMP user interfaces, e.g. PyMT [3] for touch or the Kinect SDK[1] for full body interaction. Those frameworks provide powerful tools for programmers, but are less fitted for interface designers [8, 9]. In the following, we focus on concepts assisting programmers on the one hand and supporting visual development (for the GUI designer, post-WIMP designer or client) on the other hand.

One approach to handle the complexity of the development of post-WIMP applications is to enhance a classical WIMP UI by merging post-WIMP capabilities on the same abstraction layer, as can be found in Apple's authoring tool named Xcode[2], for instance. With Xcode, the WIMP UI can be created graphically in a GUI editor. The UI elements are then bound to UI events by dragging graphical connection lines to methods or properties specifically marked in the source code. However, there is no visual support to add post-WIMP input (e.g. custom gestures) to the UI, so it can hardly be achieved without fundamental knowledge of the required programming techniques. This might lead to the programmer and designer having to spend more time negotiating specifications, which may add a significant overhead and prohibits focus on each author's individual task. Also, Xcode's concept does not include support for testing gestures directly while creating them. To inspect the created behavior, the use of a simulator or device is required. This makes the authoring process of post-WIMP capabilities inefficient, because the entire resulting application must be compiled and executed to test the behavior every time the logic or the parameters of the specific gesture change.

While Xcode's approach focuses on multi-touch interactions, there are other concepts of authoring tools that support the development of applications featuring all kinds of post-WIMP interactions. One of them is Squidy, a zoomable design environment for natural user interfaces [10]. It employs other concepts like the MaggLite authoring toolkit for post-WIMP interfaces [4] or the OpenInterface Framework [13], which offer a way to create post-WIMP user interfaces with the concept of visual dataflow programming. Squidy addresses the issue that the authors involved in the authoring process of post-WIMP UIs often use multiple toolkits or frameworks to create the desired UI and its behavior. This is achieved by tying together relevant

[1] Microsoft Kinect for Windows: http://www.microsoft.com/en-us/kinectforwindows/
[2] Apple Developer: https://developer.apple.com/

frameworks and toolkits in a common library, while a visual language is introduced to create natural user interfaces. To see the current development status as a whole, Squidy introduces the concept of semantic zooming, making it possible to control the level of complexity shown to the particular author developing the application. However, Squidy has a focus on providing a way to take input data of hardware devices (e.g. movement data recorded by a Kinect), process the data and send it to listening applications, e.g. via TUIO [6]. This focus on technical aspects is important to develop post-WIMP input alternatives actually available for a specific application. Still Squidy lacks the possibility to design the user interface and to add the created post-WIMP input methods to it. In addition, the capabilities of adding and validating the post-WIMP input on the actual application UI are missing. Thus, designers without any technical knowledge about the specific implementation of the post-WIMP input may have difficulties using the technical base and design as well as validating the UI as a whole.

Hartmann et al. present a rapid prototyping tool called Exemplar [15] that allows for authoring of sensor-based interactions by demonstration, thus enabling short iterative demonstrate-edit-review cycles. Yet, it only covers one step in the application development process, forcing the author to switch between different tools, whereas we seek to embed interactive editing (particularly of post-WIMP behavior) seamlessly into the overall authoring process.

Juxtapose [16], a prototyping code editor and runtime environment for UIs, allows for creating multiple alternatives of application logic and real-time tuning of application parameters. Juxtapose facilitates the work of UI developers, yet it only addresses authors with both UI design and programming knowledge, as the parameters are directly extracted from the code. The CLAY concept shares the goal to enable rapid parameter testing, however, it strives to provide it on a more abstract level for UI designers who do not necessarily know how each parameter is implemented in the program code. Generally, the Juxtapose concept lacks the distinction between different author roles.

Simpson and Terry [17] point out the problem of the use of heterogeneous toolsets for application development and present GUIIO, a design tool which uses ASCII text for rendering UI mock-ups. While it offers a fluid transition from program code to UI, it does not offer possibilities to inspect the UI in the way it would actually present itself to the user, nor does it provide means to graphically represent post-WIMP interaction. Instead of just visualizing the connection between code and UI from the perspective of the coder, we introduce a concept comprising the points of view of all the authors involved in the process, allowing them to work together with only one tool.

DENIM [18] is an informal website design tool presented by Newman et al. in 2003. It allows for informal sketch-based prototyping of websites and offers different representations of the sites through zooming. Similarly to CLAY, DENIM uses a slider with eleven levels to navigate from site map to storyboard and individual page view, thus enabling an almost smooth transition. Yet, the zooming in DENIM is only used to reveal more or less details of only one field of activity (the graphical user interface design), whereas CLAY uses seamless transitions to navigate between views of entirely different natures (user view, post-WIMP functionality, classic GUI and

code). So instead of zooming "horizontally" on one aspect of an application, the CLAY concept allows for a "vertical" zoom through the different areas of expertise of the authors involved.

To sum up, authoring tools with a focus on a specific platform provide visual means to develop the WIMP part of the UI. Yet the development of post-WIMP capabilities has to be done entirely in the source code by programmers with specific technical knowledge. Besides, the resulting behavior is hard to inspect without compiling the application and testing on the target device [7]. More general approaches which support adding all kinds of post-WIMP capabilities to applications often have a focus on technical aspects. They lack the support for designing the application's UI, integrating post-WIMP capabilities and visually testing the added post-WIMP interactions. Different authoring roles are not differentiated enough to allow authors to concentrate on their specific contribution to the post-WIMP application.

3 Cross Layer Authoring

We distinguish between four different author roles in the process of a post-WIMP application development: the coder, the GUI designer, the post-WIMP designer and the user. We regard the authors' contributions to the application being developed as layers placed above one another, with the code layer being at the bottom. Above that, there are the GUI layer, the post-WIMP layer and the user layer. The code layer can be regarded as the fundament of the application, which all other layers depend upon. Likewise, the post-WIMP layer builds upon the interface layout and traditional elements (such as buttons, text fields) created in the GUI layer. Each layer is designed to suit the tasks and skills of the respective author only, e.g. source code can only be edited in the code layer, graphical interface design only takes place in the GUI layer.

Apart from distinguishing between author roles, we further split up the task areas of each of these roles into four editor modes, namely content, layout, parameters and behavior, which indicate what kind of changes can currently be made to the application. Dividing all the editable attributes into four different modes aims at keeping the screen organized and minimizing the risk for accidental changes (e.g. changing the position of a text label when intending to change its content). An overview of the resulting sixteen development modes and the respective functionality available in each mode can be found in Figure 1.

During the authoring process, cross layer connections between GUI components, post-WIMP input methods and the application logic on the code layer have to be created. By visualizing the connections between the layers when navigating between them, information about how the layers work together can be provided to each author. Transitions between the author layers are designed to suit the authoring task at hand and to happen seamlessly in order to avoid disorientation due to abrupt changes in the UI of the authoring tool.

In order to be able to evaluate this concept, we implemented the CLAY editor. CLAY is realized as a classic desktop application as from our observation many developers prefer a PC-based environment with multiple screens. Development espe-

cially on smartphones with small screens does not seem to be practical. Nevertheless, the concept could be adapted to work on specific post-WIMP devices as well.

	Content	***Layout***	***Parameters***	***Behavior***
User	Edit text fields, change images etc.	Arrange images, texts, buttons etc. on the screens	Adjust fonts, colors etc.	Inspect application behavior by moving the cursor over UI elements
Post-WIMP	Simulate/test post-WIMP functionality	Compose post-WIMP actions, connect them to the GUI	Adjust e.g. speed for a double tap, length of a pan gesture, voice input parameters	Connect code actions or screen transitions to post-WIMP input
GUI	Set label texts, images etc.	Arrange screens and GUI elements on the screens	Adjust size, color, appearance of GUI elements	Connect code actions or screen transitions to WIMP input
Code	Display and edit available code files	Create and edit UI components to be used by the GUI and post-WIMP layers	Define access rights for component properties for the GUI and post-WIMP layer	Create actions to be used by the GUI and post-WIMP layer

Fig. 1. Overview of the different development modes of CLAY and their respective functionality: We distinguish between four author roles (leftmost column) and four editor modes (top row)

3.1 The CLAY Prototype

In the CLAY user interface, we place graphical representations of the previously described layers and editor modes at the borders of the authoring tool (see Figure 2). They are arranged as two orthogonal axes forming a two-dimensional matrix where one dimension serves to distinguish between the authoring layers and the other dimension is used to switch the editor mode. With this, we provide a simple mental model how the authoring functionality in the tool is structured. There is constant visual feedback in which state the authoring tool is set.

The slider on the left-hand side allows for a smooth transition between author role layers. When the slider is positioned directly over a layer icon, only the respective layer is visible at 100 percent opacity. When the slider is moved towards a different layer, the current layer becomes more and more translucent to show information from the layer the slider is being moved to. This blending should eliminate the need for entirely switching between layers in many cases. If, for instance, a button is connected to a code action, selecting the button on the GUI layer and moving the slider towards the code layer will result in the connected method shining through the GUI layer. When approaching the code layer, the semantic level of detail of the code information increases until the slider has reached the code layer icon, where the whole

code file is shown in a text editor (see Figure 3). In contrast to a navigation where layers are discretely switched, we expect the slider to provide a better orientation and understanding of connections between layers.

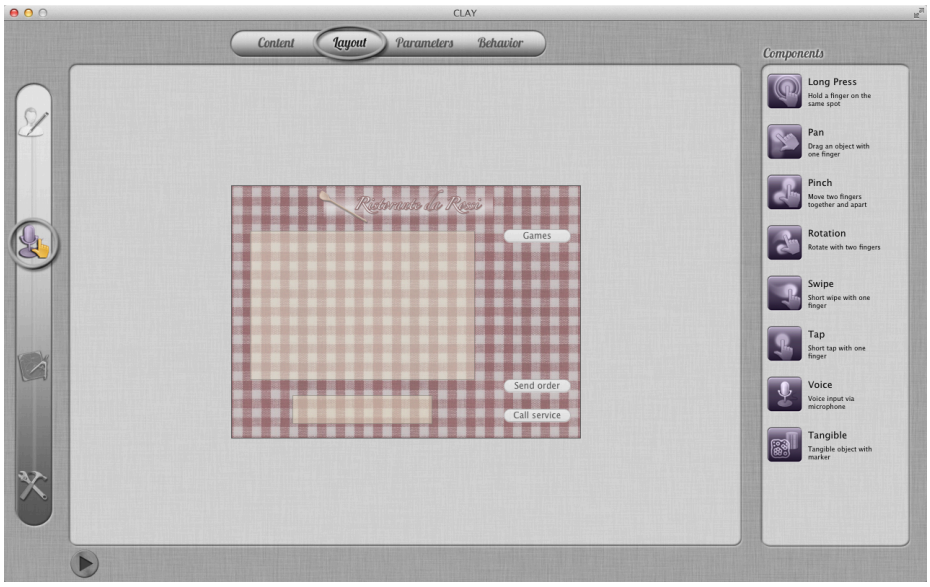

Fig. 2. The CLAY interface: A slider on the left allows for navigation through the author layers, a slider at the top can be used for switching editor modes.

Furthermore, the interface is divided up into two frames: a main frame in the center and a smaller frame on the right. The main frame serves to type code or put together GUI or post-WIMP elements, depending on the active layer. The right frame on the other hand is used to display the elements or options available in the current layer and editor mode. On the uppermost layer, the user layer, there is no right frame. Instead, the main frame is made wider to make more room for the screens of the application being inspected.

On the horizontal axis at the top, the developer can switch between the four editor modes content, layout, parameters and behavior. When switching between editor modes, the right frame's content adapts accordingly to provide a list of the specific components or parameters needed.

The play button in the bottom left can be accessed from all layers at any time. Pressing the button results in the application being compiled and run in a simulator in full screen mode. With the target device connected, this button also allows for a test run on the specific device. Though the resulting application can always be inspected in the user layer, it is important to provide a simulator where no other graphical elements of the authoring tool itself are visible in order to get a good understanding for the resulting application. It is even more important to provide this functionality as CLAY runs on desktop computers where post-WIMP input can often only be approximated by WIMP input.

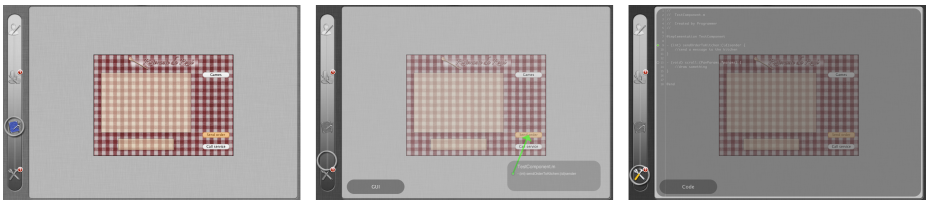

Fig. 3. Three phases of the smooth transition from the GUI to the code layer: As the slider on the left is moved from the GUI to the code layer, the interface builder becomes translucent and reveals more and more code layer details.

3.2 The CLAY Authoring Process

Throughout the next sections we illustrate the CLAY authoring process employing the following use case: Restaurant owner R has furnished his restaurant with multi-touch tables and needs a tabletop software which should allow his guests to skim through the food menu, place orders and play games while they are waiting for their meals.

There are three authors involved in the development process: Coder C is responsible for the actual functionality of the program developed. GUI designer G is responsible for the selection and layout of classical WIMP user interface elements and the transitions between screens. Post-WIMP designer P works on the screens provided by the GUI designer and adds post-WIMP functionality to the application. Restaurant owner R might be involved in the process at times, and may customize the product to a restricted extent.

Our tool provides views for each of these four roles. The view of the end user, a guest in the restaurant, is included in form of a simulator. In the following section, we elaborate in the challenges of each of the authors, introduce their layers and the associated editor modes in detail and motivate the need for smooth transitions between layers.

Code Layer. The programmers need a text editor with syntax highlighting and refactoring options in order to implement the custom GUI and post-WIMP component classes and the methods called by them. In addition to writing code, they have to provide hooks for the upper layers to enable them to connect to their methods and use their classes without intruding into the source code.

In our concept, these hooks are called components and action elements, which are being created by the coder as abstract representations for classes and methods. They create a level of abstraction for the upper layers and allow for exchangeability on the code layer without the GUI and post-WIMP layers having to perform any changes. These abstract elements also allow for simultaneous development across the layers, as the actual implementation of e.g. a specific method does not need to be completed in order for the upper layers to be able to connect to it.

In our example, coder C creates a new file in the content mode. After implementing a method, e.g. for sending an order to the kitchen, C switches to the behavior mode where she can create and edit action elements which are displayed in the right

frame. She has to provide a name and description for the action (e.g. “Send order to kitchen”), which will be displayed in the GUI and post-WIMP layers. Additionally, she can restrict the action to be only available to certain components like a button or a tap. In doing so, C would prevent G or P from inappropriate connections, e.g. the event created by interaction with a tangible object to a specific action. Wiring up the action with the code can be done graphically by dragging a line from the newly created action element in the right frame to the small circle next to the code. Circles are being displayed next to the starting point of classes and methods. An unfilled circle indicates that the method is not yet connected to an action element. After connecting, the circle displays a small green filled circle inside (see Figure 4).

Fig. 4. Detail of the code layer: Code methods are tagged with a small circle. If connected to an action element, which makes them available to the GUI and post-WIMP layer, they also display a filled green circle within.

Creating GUI and post-WIMP components works similarly. In order to create e.g. a custom button, C switches to the layout mode where existing component elements are listed in the right frame. Along with giving a name and description, property access rights can be granted to the GUI and post-WIMP designer for every component and parameter individually in the parameters mode. For instance, the color and size of the button can be made editable, while the label can remain fixed. These access rights are not enforced in the tool, they serve rather as a reminder that an author did not deem it sensible to make certain alterations. Thus, by changing layers an author is able to override restrictions.

GUI Layer. Graphical interface designers do not always have broad programming skills and may wish to focus on the design and layout of the screens. It is their job to put together standard components (screens, buttons, labels etc.) or specific custom components created in the code layer. Furthermore, certain code actions or screen transitions can be connected to them. As the GUI designers are usually not aware of the source code, they need an abstraction of the methods available to connect to.

We conceived the GUI layer to resemble already existing authoring tools where the creation of the application screens is being done graphically via drag and drop. In order to connect certain components to code methods, the action components come into play. They can be connected to the WIMP components via drag and drop as well, eliminating the need for the GUI designer to modify any of the source code.

In our scenario, graphical interface designer G creates a screen with two boxes and several buttons by dragging these components from the right frame to the canvas in the layout mode. The components can be rearranged and edited according to the active editor mode selected on the top slider: To change text labels or images, the author has to switch to content mode. Changes in size, shape or color can be performed in the parameters mode, according to the access rights granted to the components in the code layer. Actions and transitions between screens can be attached to components like buttons or table cells in the behavior mode. To connect the "Send order" method to a button, G drags a line from the associated action element on the right. When hovering over the elements on the canvas, those components potentially able to connect to an action are highlighted by rendering them in green color. Alternatively, a component can be selected by clicking first, then the right frame will only display the methods available for the specific component.

The transition from GUI layer to code layer is designed to allow for inspecting the connections from the user interface to the code. The amount of the information shown from the code layer increases as the user moves towards that layer. We illustrate this transition with our use case in the following.

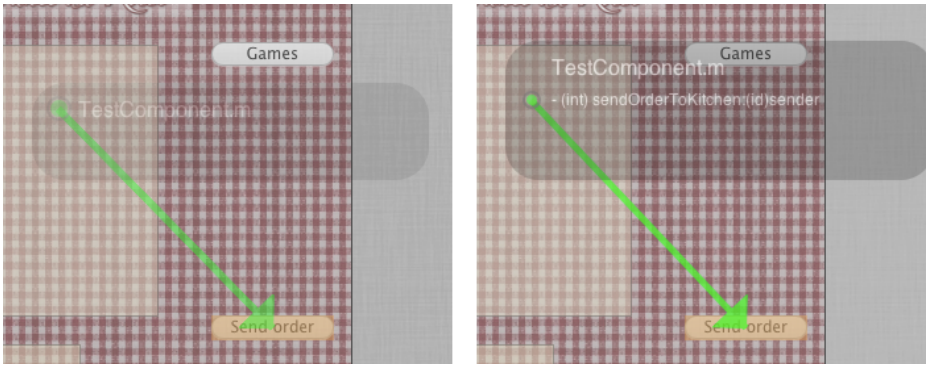

Fig. 5. Two phases of the transition from GUI layer to code layer: A small pop-up first displays the source file only (left image), then also the method the selected GUI button is connected to (right image).

As soon as G selects the mentioned button, a small number appears on the code icon on the slider to indicate how many connections to the code the button has, in this case a "1" for the connected "Send order" method. Thus G can instantly see whether or not there is code layer information attached for further inspection. When moving the slider towards the code layer, a small pop-up appears next to the button showing the name of the source file containing the connected method. When moving further to the code layer, the pop-up displays the method name (see Figure 5), and in a next step the method source code. When the slider has reached the code layer, the complete source file containing the "Send order" method is being displayed.

Post-WIMP Layer. Post-WIMP designers complement the screens designed by the GUI designers with post-WIMP functionality like touch gestures, voice input or tan-

gible object support. Such designers might be experts in usability, but maybe not know much about coding. We provide a visual authoring tool similar to the tools already available for graphically developing WIMP interfaces, where designers simply drag and drop post-WIMP elements, combine them or adjust their parameters. However, while classic WIMP UIs need to cope with visible input and output channels only (buttons, text fields etc.), post-WIMP interactions do not always have an instant graphic representation. While a button is visible and known to be clickable, the area for e.g. a swipe gesture does not need to have a perceivable boundary. Without a graphical representation, it is hard to detect which post-WIMP interaction is offered by an application.

Our goal was to create a post-WIMP layer that closely follows the concept of the GUI layer, i.e. it enables the author to graphically create input alternatives and connect them to code actions without the need to change the source code. In order to achieve this, we conceived possibilities to graphically assign post-WIMP functionality to application screen elements. The process of creating and inspecting post-WIMP elements will be further elaborated in the section "Authoring of Post-WIMP Interactions".

User Layer. Developers may want to take a look at the application screens they are currently working on without distracting connection symbols or tooltips and without having to compile the application first. Methods for user-centered design call for customers to be able to influence the development process from time to time without having to work with complex tools and risking to damage anything. That is why we included the user layer where screens are displayed the way they would look in the application when run on the target device. Here, editing is restricted to changes of the appearance or the content like exchanging texts or images, changing of colors and fonts, or rearranging of the components present. Thus, this layer may not only help developers during the authoring process, but also serves as a presentation mode for meetings with the clients.

In CLAY, the user layer consists of a big canvas showing the application's screens. When in behavior mode, the application can be inspected by moving the mouse over the screens and their elements. Whenever actions are connected to a component, they will be represented graphically. In our example, restaurant owner R may, for instance, adjust the colors to match his restaurant's interior in the parameters mode or change the texts of the food menu in the content mode. The goal is to include the customers in an iterative development process, enabling them to contribute their feedback more directly and allowing them to make changes to the application without the need to have to rely on the other authors.

3.3 Authoring of Post-WIMP Interactions

Standard authoring tools today are not designed for graphical authoring of post-WIMP interaction, but instead require programming knowledge to enhance their applications with post-WIMP features (see section 2). Our concept of cross layer authoring strives toward enabling the author of each layer to work independently, e.g. a

post-WIMP designer does not necessarily need to perform changes on the code layer. In this section, we illustrate our approach for graphically creating, combining and validating post-WIMP elements in this section by exemplarily describing some use cases of our previously introduced scenario.

Creating Post-WIMP Interactions. After the graphical UI of the application has been designed and developed, it is the task of post-WIMP designer P to enhance the interface with post-WIMP functionality. The application screen shows a box where a list of dishes is being displayed and a smaller box underneath. P wants to enable the user to scroll through the food menu by panning over the smaller box from left to right. In the layout mode, he drags a pan gesture onto the canvas resulting in a small translucent gray box containing the respective element symbol. As further configuration of the element is necessary, it is marked by a small yellow warning sign.

In order to connect a post-WIMP interaction to an application screen, the author has to define where the interaction can be conducted and which area, if any, is being affected by this action. For this, we conceived a visualization of the activity and output areas that can be defined for every post-WIMP interaction. Activity area in this context denotes the region where e.g. a touch gesture is being executed, while the optional output area denotes the region that responds to that gesture. After defining these areas, they are graphically highlighted to make it easier for the author to inspect the post-WIMP elements they created. Post-WIMP interactions do not necessarily affect elements of the user interface, they might also affect the application logic. In this case, a dedicated area is provided in the authoring tool that serves as a representation of the output area.

In our example, post-WIMP designer P wants the user to pan inside the box at the bottom, so P assigns the pan's activity area to that box by dragging from the pan symbol to the box while holding down the left mouse key. While dragging, a red arrow is being displayed pointing away from the post-WIMP element towards the cursor. After connecting, the bottom box shows a red overlay with a preview of the pan gesture, i.e. a small white circle, indicating a finger, moving up (see Figure 6).

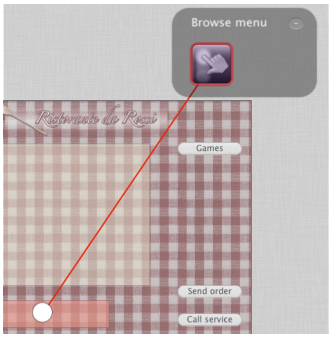

Fig. 6. Detail of the post-WIMP layer: A newly created pan gesture is being displayed in a small box. After connecting the pan to the area where it can be executed ("activity area"), the respective region shows a red overlay. A red frame around the pan symbol indicates that it has been connected to an activity area.

The pan gesture created by P provides default values containing the direction, the length and the duration of the pan, which were defined in the code layer. As coder C has made these values changeable, P can adjust them to his needs in the parameters mode, e.g. change the angle to make it a horizontal pan instead of a vertical one. The changes are instantly being reflected in the preview on the activity area.

Next, P defines the output area for the pan. In this example, he wants the box at the top to be scrolled whenever the user carries out a pan in the lower box. Assigning the output area works similarly to defining the activity area, except for now, P has to press the right mouse key and the displayed arrow and the accentuation of the components are painted blue. After connecting, the upper box has a blue overlay. A bicolored frame around the post-WIMP symbol indicates that it is connected to both an activity and an output area (see Figure 7).

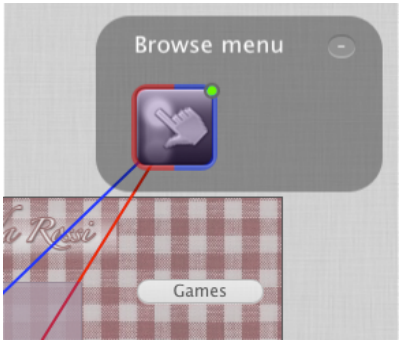

Fig. 7. Detail of the post-WIMP layer: The bicolored frame around the pan gesture symbol indicates that its activity and output areas are assigned. The green filled circle shows it is connected to a code method.

Connecting post-WIMP elements to code actions can be done in the same manner as in the GUI layer. When dragging the scrolling action to the newly created post-WIMP element, it is highlighted green. After successfully connecting the action, a small green filled circle is displayed in the upper right corner of the pan symbol (see Figure 7). As scrolling is a standard action in the tool, it also has a graphical representation, which is immediately visible on the output area overlay. Parameters like scrolling direction or speed can be set in the parameters mode along the lines of the configuration of the pan gesture.

Combined and Alternative Post-WIMP Elements. In addition to adding single post-WIMP elements to application screens, it is also possible to graphically combine several elements to one larger element group or to offer alternative input methods. This is a novel approach to deal with complex post-WIMP input, as it can be connected to a code action without the code layer having to know what exactly triggered the action.

In our next use case, P wants to add two alternative possibilities to order a meal on the food menu screen: the first is to perform a long press on the desired dish and say “Order” at the same time, the second is to perform a long press on the designated food

menu item and then swipe to the right. To achieve this, P drags a long press gesture to the canvas, and then adds a voice input element by dropping it onto the long press box. Both elements are now arranged next to one another, indicating that they have to be performed consecutively. After defining the activity areas, configuring the elements in the parameters mode (e.g. setting the voice input to the word "Order") and wiring up the post-WIMP group with the corresponding code action, the setup of the first input alternative is completed.

To implement the second input method, P adds a swipe gesture to the group, which should be performed alternatively to the voice input. To achieve this, P drops the swipe element directly onto the voice input symbol, which results in the swipe being displayed above the voice input. This indicates that either both actions have to be performed at the same time or alternatively. The behavior for these kinds of stacked elements can be switched by clicking on the label appearing above the elements, reading either "Together" or "Choice" (see Figure 8). Further configuration concerning e.g. the time interval between the long press and the swipe or voice input can be configured in the parameters mode. No changes have to be made in the behavior mode as the post-WIMP group has already been connected to a code action that will be triggered whenever one of the alternative chains of post-WIMP interactions has been performed.

Fig. 8. Detail of the post-WIMP layer: More complex post-WIMP interactions can be created by combining single gestures or events. In this case, a long press combined with either a swipe or voice input form a combined post-WIMP element, which can be connected to an action.

3.4 Visual Validation

In order to be able to inspect the post-WIMP interactions, which have been added to an application, we conceived what we call visual validation. It is meant to visualize the sequence of a specific post-WIMP interaction (or group) with their respective activities and output areas. This enables the author to test the course of the post-WIMP interactions without having to compile the application first, thus targeting on a reduction of the time needed for testing and adjusting the interaction parameters [7].

In our CLAY prototype, visual validation can be performed in the content mode. When switching to that mode, all post-WIMP boxes turn into little players with a timeline and a play button (see Figure 9). Pressing play starts a real-time animation of

the post-WIMP elements on the screen they are connected to. Touch gestures are once more previewed by a graphical representation of the user's finger moving over the activity area, standard output elements are being represented by e.g. an arrow moving along the scroll direction. This is especially useful when several elements have been combined to one more complex element group, as their interplay can be inspected chronologically. In case a post-WIMP element directly triggers a code action, this is shown by an arrow directed towards an application logic icon, which then appears in the bottom right corner of the canvas. Having these players in the content mode aims at making it easy to inspect the temporal sequence of the post-WIMP elements and check the result at a glance. Additionally, the players can be organized in different groups, which should lead to more clarity on the screen. These groups can be added, shown and hidden in the right frame when in content mode.

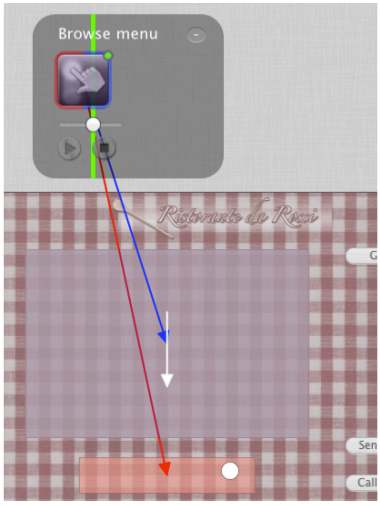

Fig. 9. On the post-WIMP layer, visual validation of e.g. a pan gesture can be performed in the content mode. Pressing the play button in the "Browse menu" box starts a real-time animation of the post-WIMP elements on the screen they are connected to.

4 User Feedback and Discussion

We evaluated our concept by presenting the prototype to three computer scientists and two media designer to gather qualitative feedback. All participants were between 20 and 40 years of age and had at least fundamental knowledge and experience regarding the design and development of post-WIMP applications.

The organization of the tool in four author layers was generally appreciated and found to be valuable, especially for large projects, though there was disagreement concerning the need for a user layer. While one subject found this layer particularly useful to include customers into the authoring process, another participant stated that customers might have no interest in actively participating in the development.

The division into four editor modes was regarded as useful. The fact that, for instance, the content mode offers rather different functionality on each layer was not regarded as confusing.

The concept of fading between layers using a slider instead of switching discretely was generally appreciated and regarded as enjoyable. Still, some participants pointed out the importance of the possibility to switch between layers faster, especially when using the tool for a longer time.

Creation of component and action elements in the code layer and usage of those elements in the upper layers was considered easy and intuitive. However, all subjects missed a possibility for the interface and post-WIMP designers to predefine custom components and actions they need the programmer to implement. In the CLAY concept, applications are built from the code layer to the upper layers, according to specifications negotiated between the authors. Future work might engage in a concept to allow the definition of custom components or actions not only in the code layer, but also in the GUI and post-WIMP layers.

It was generally considered problematic to create large applications with the tool as it was expected to become gradually overloaded. Nevertheless, our concept offers several approaches to handle complex applications. For instance, the author has the possibility to display connections between layers for every component individually and on demand. Post-WIMP element groups can be minimized to take up less screen space and organized hierarchically in groups, which might lead to a less overloaded screen. When inspecting several connections to the code layer at once, the author can at some point choose which of the boxes to expand in order to see the full source code of only the specific source file of interest.

The visual creation and validation of post-WIMP interactions was considered innovative and helpful, though some explanations had to be given concerning the setup of the interactions, alluding to the fact that it is not an intuitive concept which would benefit from explanatory tool tips.

In conclusion, all participants stated that they generally found the CLAY concept to be an improvement of the authoring process of post-WIMP applications and would consider working with a tool based on that concept in the future.

5 Conclusion

We presented a concept called cross layer authoring for a tool enabling the collaborative development of post-WIMP applications for several different author groups. Layers in this context denote the contribution of the particular author groups to the developed application. We identify four different layers, namely the code, GUI, post-WIMP and user layer, each corresponding to an author role. Furthermore, we split up the tasks for each of these author roles into the four editor modes content, layout, parameters and behavior. We suggest organizing cross layer authoring tools in form of a matrix, with one dimension serving to distinguish the author layers, and the other dimension used to switch between editor modes, which received consistently positive feedback in our user interviews. Our goal is to provide a tool enabling authors to work

on their respective tasks independently. In order to allow UI designers to work autonomously, we include possibilities to graphically create the GUI and post-WIMP functionality. A concept for visually enhancing user interfaces with post-WIMP input methods has been introduced. Furthermore, the graphical creation and inspection of combined post-WIMP interactions has been presented and found to be innovative and useful by the users. In order to enable connections between layers, we introduce the concept of abstract connection hooks. These connections can be graphically inspected by traversing the layers. We propose a seamless transition between layers, which we realized in our prototype CLAY with a slider as a tool for navigation which users found easy and enjoyable to use.

Future work might engage in a concept to soften the hierarchy of the layers, allowing GUI and post-WIMP designers to create custom component or action elements they need the programmer to implement. This might lead to a more bidirectional communication between layers and thus encourage the concurrency of the development process, which might result in a reduction of the development time.

Acknowledgements

This research work has been financially supported by the German Federal Ministry of Education and Research (BMBF-FHProfUnt grant no. 17043X10).

References

1. Bastide, R., Navarre, D., Palanque, P. A: Model-Based Tool for Interactive Prototyping of Highly Interactive Applications. In CHI ‘02 extended abstracts on Human factors in computing systems , CHI EA ‘02, ACM (New York, NY, USA, 2002), 516–517
2. Beaudouin-Lafon, M.: Instrumental Interaction: an Interaction Model for Designing Post-WIMP User Interfaces. In Proceedings of the SIGCHI conference on Human factors in computing systems , CHI ‘00, ACM (New York, NY, USA, 2000), 446–453
3. Hansen, T. E., Hourcade, J. P., Virbel, M., Patali, S., Serra, T.: PyMT: a Post-WIMP Multi-Touch User Interface Toolkit. In Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces , ITS ‘09, ACM (New York, NY, USA, 2009), 17–24
4. Huot, S., Dumas, C., Dragicevic, P., Fekete, J.-D., Hégron, G.: The MaggLite Post-WIMP Toolkit: Draw it, Connect it and Run it. In Proceedings of the 17th annual ACM symposium on User interface software and technology , UIST ‘04, ACM (New York, NY, USA, 2004), 257–266
5. Jacob, R. J., Girouard, A., Hirshfield, L. M., Horn, M. S., Shaer, O., Solovey, E. T., Zigelbaum, J.: Reality-Based Interaction: a Framework for Post-WIMP Interfaces. In Proceedings of the twenty-sixth annual SIGCHI conference on Human factors in computing systems , CHI ‘08, ACM (New York, NY, USA, 2008), 201–210
6. Kaltenbrunner, M.: reacTIVision and TUIO: a Tangible Tabletop Toolkit. In Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces , ITS ‘09, ACM (New York, NY, USA, 2009), 9–16
7. Khandkar, S. H., Sohan, S. M., Sillito, J., Maurer, F.: Tool Support for Testing Complex Multi-Touch Gestures. In ACM International Conference on Interactive Tabletops and Surfaces , ITS ‘10, ACM (New York, NY, USA, 2010), 59–68

8. Kin, K., Hartmann, B., DeRose, T., Agrawala, M.: Proton: Multitouch Gestures as Regular Expressions. In Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems , CHI '12, ACM (New York, NY, USA, 2012), 2885–2894
9. Klemmer, S. R., Li, J., Lin, J., Landay, J. A.: Papier-Mache: Toolkit Support for Tangible Input. In Proceedings of the SIGCHI conference on Human factors in computing systems , CHI '04, ACM (New York, NY, USA, 2004), 399–406
10. König, W. A., Rädle, R., Reiterer, H.: Squidy: a Zoomable Design Environment for Natural User Interfaces. In Proceedings of the 27th international conference extended abstracts on Human factors in computing systems , CHI EA '09, ACM (New York, NY, USA, 2009), 4561–4566
11. Lawson, J.-Y. L., Coterot, M., Carincotte, C., Macq, B.: Component-Based High Fidelity Interactive Prototyping of Post-WIMP Interactions. In International Conference on Multimodal Interfaces and the Workshop on Machine Learning for Multimodal Interaction , ICMI-MLMI '10, ACM (New York, NY, USA, 2010), 47:1–47:4
12. MacIntyre, B., Gandy, M., Dow, S., Bolter, J. D.: DART: a Toolkit for Rapid Design Exploration of Augmented Reality Experiences. In Proceedings of the 17th annual ACM symposium on User interface software and technology , UIST '04, ACM (New York, NY, USA, 2004), 197–206
13. Serrano, M., Nigay, L., Lawson, J.-Y. L., Ramsay, A., Murray-Smith, R., Denef, S.: The Openinterface Framework: a Tool for Multimodal Interaction. In CHI '08 extended abstracts on Human factors in computing systems , CHI EA '08, ACM (New York, NY, USA, 2008), 3501–3506
14. van Dam, A. Post-WIMP User Interfaces. Commun. ACM 40 , 2 (Feb. 1997), 63–67.
15. Hartmann, B., Abdulla, L., Mittal, M., Klemmer, S. R.: Authoring Sensor-based Interactions by Demonstration with Direct Manipulation and Pattern Recognition. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '07). ACM (New York, NY, USA, 2007), 145-154.
16. Hartmann, B., Yu, L., Allison, A., Yang, Y., Klemmer, S. R.: Design as Exploration: Creating Interface Alternatives Through Parallel Authoring and Runtime Tuning. In *Proceedings of the 21st annual ACM symposium on User interface software and technology*(UIST '08). ACM (New York, NY, USA, 2008), 91-100.
17. Simpson, J., Terry, M.: Embedding Interface Sketches in Code. In *Proceedings of the 24th annual ACM symposium adjunct on User interface software and technology* (UIST '11 Adjunct). ACM (New York, NY, USA, 2011), 91-92.
18. Newman, M. W., Lin, J., Hong, J. I., Landay, J. A.: DENIM: An Informal Web Site Design Tool Inspired by Observations of Practice. *Hum.-Comput. Interact.* 18, 3 (September 2003), 259-324.