A Design Pattern for Multimodal and Multidevice User Interfaces

Alessandro Carcangiu
Department of Electrical and Electronic Engineering, University of Cagliari
Via Marengo 2, 09123 Cagliari, Italy
alessandro.carcangiu@diee.unica.it

Gianni Fenu
Department of Mathematics and Computer Science, University of Cagliari
Via Ospedale 72, 09124, Cagliari, Italy
fenu@unica.it

Lucio Davide Spano
Department of Mathematics and Computer Science, University of Cagliari
Via Ospedale 72, 09124, Cagliari, Italy
davide.spano@unica.it

ABSTRACT
In this paper, we introduce the MVIC pattern for creating multidevice and multimodal interfaces. We discuss the advantages provided by introducing a new component to the MVC pattern for those interfaces which must adapt to different devices and modalities. The proposed solution is based on an input model defining equivalent and complementary sequence of inputs for the same interaction. In addition, we discuss Djestit, a javascript library which allows creating multidevice and multimodal input models for web applications, applying the aforementioned pattern. The library supports the integration of multiple devices (Kinect 2, Leap Motion, touchscreens) and different modalities (gestural, vocal and touch).

ACM Classification Keywords
H.5.2. Information Interfaces and Presentation (e.g. HCI): User Interfaces; Input devices and strategies (e.g., mouse, touchscreen).

Author Keywords
Input Modelling; User Interface Engineering; Design Pattern; Multimodal Interfaces; Device Independence

INTRODUCTION
In later years, we witnessed the introduction in the mass-market of different interaction devices that made it feasible to transform into interactive environments different places that usually had no interactive capabilities. In this category, we can remember for instance Microsoft Kinect[1], which has been employed in different settings such as home environments, interactive showcases, artistic installations etc. However, even if the Kinect is one of the most famous, there are also other devices that lately provided new interaction capabilities to a wider audience: the Leap Motion[2], which provides an accurate tracking of the hand and fingers position in 3D at an affordable price, head-mounted displays such as the Oculus Rift[3] or even mobile phones mounted inside an headset (e.g. Samsung Gear VR[4]), which allow to create virtual reality experiences at a consumer level.

The currently increasing popularity of the aforementioned devices allows interaction designers to exploit them for creating new experiences outside the game environment, in particular using the web for reaching a wider audience. However, as usual in the web, it is difficult to select a particular device for an application since the same input may be provided by different devices or by different combinations of devices. Therefore, if a designer focus on one particular device (or one particular setting), a portion of the potential audience is lost.

In this paper, we propose a variation of the Model View Controller (MVC) pattern that allows developers to redefine equivalent inputs for different devices, decoupling the input sources from their interpretation inside the application. We called this pattern Model View Input Controller (MVIC). We first detail our contribution abstracting from a particular implementation technology, discussing how it is possible to add to the classical MVC pattern a new component that translates input events from a particular device into more abstract events, which may be related to commands or manipulations. We consider that there may be more than one modality that can be exploited by the user for conveying the same information. After that, we discuss the implementation of a proof-of-concept library, called Djestit, which supports the pattern for the web environment.

THE MVIC PATTERN
In this section we detail the main contribution of this paper, which is a pattern for defining a set of input events that abstract from a particular device. This allows to map events coming from different devices and provided through different modalities into high-level interaction events, isolating the reconfiguration of the underlying interaction according to the available devices, without handling this aspect in different parts of the source code.

[1] http://www.microsoft.com/en-us/kinectforwindows/
[2] https://www.leapmotion.com/
[3] http://www.oculus.com/
[4] http://www.samsung.com/global/galaxy/wearables/gear-vr/

EICS'16, June 21–24, 2016, Brussels, Belgium
ACM 978-1-4503-4322-0/16/06.
DOI: http://dx.doi.org/10.1145/2933242.2935876

The Input Management Problem

Before introducing the scheme of the general solution, we introduce the input management problem through a small example. We consider a web interface that enables the user to select a video inside a collection. The user interface can be organised as a grid of tiles (e.g. 4 columns and 3 rows), and the user can watch a video selecting one of the tiles. In addition, if the video list has more entries than the grid, the results are paginated and the user can sequentially explore the different pages.

If we consider a classical desktop setting, it is quite trivial to support the selection mechanism through mouse pointing and the page switching through dedicated buttons. We can add the keyboard support through e.g. navigating sequentially the grid through the TAB key and providing a short-cut for changing the page (e.g. using the arrow keys).

Applying the MVC pattern, this simple application should be described through three components:

1. The *Model*, which manages the data and models the application domain, contains the data and the metadata of the different videos.
2. The *View*, which manages the information visualization, defines the grid and the pagination layout.
3. The *Controller*, which interprets the user's input, manipulates the model and selects the view accordingly, is responsible for reacting to mouse and keyboard events.

We may want to add the support for the gestural modality into this simple interface. We can use for instance the Leap Motion, which support easily the screen pointing through a 3D finger tracking. In this case, we have modify the controller in order to add some listeners to the Leap events. And what about adding the support also for Microsoft Kinect? The device supports (hand) pointing through the skeleton tracking, which again requires a controller extension.

In our idea, what is missing in our application is a more abstract concept of pointer, whose events are handled by the controller for defining the UI reaction to the input, independently from the actual used device. Therefore, we isolate the definition of the mapping between the different devices and the pointer abstraction into a new component, that we call *Input*. Such mapping can be very simple, as happens in this case, or it may be more complex. For instance, it may involve a complete change of modality (e.g. mapping the same command associated to a button on a vocal command) or activating an abstract event through the tracking a sequence of inputs (e.g. for recognizing a gesture).

Pattern Description

As we anticipated in the previous section, we propose to define a dedicated component that manages the input coming from different devices. The component diagram for the MVIC pattern is shown in Figure 1. The components with a gray background are derived from the MVC pattern in its original version. They maintain their responsibilities in the proposed pattern: the *Model* manages the behaviour and the data of the application domain, the *View* renders the information contained in the model in a way that is both comprehensible for the user and appropriate for the considered task, while the *Controller* glues these two components, interpreting user's manipulations on the view reflecting such modifications on the model, and updating views according to model changes.

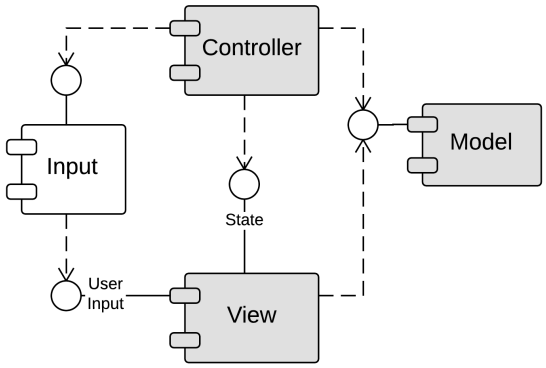

Figure 1. MVCI component diagram

In our variant of the pattern, the *Controller* does not directly receive notifications about the user's actions from the *View*, but such events are mediated by the *Input* component, which acts as a façade for the *Controller* towards the different input devices and modalities. In this way, the interface between the *Input* and the *Controller* components defines the set of abstract events that can be generated by different devices and through different modalities, which are exploited by the application.

Internally, the *Input* applies a Strategy pattern [3] for raising the abstract events, which is represented in Figure 2 by the *InputStrategy* class. The component may select one among the different available strategies according to both the available devices and a developer-defined policy for the particular application. As we better detail discussing the proof of concept implementation, the selection of the strategy can be specified either through programming code or in a declarative way (e.g. using a domain specific language).

In addition, each concrete strategy may exploit one or more data sources, coming from the same and/or different devices in order to raise one abstract event. This is modelled in Figure 2 applying the Composite pattern [3] among different refinements of the *BaseStrategy* class, which are responsible to track the input coming from a specific data source (e.g. one the touch surface of a multitouch screen, one for the audio source and one for the skeleton tracking source provided by the Kinect etc.). A generic composition is represented in the UML diagram by the *CompositeStrategy* class.

Considering the different techniques for composing the input coming from different data sources, we propose already in the pattern definition four composition techniques, according to the well-know CARE properties for multimodal interaction [1]:

- *Complementarity*: two (or more) data sources must be used for triggering the abstract event, but none of them is able to complete the change individually.
- *Assignment*: a specific abstract event can be triggered exclusively using a specific data source. In order to model

this property, there is no need to introduce a *CompositeInputStrategy* subclass, since a *BaseInputStrategy* subclass is sufficient if no other *InputStrategy* raises the same abstract event.
- *Redundancy*: an abstract event is triggered providing the same information through more than one modality (e.g. pointing plus vocal confirmation of the same selection). More than one input is needed for concluding the action.
- *Equivalence*: the same abstract event can be triggered through different data sources. The user can select one among them for completing the interaction.

Pattern Application

Adding a component in the structure of the MVC pattern helps in solving the problem of managing different input devices and to reconfigure the input strategy according to the available resources. However, the proposed solution has a cost in terms of complexity, which is adding different classes that manage the abstract events and implement the composition strategies.

It is worth pointing out that it is not always reasonable to pay such cost. First of all, it is obvious that for applications that require only mouse and keyboard for receiving input from the user, there is no need to introduce the new abstraction level. The same applies if the application designers decide to focus on one particular device for supporting the interaction, in case it is not possible or feasible to receive the same input from different devices. Therefore, the need of adapting the input management strategy to different devices or modalities is the primary criterion we suggest to consider for applying the pattern: if there is not such need, developers should not pay the additional complexity cost.

Moreover, considering the structure of the proposed pattern, it is easy to refactor the structure of an existing application that is compliant with the MVC pattern in order to support the adaptation: the *View* input events that are observed by the *Controller* are the starting point for defining the abstract events that should be raised by the new *Input* component. Once the *Input* defines the adaptation strategies, there is only the need to connect them with the *View*. No other operations on the *Controller* or the *Model* are needed.

The Input component in other UI patterns

Even if this paper focus on the MVC pattern, it is worth pointing out that a similar solution can be included also in applications that exploit different structures for separating the presentation from definition of the domain logic in a user interface. The following is a brief list of widely-known alternatives to MVC, together with some hints on how it would be possible to take advantage of the *Input* component also with these patterns.

The *Model-View-Presenter* (MVP) [11] differs from MVC since it assigns more responsibilities to the *View* and removes the *Controller*, introducing a *Presenter* component which maintains the *View*'s state and defines a set of commands. In this case, the *Input* component can be connected with the *Presenter* that, as happens for the *Controller* in MVC, reacts to abstract events rather than the ones raised by the *View*.

The *Presentation Model* [2] applies a double view-model mechanism: the *Model* has different views called *Presentation Models*, which are in turn considered as models for the *View*s. In this way, it is possible to e.g. persist the state of a View without committing the changes in the domain *Model*. In this case, the *Presentation Model* should receive updates from the *Input* component that, in turn, receives the updates from the *View*, separating the appearance from the input interpretation.

The *Model-View-View Model* (MVVM) [13] has been inspired by the Presentation Model pattern and it is exploited by recent Microsoft UI technologies such as WPF. It enforces the *View* to contain only the layout declaration (e.g. using XAML), while the behaviour part is delegated to the *View Model*, which is nearly equivalent to the *Presentation Model* in the previous pattern. The schema for adding the *Input* component is therefore equivalent to the previous case.

JAVASCRIPT IMPLEMENTATION

In this section, we provide some details on Djestit[5] proof-of-concept library that allows to create web interfaces applying the MVIC pattern introduced in this paper. First of all, we discuss the general abstractions provided by the library, then we present the extension mechanism that allows to introduce the support for different interaction devices. Finally, we provide the description of a sample UI.

Djestit is javascript library that provides support for web developers that want to apply the MVIC design patter to their web application. The library models the input (coming from different devices) through an expression that defines the input temporal sequencing. The expression is defined through the composition of two different types of terms: *ground* or *composed*.

The ground terms represent atomic events, which are notifications that cannot be further decomposed in smaller ones. In general, they are associated to a change in a specific data field tracked by an input device. For instance, we can consider a ground term the current mouse pointer position, the key press on a keyboard, the position of a skeleton joint tracked by MS Kinect or, in the vocal modality, the recognition of an utterance. Ground terms can be associated to boolean functions, in order filter the atomic events that do not fulfil a specific condition (e.g. tracking the position of a hand only if it is open).

It is possible to define composed terms connecting starting from ground terms and connecting expression through a set of temporal operators. The following is the set supported by Djestit:

- *Iterative*: recognizes an input expression an indefinite number of times. It is also possible to specify a minimum and/or a maximum number of repetitions.
- *Sequence*: connects two or more input expressions that must be recognized in sequence, in the specified order.
- *Parallel*: connects two or more expression that can be recognized at the same time.

[5]The library is publicly available at the following URL: https://github.com/davidespano/DjestIT

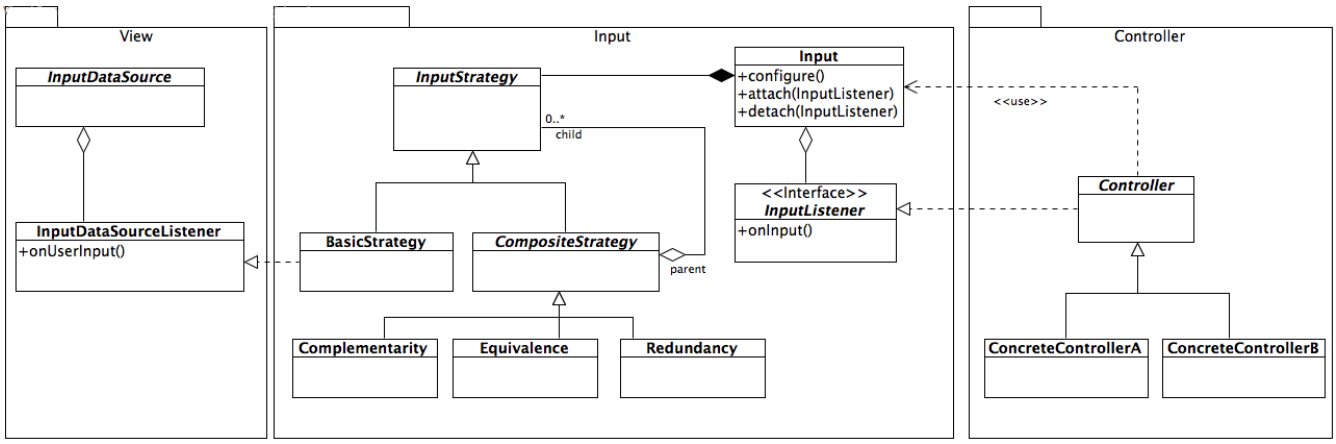

Figure 2. Class diagram for the Input and the Controller component

- *Choice*: allows the user to select one among the connected input expression for completing the recognition.
- *Disabling*: defines that the right input expression stops the recognition of the left one (usually exploited as exit conditions for loops).
- *Order Independence*: the connected input expression must be all recognized, without any constraint on which order.

Such composition mechanism provides a domain specific language (DSL) for the device and modality independent event definition. Indeed, the library raises events for the completion of each term included in the input expression (both ground or composed). In addition, the library raises error events when the received input sequence does not match the expression. Usually, such events are used for undoing uncompleted actions.

Input expressions

In order to show how it is possible to define the strategy modelling the input with the temporal expression, we consider the example in section 2.1. In this case, we want to support the selection of a tile in the video grid using i) the mouse, ii) the Leap Motion and iii) the Kinect. All of them support the pointing task. Abstracting from the device, what the controller needs is a point in the screen coordinates for highlighting one of the tiles while pointing, and a notification when the user confirms that the focused tiles is the one she wants to select.

In Djestit, it is possible to create input expressions simply defining a javascript object as shown in Table 1, which defines the pointing support for the sample application considering mouse pointing, Leap Motion and Kinect. Each ground term (`gt`) has a type that specifies the device feature it tracks. For instance the type `kinect.skeleton.handLeft` refers to the left hand point of the skeleton tracked by the Kinect. In addition, a ground term can specify a further condition (a boolean function) for notifying the input, which is useful for tracking trajectories or other input characteristics. The composite expressions are defined choosing a composition operator (e.g. `choice`) and recursively defining the operands.

After the input expression has been defined, it is possible to invoke a library function that tracks the user's input and raises the completion and error events related to the different expression terms. The support for the different devices is provided by different extension modules, one for each device, each one adding a group of ground terms types that can be tracked.

In Table 1, the expression is a choice among three sub-expressions, each one corresponding to a device. Lines 2 to 5 define the input expression for the mouse pointing: the user can move iteratively the mouse until (`disabling`) she presses the left button. Lines 6 to 17 define the pointing with the Leap Motion. The user points either with the left or right hand, therefore we first define a `choice` between two sub-expressions: one related to the left (line 7 to 11) and the right hand (line 12 to 16). The two definitions are symmetrical, both track the index fingertip iteratively until the user completes a screen tap (a rapid movement of the finger towards the screen, recognized natively by the Leap SDK). The lines 18 to 31 define the pointing using the Kinect: the user can again use either one or the other hand, moving it while it is `open` (which is a boolean function, defined through plain javascript code) and closing it for confirming the selection. Finally, the lines from 32 to 42 define a combination between the input provided by the Leap motion for moving the pointer either with the left (line 34) or the right hand (line 36), which is confirmed vocally using the Kinect voice recognition features (line 39). The ground term definition provides the grammar for recognizing the vocal command, which is the word 'select' in our case.

It is worth pointing out that the choice at the top level is ambiguous, since there are two ground terms of type `leap.handLeft.index.4` and `leap.handRight.index.4` (respectively the left and right fingertips tracked by the Leap Motion). The library is able to handle such ambiguous definitions with the techniques described in [15].

Attaching the behaviour

While the javascript object defines *what* input sequence is relevant for the application, it does not define *how* the *Input* component reacts to such sequence. In order to decouple the definition of the temporal sequence from the definition of the

```
1  var exp = {choice :[
2     {disabling:[
3        {gt: "mouse.move", iterative : true},
4        {gt: "mouse.leftButton"}
5     ]},
6     {choice :[
7        {disabling:[
8          {gt: "leap.handLeft.index.4",
9                iterative: true},
10         {gt: "leap.handLeft.screenTap"}
11       ]},
12       {disabling:[
13          {gt: "leap.handRight.index.4",
14                iterative: true},
15          {gt: "leap.handRight.screenTap"}
16       ]}
17    ]},
18    {choice :[
19       {disabling:[
20          {gt: "kinect.skeleton.handLeft",
21                condition: open
22                iterative: true },
23          {gt: "kinect.skeleton.handLeft.closed"}
24       ]},
25       {disabling:[
26          {gt: "kinect.skeleton.handRight",
27                condition: open,
28                iterative: true},
29          {gt: "kinect.skeleton.handRight.closed"}
30       ]}
31    ]},
32    {disabling :[
33       {choice :[
34          {gt: "leap.handLeft.index.4",
35               iterative: true},
36          {gt: "leap.handRight.index.4",
37               iterative: true},
38        ]},
39        {gt: "kinect.voice",
40              grammar="select"}
41    ]}
42 ]};
```

Table 1. Screen pointing input definition in Djestit (mouse, Leap Motion and Kinect)

behaviour, we chose to provide a term selection mechanism for attaching handlers to the completion and error events related to the recognition of an input expression. The mechanism works similarly to the specification of CSS properties for HTML elements: first we write a selector for choosing a set of elements (input expressions) we want to modify and then we specify their properties. For selecting the expressions we use the javascript library JSON Select[6], while the properties to be set are the *Input* component reactions to specific user input sequences.

In our example, we define as interface between the *Input* and the *Controller* components two functions: `pointingMoved(point)` which notifies that the user has changed the pointing location and `selectionConfirmed` for selecting the currently pointed object. The first abstract event should be raised when the user completes the terms at line 3, 8, 13, 20, 26, 34 and 36 in Table 1. All we have to do is transforming map the point tracked in the device space into a 2D screen point. This is trivial for mouse pointing, while for the Leap Motion and the Kinect this can be achieved with different techniques (e.g. head to hand ray tracing). Instead, the selection confirmation, which is the second abstract event,

[6] http://jsonselect.org/

```
1  djestit.onComplete( exp,
2  [":has(:root > .gt:val(\"mouse.move\"))",
3   ":has(:root > .gt:val(\"leap.handLeft.index.4\"))",
4   ":has(:root > .gt:val(\"leap.handRight.index.4\"))",
5   ":has(:root > .gt:val(\"kinect.skeleton.handLeft\"))",
6   ":has(:root > .gt:val(\"kinect.skeleton.handRight\"))"
        ],
7   function(args) {
8       var point = pointToScreen(args);
9        pointingMoved(point);
10 });
11
12 djestit.onComplete( exp,
13 [":has(:root > .gt:val(\"mouse.leftButton\"))",
14  ":has(:root > .gt:val(\"leap.handLeft.screenTap\"))",
15  ":has(:root > .gt:val(\"leap.handRight.screenTap\"))",
16  ":has(:root > .gt:val(\"kinect.skeleton.handLeft.closed
        \"))",
17  ":has(:root > .gt:val(\"kinect.skeleton.handRight.
        closed\"))",
18  ":has(:root > .gt:val(\"kinect.voice\"))"],
19   function(args) {
20      selectionConfirmed();
21   });
```

Table 2. Behaviour attachment to different input model subexpressions

should be raised by the ground terms defined at lines 4, 10, 15, 23, 29 and 39.

The Djestit library allows to attach handlers to an expression through two functions: `onComplete` and `onError`. Both take as parameter an input expression, an array of JSON Selectors that specify to which ground and/or complex terms the handler should be attached and the handler function. The code in Table 2 shows the javascript code needed for attaching the behaviour at the expression in Table 1, according to the previous description.

Discussion

We summarize here the advantages provided by the Djestit library. First of all, it helps web developers in being compliant with the MVIC pattern proposed in this paper, separating the detection of the input sequences received by different devices from their semantics. In particular, the proposed composition operators are able to support the CARE [1] properties for multimodal input. Considering a generic abstract event *eventA* that should be notified to the *Controller* by the *Input* component, the *assignment* is supported associating the abstract event only to expressions that exploit the same data source (e.g. `kinect.voice`). The *equivalence* can be supported associating the *eventA* to a choice between expressions each one exploiting different data sources, as happens for instance in Table 1. In order to support the *redoundancy* we use the order independence operator following the same scheme. Finally the *Complementarity* can be achieved connecting the expressions with all operators but choice or order independence.

Besides the MVIC pattern advantages, the Djestit implementation provides two other advantages. The first one is the modularity with respect to the devices: each one has a dedicated plugin and the developer can, including the script in a HTML page, load only those needed by the application. Each plugin registers a set of ground terms to the core support. In order to extend the library for supporting a new device, there is only the need to write the javascript code for recognizing its ground terms.

The second advantage is that Djestit supports the definition of reusable input sequences, that can be shipped with a given plugin, for recognizing common inputs (e.g. multitouch or full-body gestures). Such advantage derives from the separation between the declaration of the temporal sequence and the mechanism for attaching the behaviour to ground and composite terms. Since the same input sequence (e.g. a gesture) can be used in different contexts and may produce different effects on the UI, the separation allows decoupling the input description from its effects.

RELATED WORK

The idea of providing a model for describing the different input sources has been widely investigated in literature, especially exploiting formal notations. For instance, already Myers [10] defined a set of reusable interactors that encapsulate the interactive behaviour, hiding the details of the underlining window-manager events. Such description needed more complex interfaces for having a good exploitation: Jacob et al. [6] applied FSM to non-WIMP user interfaces, separating two aspects of such kind of interfaces. The first one is the response to continuous input, which is managed by data-flow oriented variables. The second aspect is the connection among these continuous variables that can change according to different discrete events. Another application was the bimanual interaction [4], where Petri Nets provided a simple model for parallel manipulation.

More recently, the increasing popularity of multitouch and full-body gestures reopened the quest for a more structured input modelling. Considering multitouch input modelling, Kammer et al. [7] introduced GeForMT, a formalization of multitouch gestures that aimed to fill the gap between the high level complex-gestures (such as pinch to zoom) and the low level touch events provided by different toolkits. The description language is based on grammars. Khandkar et al. [8] proposed GDL (Gesture Description Language), which separated the gesture recognition code from the definition of the UI behaviour. Proton++ [9] decoupled the definition of the multitouch gesture (described as regular expression) and its behaviour, which is attached to the expression terminal characters. GISMO [12] is a gesture definition domain-specific language, which allows to simulate and execute the behaviour of an application separating interaction controls and gestures definition.

Considering a multimodal setting, Hoste et al. [5] introduced Mudra, a rule language able to unify the input stream coming from different devices, which exploits even different modalities. It supports facts coming from different modalities (e.g. voice and hand movements), but its structure still mix the recognition and the behaviour definition. Spano et al. [14, 15] proposed a gesture modelling technique which abstracts from the recognition technology, decoupling the description of the temporal sequence from the UI behaviour. In this paper, we extend the approach for supporting generic input devices, enlarging the scope of previous work. We allow to manage different devices and modalities with a single input model, explaining how to integrate it in the MVC controller patter, which is widely applied to the development of user interfaces.

CONCLUSION AND FUTURE WORK

In this paper we described a variant of the MVC pattern for supporting different equivalent input device and modalities configuration for the same interactive application. We described how it is possible decouple the device management providing an abstraction of the user input towards the *Controller* component. In addition, we discussed the implementation of a proof of concept library which helps developers in applying the pattern for creating web applications.

In future work, we want to explore the possibility to automatically check properties on the input model, such as the access to all UI functionalities if one device or modality cannot be used. In addition, we want to provide an explicit support for input uncertainty, which may derive from the analysis of the voice signal or gesture trajectories.

REFERENCES

1. J. Coutaz, L. Nigay, D. Salber, A. Blandford, J. May, and R. Young. Four easy pieces for assessing the usability of multimodal interaction: the CARE properties. In *Proc. of INTERACT'95*, pages 115–120, 1995.
2. M. Fowler. Presentation model, 2004. Retrieved from: `http://goo.gl/Yx1PCW`, Accessed: 2016-05-17.
3. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.
4. K. Hinckley, M. Czerwinski, and M. Sinclair. Interaction and modeling techniques for desktop two-handed input. In *Proc. of UIST'98*, pages 49–58. ACM, 1998.
5. L. Hoste, B. Dumas, and B. Signer. Mudra: a unified multimodal interaction framework. In *Proc. of ICMI '11*, pages 97–104. ACM, 2011.
6. R. J. K. Jacob, L. Deligiannidis, and S. Morrison. A software model and specification language for non-WIMP user interfaces. *ACM Transactions in Computer-Human Interaction*, 6(1):1–46, 1999.
7. D. Kammer, J. Wojdziak, M. Keck, R. Groh, and S. Taranko. Towards a formalization of multi-touch gestures. In *Proc. of ITS'10*, pages 49–58. ACM, 2010.
8. S. H. Khandkar and F. Maurer. A domain specific language to define gestures for multi-touch applications. In *Proc. of the DSM'10 workshop*, DSM '10, pages 1–6. ACM, 2010.
9. K. Kin, B. Hartmann, T. DeRose, and M. Agrawala. Proton++ : A Customizable Declarative Multitouch Framework. In *Proc. of UIST'12*, pages 477–486. ACM, 2012.
10. B. A. Myers. A new model for handling input. *ACM Transactions in Information Systems*, 8(3):289–320, 1990.
11. M. Potel. MVP: Model-view-presenter the taligent programming model for C++ and Java, 1996. Retrieved from: `http://goo.gl/VBu7Ap`, Accessed: 2016-05-17.
12. D. Romuald and T. Mens. GISMO: A Domain-specific Modelling Language for Executable Prototyping of Gestural Interaction. In *Proc. of EICS'15*, pages 34–43. ACM, 2015.
13. J. Smith. WPF Apps With the Model-View-ViewModel Design Pattern, 2009. Retrieved from: `https://goo.gl/Kn3gCN`, Accessed: 2014-10-08.
14. L. D. Spano, A. Cisternino, and F. Paternò. A Compositional Model for Gesture Definition. In *Proc. of IFIP HCSE'12*, pages 34–52. Springer, 2012.
15. L. D. Spano, A. Cisternino, F. Paternò, and G. Fenu. GestIT: A Declarative and Compositional Framework for Multiplatform Gesture Definition. In *Proc. of EICS'13*, pages 187–196. ACM, 2013.