IOWAState: Models and Design Patterns for Identity-Aware User Interfaces Based on State Machines

Yann Laurillau
University of Grenoble, UPMF, CNRS, LIG Laboratory
110 av. de la Chimie, Domaine Universitaire, BP 53, 38041 Grenoble cedex 9, FRANCE
{first name}.{last name}@imag.fr

ABSTRACT
The emergence of interactive surfaces and technologies able to differentiate users allows the design and development of Identity-Aware (IA) interfaces, a new and richer set of user interfaces (UIs). Such user interfaces are able to adapt their behavior depending on who is interacting. However, existing implementations, mostly as software toolkits, are still ad-hoc and mostly based on existing GUI toolkits which are not designed to support user differentiation. The problem is that the development of IA interfaces is more complex than the development of traditional UIs and still requires extra programming efforts. To address these issues, we present a set of implementation models, named IOWAState models, to specify the behavior as state machines, the architecture and the components of IA interfaces. In addition, based on our IOWAState models and a classification of IA user interfaces, we detail a set of design patterns to implement the behavior of IA user interfaces.

Author Keywords
Identity-aware user interfaces, Interactive surfaces, Software design patterns, Architecture model, State machine model.

ACM Classification Keywords
H.5.2. User Interfaces: Graphical User Interfaces, Interaction styles, Prototyping, User-Centered design. H.5.3. Group and Organization Interfaces: Web-based interaction. D.2.2. Design Tools and Techniques: User interfaces.

INTRODUCTION
Research on multi-touch interactive surfaces, in particular interactive tabletops, is now well established in the fields of Human-Computer Interaction (HCI) and of Computer-Supported Cooperative Work (CSCW). The directness of interaction and the multiuser capabilities of tabletops may explain the growing interest for these systems. Currently, several technological solutions are available [12,32] including commercial ones [9,17]. Among these technologies, few are able to differentiate users touching the surface [9,16,24,32].

In conjunction with the growing number of technological solutions allowing user identification and differentiation (e.g., [1,16]), work is done on the development of identity-aware (IA) user interfaces, taking advantage of user differentiation and showing the capabilities and benefits of such UIs (e.g. [26,27]). For instance, SIDES [25] is an IA multi-user tabletop-based interactive system designed to develop effective social skills. It shows that such category of technology is helpful for a therapeutic purpose considering teenagers with Asperger's syndrome. In particular, IA widgets requiring synchronous actions were key in its success.

As Identity-Aware User Interfaces (IAUIs) are more complex than traditional and single-user interfaces, their development is still challenging. We identify several issues:

Lack of implementation models and guidelines: developing IAUIs requires extra programming efforts due to the lack of models and of capitalization of best practices (e.g. guidelines, design patterns). We observed that existing IA applications are mostly developed from scratch and, similarly to the development of multi-touch gesture-based interactive systems, developers must deal with low-level events.

User interfaces' behavioral model split across the code: traditional UI toolkits (e.g. Java's Swing), including UI toolkits that support user differentiation (e.g. DiamondSpin [28] toolkit is based on Java's Swing), massively rely on the well-known callback-based programming model: developers have to write a bunch of callbacks to handle each input event for each UI component. Thus, they must maintain the state of the UI component across these callbacks which usually leads to produce "spaghetti" of code [21].

Dealing with concurrent inputs and differentiated outputs: although a traditional UI receives and deals with events generated by the same user, an IAUI has to manage input events generated by different users due to simultaneous actions, sometimes concurrent. Furthermore, such an UI must maintain a much more complex state model in order to produce consistent and customized outputs.

At implementation level, although most of the work done focuses on technical issues to allow user differentiation such as dedicated software toolkits, we investigate the building of software models that would help and drive the development of IAUI components. In particular, we investigate the use of state machines as a means to address the two last issues.

This paper is structured as follows. First, we introduce an example to illustrate IAUIs. Then, we present the IOWAState models, our first contribution: a set of models to specify the behavior, the main components and the architecture of IAUI components. Based on our IOWAState models, we detail our second contribution, seven design patterns to implement the behavior of IAUI components, and our methodology. We conclude with a discussion and perspectives.

ILLUSTRATIVE EXAMPLE

(a) (b) (c)

Figure 1: Cooperative gesture to transfer ownership [18].

Let us consider the following scenario: two users, Green and Blue, are interacting simultaneously on a user-differentiating multitouch surface, manipulating digital artifacts (widgets, images, shapes, etc). Some are public while others are private. Thanks to user differentiation, supporting privacy, private artifacts are accessible by their owner only. However, user Green wants to give an image he/she owns to user Blue. Thanks to user differentiation, the users Green and Blue just have to accomplish a cooperative gesture [18] to transfer ownership. As shown in Figure 1, having first activated ownership transfer mode, (a) user Green touches the image he/she wants to relinquish; (b) user Blue touches Green's image to indicate that he/she will be the next owner; (c) ownership is granted to user Blue when user Green releases his/her finger from the surface. This example is used further in the part about design patterns.

BACKGROUND AND RELATED WORK

As underlined in introduction, multi-touch technologies, especially interactive surfaces, are intensively studied and are now well known in our research communities. Therefore, in this paper we concentrate on IA User Interfaces and on development tools supporting user differentiation.

Identity-aware user interfaces

In the 90's, researchers started to investigate the development of groupware using a single and shared display: Single Display Groupware systems (SDG) are ancestors of actual research on interactive surfaces such as tabletops: co-located users were able to interact simultaneously using multiple input devices [29]. Therefore, assigning an input device per user allows user identification and thus the development of identity-aware applications. The most basic example is multi-pointers on a shared display: each user owns a pointer and is allowed to manipulate simultaneously the shared UI elements displayed on the screen. In particular, MMM [4], Pebbles [22], and Kidpad [10] are usually considered in the literature as the very first systems implementing and illustrating the concept of SDG. These systems are the first to take advantage of user identification to develop identity-aware interfaces.

Proxy-Sketch [1] is another example of identity-aware interface dedicated to the creation of GUI prototypes. User identification is used to associate owners to content. It also supports casual observers (i.e. not logged in) that prevent from accidental changes.

Idlenses [27] is an identity-aware interaction technique that revisits magic lenses to provide a moveable personal area. Once identified, users benefit of personal tools that support access control to restricted and personal data, personalized actions such as automatic filling of web forms with personal data, a private clipboard, etc.

Tse et al. [30] have investigated multi-user and multimodal identity-aware interactive systems for gaming, based on DiamondTouch [9]. The underlying mechanism for multimodal fusion uses user identification to link speech with gesture.

To capitalize the work done in this area, Ryall et al. [26] propose the conceptual iDwidgets framework. The authors define identity-aware widgets (i.e. called iDwidgets for identity-differentiated widgets) as an extension of "*the widget concept by including identity as an input parameter, which lets us customize interactions in a variety of ways*". For instance, an identity-aware paintbrush tool will adapt its color or stroke size according to the user.

Toolkits supporting user differentiation

In order to facilitate the development of identity-aware interfaces and widgets, several toolkits have been designed and developed to support user differentiation.

The very first toolkits used peripherals as a means to differentiate users. The implicit user differentiation mechanism was "*one input device, one user; one user, one input device*". For instance, Multiple Input Devices (MID) [13] is a software library built on top of Java. In order to support multiple mice, MID revisits the underlying Java event mechanism. Therefore, it allows developers to implement identity-aware interfaces based on the mouse ID. Such a piece of information is implemented as an extra attribute of event objects.

SDGToolkit [31] is an extension of MID as it supports multiple keyboards. At UI level, the toolkit provides mechanisms to support orientation in tabletop setups. This toolkit is built on top of the .NET framework and is written in C#. Similarly to MID, events generated by input devices are associated to devices based on a device ID. It allows the use of standard widgets provided by the .NET framework to develop identity-aware interfaces as well to develop its own identity-aware widgets from scratch. This toolkit gave rise to IdenTTop [24], adding support for any multi-touch devices and support for a Polemus motion tracker. In addition, IdenTTop proposes a development framework for identity-aware applications based on a set of software components.

For touch surfaces, especially DiamondTouch [9], DiamondSpin [28] is the most well-known toolkit. It is built on top of Java and extends Java's Swing GUI toolkit to support widget orientation. User identification is achieved using a similar mechanism as SDGToolkit: events generated by touches are associated to users by the way of a specific attribute: a user ID. In particular, the toolkit provides identity-aware frames (DSFrame component) allowing users to customize the appearance: a frame can be rotated, zoomed or resized. Similarly to SDGToolkit, it allows developers to reuse standard Java's Swing components in a DSFrame. Compared to DiamondSpin, the GIL Library (gil.imag.fr) is another toolkit based on DiamondTouch but built on top of Tcl/Tk

While the java-based T3 toolkit focuses on high-resolution tabletop interfaces using wireless pens as devices for user identification [32], TouchID [16] goes beyond user identification as it investigates user-, hand-, and handpart-aware tabletops. Similarly to SDGToolkit and IdenTTop, TouchID is build on top of the .NET framework and based on the Microsoft Surface touch table [17].

IOWASTATE MODELS

As our model is intended for the design and the implementation of Identity-Aware UIs (IAUI), the IOWAState model encompasses three modeling primitives:

- A behavior model based on standard state machine models to describe the behavior of an IAUI. As detailed further, we used this modeling primitive to identify recurrent behavior patterns. In particular, we highlight how user differentiation is achieved in terms of state machine.
- A component model that identifies the main components of an IAUI and their relationships. In particular, this model highlights how we handle multiple state machines in order to allow parallel or concurrent user actions on an IAUI.
- An architecture model to describe the structure of an IAUI component. It illustrates how low-level events are processed to produce high-level events and are propagated to sub-components.

In the following, as the IOWA component model is based on the Model-View-Controller (MVC) design pattern, we will refer to it.

IOWA Behavior model

We chose to model and implement the Model part using hierarchical state machines (HSM), a derivative of finite state machines (FSM). Since Newman's work [23], user interfaces are often specified using state machines [15,21,33]. In addition, several works have demonstrated the feasibility and the benefits implementing state machine-based UIs [1,5,14].

As state machines are well suited to specify mode-driven interactions, we allow the Model to encompass several state machines, one per user, and support their parallel execution. Indeed, collaborative settings such as tabletops enable the interleaving modal actions.

In addition, using state machines facilitated the comparison of identity-aware widget's implementations and helped us to identify classes of identity-aware widgets based on their implementation model.

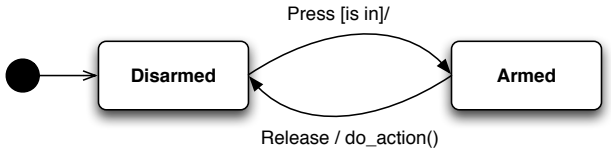

Figure 2: Example of state model of a button.

A state machine is a combination of states and transitions connecting states. Using UML statecharts, transitions are labeled according to the following syntax: *trigger* [*guard*] / *effect*. *Trigger* is an event name, *guard* is a set of conditions and *effect* is an action executed when the transition is triggered. Figure 2 shows a classic state model of a button constituted of two states: disarmed: the button is raised; armed: the button is pushed. Such a state model responds to the press and release events. For instance, if the active state is "Armed" while a release event occurs, the *do_action()* is fired and the button goes in the "Disarmed" state.

Finally, the main advantage of Hierarchical State Machines is to facilitate the control of the state explosion problem as it allows the refinement of states as finite state machines. Indeed, specifying a state model using HSMs is a top-down approach like problem solving: an overall state model is decomposed into FSMs as problems are decomposed into smaller problems. For instance, HSMs are part of UML to specify state machines.

IOWA Component model

The IOWA component model slightly differs from the MVC design pattern as an IOWA Component inherits from an IOWA StateMachine (i.e. Model) and an IOWA UI (i.e. View). The main advantage is to present a component that looks externally as a whole, hiding the model and view

parts, while preserving modularity and loose coupling between the View and the Model.

In order to support the design of IAUIs, user differentiation is first achieved at the Model level. As shown in Figure 3, the Model is an instance of an IOWA StateMachine that describes the behavior of an IAUI, as explained in the previous part, with an IA state machine. Such a state machine is hierarchical as each state (i.e. IOWA State) may be described as a hierarchy of states. Transitions between states are triggered by events sent through a *post()* operation. Events are propagated in the state hierarchy. As events carry the identity of the user (i.e. user ID) who performs the associated action, this mechanism allows the design of IA state machines.

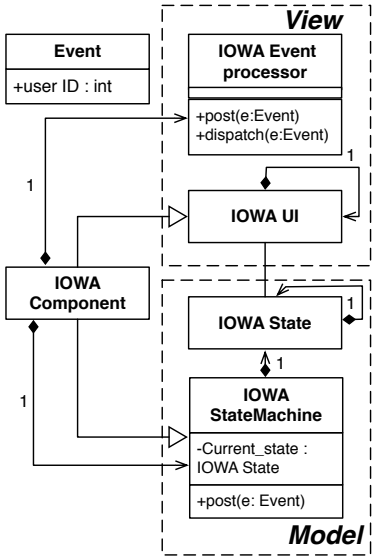

Figure 3: IOWAState's component model.

In order to support the interleaving of different user's actions and concurrent actions, although an IOWA Component is already statemachine, an IOWA Component may handle a set of IOWA StateMachines, one per user. Indeed, each event received by an IOWA Component and processed by the *post()* operation is dispatched to the state machine associated with the user ID that produces such an event.

An IOWA State component is responsible for handling high-level events supplemented with a user ID and achieving user-differentiation. Indeed, depending on the event type and the user ID, an IOWA State component verifies conditions on transitions associated to it: if a condition is verified, this component indicates to the related IOWA StateMachines component what the new state is.

As part of the View, an IOWA UI produces an output representation to the user. It defines the look and feel of an IAUI. In this model, similarly to HsmTk [5], an IOWA UI is a composition of IOWA UIs, one per state. For input events, an IOWA UI is associated with an IOWA Event Processor that receives low-level events and produces high-level events sent to the IOWA Component through a *post()* operation. Such an IOWA Event Processor may be seen as a pipeline of event filters.

IOWA Architecture model

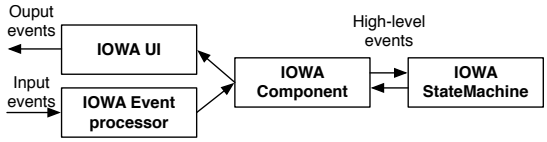

Figure 4: IOWAState's architecture model.

As shown in Figure 4, the IOWA architecture model is layered according to the MVC design pattern. As explained previously, an IOWA UI and an IOWA Event Processor constitutes the View while an IOWA StateMachine constitutes the Model. They are assembled to constitute an event processing chain that processes user' input events and generates an output representation. As an IOWA Component may be a composition of sub-IOWA Component, in addition to the dispatch of events to the state machine, the IOWA Event Processor dispatches events to the sub-components. Furthermore, the state machine may generate events that are also dispatches to the sub-components.

IMPLEMENTING IOWASTATE MODELS

The IOWAState Models, in particular the IOWA behavior model, may be directly specified with an object-oriented programming language that allows a one-to-one correspondence between the IOWAState Models and the implementation. We chose such an approach because, as underlined in introduction, IAUIs are more complex to design and to implement than traditional single-user UIs. The implementation step is usually complex as existing toolkits that support user differentiation mostly rely on usual WIMP toolkits (e.g. Java's Swing). To address this issue, in particular about the implementation of state machines, existing works advocate a developer-centric approach claiming a tight integration of models with dynamic programming languages [2,5,11]. Indeed, a state machine leads to produce code easier to read and to maintain. In addition, it supports a better reusability and extensibility as we may easily add, remove or modify states and transitions thanks to the inheritance mechanism supported by object oriented programming languages.

In order to demonstrate the validity of our IOWAState models, without giving implementation details, we implemented eight very different IAUI components. Although existing implementations focus on customization of appearance [16,24,28,30,32] (e.g. orientation to a particular user), we focus on component's behavior in terms of internal/external functionality and group input [26]. Precisely, in order to cover the largest range of IAUI component classes as identified by Ryall et al. [26], the components we implemented are taken and adapted from [18,19,20,26].

For instance, one of the eight components we implemented is a multi-user slider having a differentiated behavior, performing the same action (i.e. selecting a value) whoever the user is. However, it behaves with different styles depending on the user's identity. For instance, one user may slide the cursor from tick to tick and select a value on a discrete scale, another user would slide the cursor continuously.

Another example is a cumulative voting component allowing different users clicking on a same button to perform an action. Achieving the action requires a minimum number of users performing the interaction.

The height IAUI components we implemented are developed in Python, to be used with a Diamondtouch device [9]. In order to be independent from any GUI toolkit and their associated programming paradigm, we used basic graphic primitives to draw the components (i.e. OpenGL rendering engine). In order to support identity-awareness, we rely on the user-differentiation mechanism provided by the Diamondtouch device [9], able to differentiate up to four users. The low-level events sent by the device are supplemented with a user ID represented as an integer value in a range of 0-3. It allowed us to implement an event loop that sends high-level events supplemented with a user ID to the user interface and thus to our IAUI components.

DESIGN PATTERNS

Methodology

In order to identify recurring design patterns for IAUIs, we defined and followed a twofold method. The first part of this method consists in analyzing and in reverse-engineering the source code of existing identity-aware widgets to detect recurring implementation patterns. The second part of this method consists in developing identity-aware widgets using state machines to model and implement widget's behavior. We chose to reuse and adapt existing identity-aware widgets that are the ones described in the previous section. Obviously, these developments are on our IOWA state models.

Code-based analysis of existing IA widgets

Concomitantly with the development of the eight widgets detailed in the previous section, we analyzed the code of a set of existing prototypes that includes IA widgets. We focused on prototypes developed with toolkits allowing user identification: SDG [31], DiamondSpin [28], T3 [32], TouchID [16] and GIL [3]. We did not consider the IdenTTop toolkit [24] because the code is not publicly available. Although several IA widgets and the related source code are available online, we also requested additional examples from the authors of the DiamondSpin and GIL toolkits.

We analyzed seven IA widgets taken from SDG, DiamondSpin, and GIL. We found no relevant widget for the T3 and TouchID toolkits. The source code was reversed engineered to identify implementation patterns of identity-aware widgets. First, we carefully examined the code as follows: (1) identification of callbacks or related methods managing user input events supplemented with a user ID; (2) identification of attributes used to store the component state; (3) identification of control structures that use the user ID to update the attributes related to the component state. Then, we modeled IA widgets using state machine representations. In order to verify our models, we compared the models at runtime. In order to classify state machines and to derive patterns, based both on our developments and on the analysis of existing components, we focused on similarities and differences in terms of states (e.g. associated states) and transitions (i.e. conditions).

The IA widgets and IA interaction techniques we analyzed are:

- From SDG toolkit, a multi-user button (SDGButton) allowing two interaction modes: (1) restricted interaction to the first user pushing the button (one-user-at-a-time); (2) cumulative effect; a multi-user check button (SDGCheckButton) that paints parts of its border with the color related to the users that checked it; a multi-user slider with multiple cursors (SDGTrackBar), one per user.
- From DiamondSpin toolkit: an identity-aware and moveable menubar (DSMenuBar); a multi-user chess board [8] (RealTimeChess); a RingMenu.
- From GIL toolkit: a cooperative design application to assemble shapes in order to design a building.

Design patterns

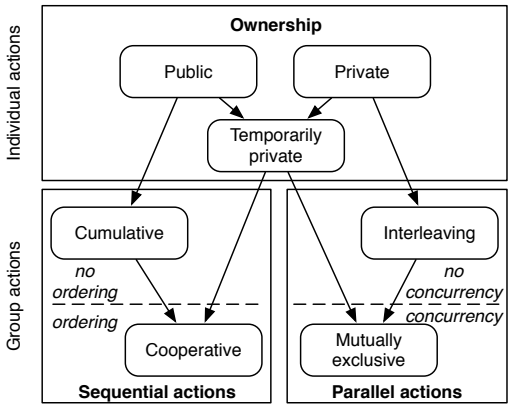

Figure 5: Design pattern graph.

As shown in Figure 5, our method leads us to identify three categories of patterns related to:

- *Individual actions*: these patterns deal with ownership, i.e. how a UI component is owned by one or multiple users. We identify three kinds of ownership: (a) public UI components that are free and not owned; (b) private UI components that are owned by one or multiple users and that can exclusively be used by the owners; (c)

temporarily UI components that are free UI components owned for a limited amount of time.

- *Group actions to achieve a sequence of actions*: these patterns identify UI components that require multiple users to achieve a group action: (a) cumulative UI components that take into account the number of users whatever the sequence of action is; (b) cooperative UI components that imply a well-defined and ordered sequence of actions.
- *Group actions allowing parallel execution of actions*: we identify two situations: (a) the interleaving of actions with no concurrency; (b) mutually exclusive UI components to deal with concurrency.

In the following, we detail each design pattern using Borchers' pattern language [6]. In addition, illustrations of state machines are given using UML statecharts.

Public IAUI

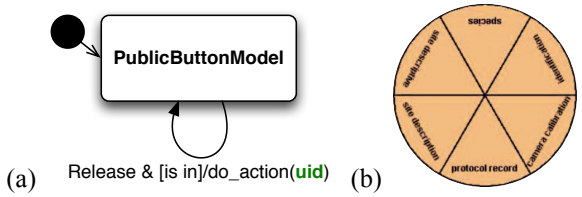

Figure 6: (a) SM model for Public IAUI; (b) TeamTag centralized control [19].

Context: in order to achieve an individual task, different users simultaneously interact with a same UI element (e.g. a button) of the shared workspace to issue a command that acts on an artifact associated with her/him.

Problem: First, traditional widgets are single-user and do not support simultaneous actions. Secondly, the display may offer a limited amount of space: replicated UI elements would clutter the interacting space and would waste pixels. Thirdly, simultaneous but opposite actions on a same UI element would produce an inconsistent visual representation or have no effects: for instance, a user is pressing his/her finger on a button that should look armed while another user releases his/her finger on the same button that should look disarmed.

Solution: a single instance of an identity-unaware state machine composed of a single state would support simultaneous actions: transitions are labeled without uid-based conditions. Thus, user differentiation is achieved by an external function triggered when an action is performed on the UI (i.e. associated to the triggered state transition such as the function `do_action(uid)` shown in Figure 6 (a)). Such a function takes the user id associated to a user event as an argument: different actions are executed according to the user id.

To support presentation consistency for simultaneous actions, a unique output representation is coupled with the state machine because the state machine is composed of a single state.

Examples: TeamTag's IA controls [19] (Figure 6 (b)).

Private IAUI

Context: an interactive surface is partitioned into shared and private territories, allowing users to interact with private artifacts located in their private territory and to perform individual tasks.

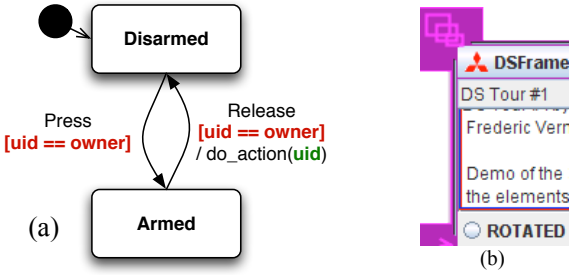

Figure 7: (a) SM model for Private IAUI; (b) Swing widgets in a DSFrame [28].

Problem: an interactive surface is naturally a public shared resource as everything is visible and potentially free, including private territories. Tacit social rules are the most common mechanism that preserves private territories.

Solution: an IAUI exclusively associated to an owner, based on his/her user id, prevents other users to interact with such private UI elements. All transitions of the state machine associated with the private IAUI must be labeled with uid-based conditions: when an event is received, a transition is triggered if the user ID carried by the event matches the owner ID (e.g. condition `[uid == owner]` as shown in red in Figure 7 (a)). We may consider that an owner is associated to such an IAUI element at instantiation time.

Examples: Swing widget in a DSFrame [28] (Figure 7 (b)), IdLenses [27].

Temporarily Private IAUI

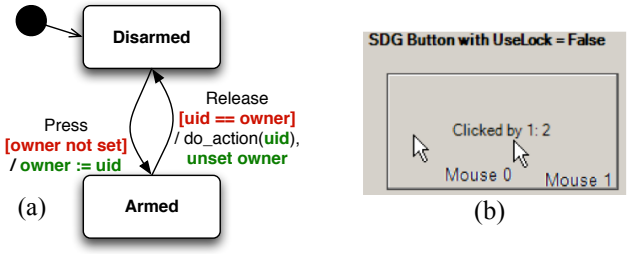

Figure 8: (a) SM model for Temporarily Private IAUI; (b) Single-user lock SDGButton [31].

Context: different users simultaneously access to a shared and free UI element such as a widget or an artifact (e.g. digital photo).

Problem: although some UI elements are public and freely available, some UI elements may only support interactions for one user at a time.

Solution: an IAUI element temporarily owned by the current user interacting with the IAUI: ownership is granted

to the very first user that interacts with the IAUI element; ownership is released when the user action is completed. To support such mechanism, the state machine associated to a temporarily private IAUI element should be designed based on two categories of transitions: transitions labeled (a) without and (b) with uid-based conditions. The first category allows any user to take ownership on a free IAUI element (e.g. condition `[owner not set]` as show in red in Figure 8 (a)): when this kind of transition is triggered, the current user is then marked as the current owner of the IAUI element he/she is manipulating (e.g. effect `owner := uid` as shown in green in Figure 8 (a)). Therefore, the IAUI element is considered as private. Similarly to Private IAUI, the remaining transitions are related to the second category (e.g. condition `[uid == owner]` as shown in red in Figure 8(a)). However, when triggered, at least one transition of the second category must release ownership (e.g. effect `unset owner` as show in green in Figure 8 (a)).

Examples: DSMenuBar [28], Single-user lock SDGButton [31] (Figure 8 (b)).

References: PUBLIC IAUI, PRIVATE IAUI.

Cumulative IAUI

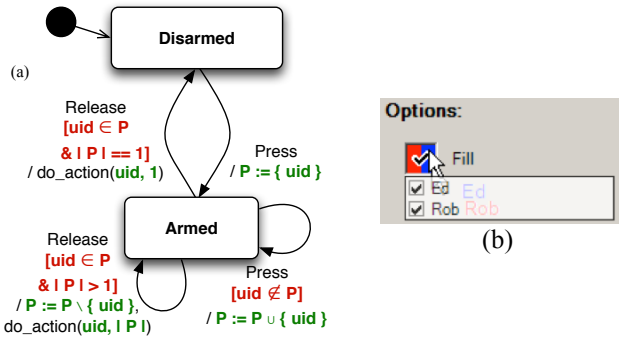

Figure 9: (a) SM model for Cumulative IAUI; (b) Cumulative SDGCheckButton [31].

Context: different users are interacting with the same UI element to perform a group and synchronized action.

Problem: the UI element must consider how many users (i.e. critical mass) are interacting to achieve a group action (e.g. majority). Furthermore, this UI element must remember who is interacting to take into account each user only once: for instance, a user touching a button with two different fingers must be counted as a single touch.

Solution: an IAUI element that maintains a list of users already interacting with it. This list is updated when transitions of the associated state machine are triggered. Three categories of conditions are observed:

- Conditions verifying if a user is not already in the list to avoid duplicate entries (e.g. condition `[uid ∉ P]` as shown in Figure 9 (a)). Consequently, for transitions that verify such a condition, the associated action consists in adding the new interacting user to the list (e.g. condition `[P := P ∪ {uid}]` as shown in Figure 9 (a)).
- Conditions verifying if a user is already on the list (e.g. condition `[uid ∈ P]` as shown in Figure 9 (a)) when the user interaction is completed. Consequently, for transitions that verify such a condition, the associated action consists in removing the associated user from the list (e.g. condition `[P := P \ {uid}]` as shown in Figure 9 (a)).
- Conditions verifying if no more users are interacting with the IAUI element to maintain state consistency (e.g. condition `[|P| > 1]` where `|P|` denotes the cardinality of set `P` as shown in Figure 9 (a)). Such a condition can be seen as threshold to reach in order to select a state transition in case of alternatives.

Although a Public IAUI element responds to individual actions, a Cumulative IAUI element responds to group actions. Similarly, there is no owner associated with it.

Examples: SDGButton [31] (Figure 9 (b)), Voting button [20], SIDES [25], SDGTrackBar [31].

References: PUBLIC IAUI.

Cooperative IAUI

Context: different users are interacting with the same UI element to perform a synchronized group action, involving a limited number of users. Achieving the group action requires to execute actions in a certain order (i.e. ordered sequence of actions). Depending on the number of users or depending on who is interacting, the UI element behaves in different ways (modes).

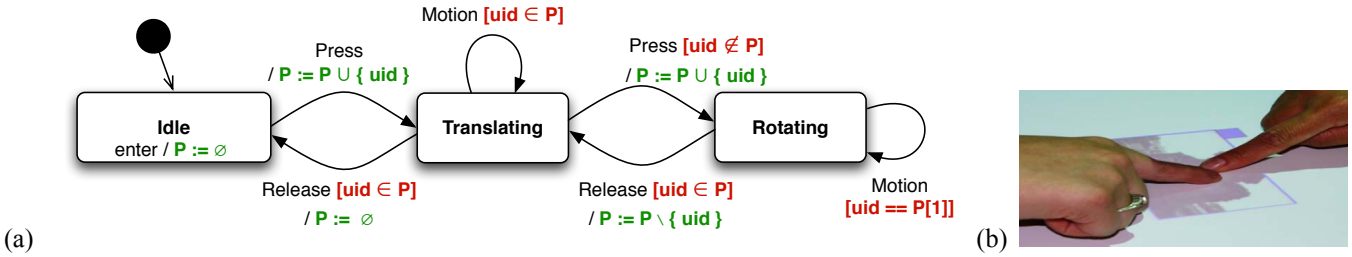

Figure 10: (a) SM model for Cooperative IAUI; (b) Cooperative gesture [17].

Problem: the UI element must consider how many users are interacting to achieve a group action. Furthermore, this UI element must remember who is interacting to take into account each user only once. As the UI element behaves differently depending on who is interacting, several states must be considered to represent the sequence of actions.

Solution: an IAUI element's state machine composed of an ordered set of states. This set corresponds to the ordered sequence of actions that the users must execute to achieve the group action. Each state is associated to different behaviors of the IAUI element. User differentiation is performed to (1) limit the number of users interacting with the IAUI element using a list similarly to a Cumulative IAUI element; (2) to associate a user for different modes of interaction using uid-based conditions (e.g. condition `[uid == P[1]]` as shown in red in Figure 10 (a)). As the number of users allowed to interact with a Cooperative IAUI element is limited, such a component may be seen as Temporarily private IAUI element.

Examples: Cooperative gesture [18] (Figure 10 (b)), Rotating shape (Figure 1).

References: TEMPORARILY PRIVATE IAUI, CUMULATIVE IAUI.

Interleaving IAUI

Context: different users are simultaneously interacting in a shared workspace on different artifacts. Some of the users may execute destructive actions (e.g. delete).

Problem: using a global mode (i.e. the same mode for all) in a shared workspace does not support parallel moded interactions. For instance, if one person is in an erasing mode, other persons cannot be in a different mode such as drawing: once the erase mode is activated, the next selected stroke would be erased.

Solution: an IAUI component's state machine managing a set of multiple instances of the same sub-state machine that are running in parallel (Figure 11 (a)). The master state machine intercepts the events and, as a proxy, dispatches events to each instance. Each instance is owned (i.e. private) by a user (e.g. conditions `[uid == user_N]` on transitions as shown in red in Figure 11 (a)) and is responsible for the management of moded interactions. Such a mechanism allows the interleaving of actions and avoids concurrent actions, even for destructive actions.

Examples: DTMap [26] (Figure 11 (b)).

References: PRIVATE IAUI.

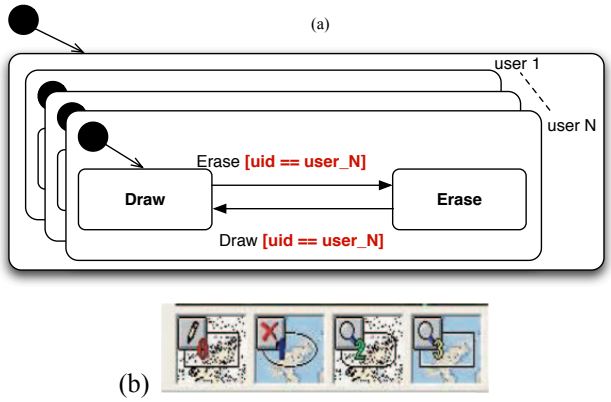

Figure 11: (a) SM model for Interleaving IAUI; (b) DTMap.

Mutually exclusive IAUI

Context: Two users are interacting simultaneously with the same UI component.

Problem: a user must wait for the first user already interacting to end up taking his/her turn and then accomplish his/her own action.

Solution: similarly to an Interleaving IAUI component, a Mutually exclusive IAUI component is based on a master state machine that manages several sub-state machines running in parallel. In addition, each sub-state machine implements an Idle/Active mechanism: the idle state is reached when a user is not interacting; the active state is reached when a user is interacting. For the latter, two sub-states are considered in order to support mutual exclusion and the fact that a user must wait his/her turn: two sub-sub-states are considered as show in Figure 12 (a): an operative state that locks the IAUI component (i.e. ownership taken) until the interaction is ended up (i.e. ownership released); a non-operative state that corresponds to a stand-by period.

Examples: Waiter's Diamondspin mechanism [28], RingMenu [8] (Figure 12 (b)).

References: TEMPORARILY PRIVATE IAUI, INTERLEAVING IAUI.

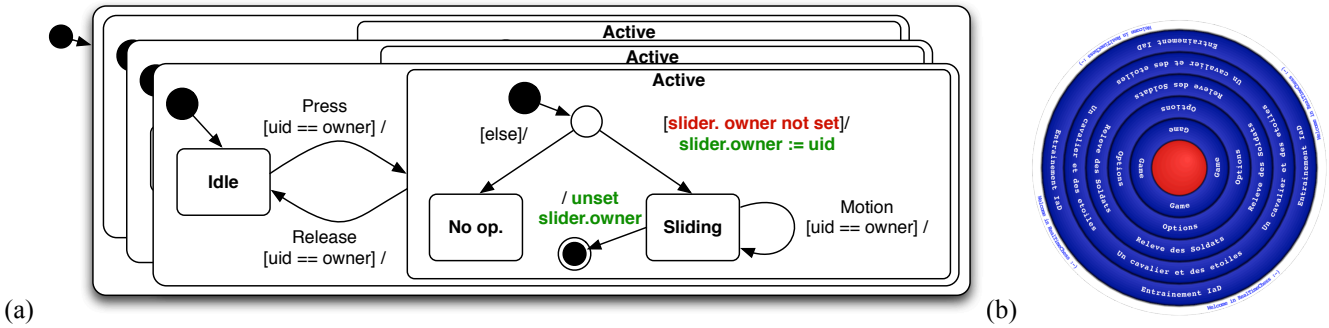

Figure 12: (a) SM model for Mutually exclusive IAUI; (b) RingMenu [7].

DISCUSSION

IOWA models

As a first evaluation, we instantiated the IOWAState models to develop eight very different IAUI components. As a second evaluation, we used our models as a framework to analyze existing implementations and to identify recurring patterns. Of course, a long-term evaluation would be clearly appropriate for a good understanding of the strengths and weaknesses of our models. In particular, we currently use our models to implement a serious game, based on a DiamoundTouch device, for the learning of cooperative practices for engineering tasks.

As the IOWA models are based on HSMs to specify the behavior of IAUIs, our approach is similar to SwingState [2], StateStream [11] or HsmTk [5] models and implementations. Although these works target single-user interfaces, our models are designed to support IA user interfaces. In particular, an IOWA component supports simultaneous user inputs and an ownership mechanism in order to allow the development of Private and Temporarily private IAUIs. In addition, our models are designed to support the parallel execution of HSMs within a MVC-like architecture in order to allow the development of Interleaving and Mutually exclusive IAUIs. The IOWA architecture model is designed to allow compositions of state machines. However this point is out of the scope of this article.

Compared to existing IA toolkits [15,23,28,32,33] widely based on a callback-based programming model inherited from traditional GUI toolkits, since our models are based on HSMs to specify the behavior of IAUIs, our several developments show it can be easily translated into code in order to produce code easier to read and to maintain, avoiding the use of a specialized and additional language. Furthermore, as we adopted an object-oriented programming approach for the implementation of the IOWAState models, we observed that the inheritance mechanism facilitates the reuse of existing HSMs. It also facilitates the creation of new behaviors with minor modifications of existing HSMs. It seems an interesting property to investigate further in order to address state explosion.

Currently, as explained previously, a first limit of our approach is the lack of long-term evaluation. Particularly, we consider another long-term evaluation with Master students following computer engineering courses, asked to implement IAUIs based on our models. Focusing only on IAUI's behavior constitutes another limit. Investigating how our models are extensible to support user-differentiation at presentation level must be considered further. Finally, we do not address the combination of two IAUI components, in particular two IAUI components having conflictual behaviors.

Design patterns

In terms of evaluation, according to [7], a pattern follows a lifecycle model composed of several steps. Currently, our patterns have reached step #5 "Pattern Gestalt" for which readers review the patterns. This article contributes to this step. The next step must be "Popular acceptance". Contributing to the evaluation as well as demonstrating the completeness of our patterns, our pattern classification covers the classification of the IDWidgets framework [26] related to behavior, and coherently integrates cooperative gestures [18]. In addition, we go one step further towards software implementation of IAUIs as we provide and detail seven design patterns. Furthermore, although CSCW literature considers UI elements' ownership as private or public, we identify a new and intermediate situation of ownership: temporary ownership.

Complementary to the conceptual IDWidgets framework [26] providing classes of IAUI widgets, our pattern classification is at implementation level and identifies classes of identity-aware user interactions.

Except the fact that our patterns should reach step "Popular acceptance" (step #7), an unanswered issue is the completeness of our design patterns and the related classification. Particularly, our patterns focus on behavior only and patterns for user-differentiated presentations should be investigated further.

CONCLUSION

Focusing on the design and development of Identity-Aware User Interfaces, this article presents two main findings. First, the IOWAState models revisit the MVC architecture model to rely on hierarchical state machines in order to support identity awareness, simultaneous user inputs, and to help developers to produce code easier to read and to maintain. Another significant contribution is a classification of IAUIs based on a set of seven design patterns to specify the behavior of IAUIs using state machines.

As a perspective, we need to investigate rules to combine several IAUI components. Indeed, combining two IAUI components may lead to the combination of conflictual HSMs such as a Private IAUI component embedding a Public IAUI component. In terms of implementation, we need to investigate alternative programming languages to Python to demonstrate the generative power of the IOWAState models. Finally, we plan to extend our patterns and the state machine approach to single-user multi-touch user interfaces.

ACKNOWLEDGMENTS

To Renaud Blanch for advice and hints he gave about HSMs and the permission to use his HSM-based SWIT toolkit.

REFERENCES

1. Annett, M., Grossman, T., Wigdor, D., and Fitzmaurice, G. Medusa: a proximity-aware multi-touch tabletop. In *Proc. of UIST 2011*, ACM Press (2011), 337–346.

2. Appert, C., and Beaudouin-Lafon, M. SwingStates: adding state machines to the Swing toolkit. In *Proc. of UIST 2006*, ACM Press (2006), 319–322.
3. Bérard, F., and Laurillau, Y. Single User Multitouch on the DiamondTouch: From 2x1D to 2D. In *Proc of ITS 2009*, ACM Press (2009), 1–8. http://gil.imag.fr
4. Bier, E., and Freeman, S. MMM: A User Interface Architecture for Shared Editors on a Single Screen. In *Proc. of UIST 1991*, ACM Press (1991), 79–86.
5. Blanch, R., and Beaudouin-Lafon, M. Programming rich interactions using the hierarchical state machine toolkit. In *Proc. of AVI 2006*, ACM Press (2006), 51–58.
6. Borchers, J. A pattern approach to interaction design. In *Proc of DIS 2000*, ACM Press (2000), 369–378.
7. Brown, W., Malveau, R., McCormick, H., Mowbray, T., and Thomas, S.W. The Software Patterns Criteria (1998), http://www.antipatterns.com/whatisapattern/
8. Chaboissier, J., Isenberg, P., and Vernier, F. RealTimeChess: lessons from a participatory design process for a collaborative multi-touch, multi-user game. In *Proc. of ITS 2011*, ACM Press (2011), 97–106.
9. Dietz, P., and Leigh, D. DiamondTouch: A multi-user touch technology. In *Proc. of UIST 2001*, ACM Press (2001), 219–226.
10. Druin, A., Stewart, J., Proft, D., Bederson, B., and Hollan, J. KidPad: a design collaboration between children, technologists, and educators. In *Proc. of CHI 1997*, ACM Press (1997), 463–470.
11. de Haan, G., and Post, F. StateStream: a developer-centric approach towards unifying interaction models and architecture. In *Proc of EICS 2009*, ACM Press (2009), 13–22.
12. Han, J. Y. Low-cost multi-touch sensing through frustrated total internal reflection. In *Proc of UIST 2005*, ACM Press (2005), 115–118.
13. Hourcade, H.P., and Bederson, B. Architecture and implementation of a java package for multiple input devices (MID). *Univ. of Maryland Human–Computer Interaction Lab. (HCIL)*. Tech. report no. 99–08 (1999).
14. Kin, K., Hartmann, B., DeRose, T., and Agrawala, M. Proton: multitouch gestures as regular expressions. In *Proc. of CHI 2012*, ACM Press (2012), 2885–2894.
15. Letondal, C., Chatty, S., Phillips, G., André, F., and Conversy, S. Usability requirements for interaction-oriented development tools. *Psychology of Programming*, Maria P. D. Pérez and M.B. Rosson (2010), 12–26.
16. Marquardt, N., Kiemer, J., Ledo, D., Boring, S., and Greenberg, S. Designing user-, hand-, and handpart-aware tabletop interactions with the TouchID toolkit. In *Proc. of ITS 2011*, ACM Press (2011), 21–30.
17. Microsoft Surface, www.microsoft.com/surface/
18. Morris, M.R., Huang, A., Paepcke, A., and Winograd, T. Cooperative gestures: multi-user gestural interactions for co-located groupware. In *Proc. of CHI 2006*, ACM Press (2006), 1201–1210.
19. Morris, M.R. TeamTag: exploring centralized versus replicated controls for co-located tabletop Groupware. In *Proc. of CHI 2006*, ACM Press (2006), 1273-1282.
20. Morris, M. R. Designing Tabletop Groupware. In *Adjunct Proc. of UIST 2005*, ACM Press (2005).
21. Myers, B.A. Separating application code from toolkits: eliminating the spaghetti of callbacks. In *Proc. of UIST 1991*, ACM Press (1991), 211–220.
22. Myers, BA, Stiel, H., and Gargiulo, R. Collaboration using multiple PDAs connected to a PC. In *Proc of CSCW 1998*, ACM Press (1998), 285–294.
23. Newman, W.M. A system for interactive graphical programming. In *Proc. of AFIPS 1968*, ACM Press (1968), 47–54.
24. Partridge, G.A., and Irani, P.P. IdenTTop: a flexible platform for exploring identity-enabled surfaces. In *Ext. Abstr. of CHI 2009*, ACM Press (2009), 4411–4416.
25. Piper, A.M., O'Brien, E., Morris, M.R. and Winograd, T. SIDES: A cooperative tabletop computer game for social skills development. In *Proc. of CSCW 2006*, ACM Press (2006), 1–10.
26. Ryall, K., Esenther, A., Forlines, C., Shen, C., Shipman, S., Morris, M.R., Everitt, K., and Vernier, F. Identity-differentiating widgets for multiuser interactive surfaces. *IEEE Comput. Graph. 26*, 5 (2006), 56–64.
27. Schmidt, D., Chong, M.K., and Gellersen, H. IdLenses: dynamic personal areas on shared surfaces. In *Proc. of ITS 2010*, ACM Press (2010), 131–134.
28. Shen, C., Vernier, F., Forlines, C., and Morris, M.R. DiamondSpin: an extensible toolkit for around-the-table interaction. In *Proc. of CHI 2004*, ACM Press (2004), 167–174.
29. Stewart, J., Bederson, B., and Druin, A. Single display groupware: a model for co-present collaboration. In Proc. of CHI 1999, ACM Press (1999), 286–293.
30. Tse, E., Greenberg, S., Shen, C., Forlines, C., and Kodama, R. Exploring true multi-user multimodal interaction over a digital table. In *Proc. of DIS 2008*, ACM Press (2008), 109–118.
31. Tse, E., and Greenberg, S. Rapidly prototyping Single Display Groupware through the SDGToolkit. In *Proc. of AUIC 2004*, Australian Computer Society (2004), 101–110.
32. Tuddenham, P., and Robinson, P. T3: A toolkit for high-resolution tabletop interfaces. In *Ext. Abstr. of CSCW 2006*, ACM Press (2006), 2237–2242.
33. Wellner, P. Statemaster: A UIMS based on statechart for prototyping and target implementation. In *Proc. of CHI 1989*, ACM Press (1989), 177–182.