Designing Graphical User Interfaces Integrating Gestures

François Beuvens
Université catholique de Louvain
Place des Doyens, 1
B-1348 Louvain-la-Neuve, Belgium
+3210478179
francois.beuvens@uclouvain.be

Jean Vanderdonckt
Université catholique de Louvain
Place des Doyens, 1
B-1348 Louvain-laNeuve, Belgium
+3210478525
jean.vanderdonckt@uclouvain.be

ABSTRACT

The world of today and its new technologies like smartphones, tablets, or any flat interaction surface has increasing the need for graphical user interfaces integrating gestural interaction in which 2D pen-based gestures are properly used. Integrating this interaction modality in streamlined software development represents a significant challenge for designers or developers: it requires important knowledge in gestures management, in deciding which gesture recognition algorithm should be used or refined for which types of gestures, or which usability knowledge should be used for supporting the development. These skills usually belong to experts for gesture interaction and not actors usually involved in user interface design process. In this paper, we present a structured method for facilitating the integration of gestures in graphical user interfaces by describing the roles of the gesture specialist and other stakeholders involved in the development life cycle, and the process of cooperation leading to the creation of a gesture-based user interface. The method consists of three pillars: a conceptual model for describing gestures on top of graphical user interfaces and its associated language, a step-wise approach for defining gestures depending on the end user's task, and a software that supports this approach. This method is exemplified with a running example in the area of document navigation.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques – User interfaces. H.5.2 [**Information Interfaces and Presentation**]: User Interfaces – Graphical user interfaces. I.3.6 [**Computer Graphics**]: Methodology and Techniques – Interaction techniques.

Keywords

Method engineering, model-driven architecture, user interface, pen-based gesture, sketch.

1. INTRODUCTION

Gesture-based user interfaces are getting more popular last years with the emergence smartphones, tablets, and any other flat interaction surface that could accommodate pen-based gestures. These new platforms usually require gesture-based interaction with – often but not always – finger or pen as inputs. Despite their recent increased popularity, such user interfaces are considered for a long time and several tools have been realized in order to bring support during their creation.

Pen-based gesture recognition [2,4,14] typically consists in interpreting hand-made marks, called *strokes* [1,7], made with a pointing device (e.g., a mouse, a stylus, a light pen) on a flat constrained vertical or horizontal surface (e.g., a table, a wall or a graphic tablet). Pen-based gestures are applicable to a large area of tasks (e.g., music editing, drawing, sketching, spreadsheets, web navigation, equation editing) in many different domains of activity (e.g., office automation [32], ambient intelligence [10], multimodal systems [30]) and a growing set of devices, ranging from smartphones to tabletop interaction. Pen-based gestures can even be considered across several platforms: starting on a smartphone and finishing on a tabletop [10]. When the locus of input is different from the locus of output (e.g., with a graphic tablet), gestures are drawn outside the main display, thus posing a visual discontinuity problem. When locus of input and output match, a risk of occlusion occurs since the gesture is drawn on top of the main display. The surface used for pen-based gestures is however used as a way to constrain the gesture, thus helping its recognition.

Pen-based gestures have received considerable attention in both research and development, namely for addressing the scientific problem of modeling, analyzing, learning, interpreting, and recognizing gestures in a large spectrum of setups. The large inclusion of pen-based gestures in widely-available interactive applications has however not reached its full potential due to at least the following reasons: designers and developers do not know which recognition algorithm to select from such as large offer, how to tune the selected algorithm depending on their context of use, and how to incorporate the selected algorithm into streamlined User Interface (UI) development in an effective and efficient way. Incorporating pen-based gestures may also involve using Application Programming Interfaces (APIs), libraries, toolkits or algorithm code that could be considered hard to use [31]. Consequently, in this paper, we do not address the scientific problem for modeling, analyzing, interpreting, and recognizing gestures. Rather, we are interested in integrating the results provided by this body of research [3,9,21,22,23,25] into streamlined UI development with device independence, extensibility, and flexibility

2. RELATED WORK

2.1 Motivations for pen-based gestures

Pen-based gestures are appreciated by end users for several reasons: they are straightforward to operate [5], they offer a better precision than finger touch [8], which make them particularly adequate for fine-grained tasks like drawing, sketching. Most people found gestures quick to learn and to reproduce once learned, depending on properties [15,32]:

Iconicity («memorable because the shape of the gesture corresponds with this operation» [17]). When humans are communicating, they are using gestures to increase the understanding of the listener and obviously, the gesture usually means what the speaker is saying. Iconicity principle is based on this. It means that gestures that are designed are close to the interpretation of this reality. For example, Fig. 1(a) depicts the "Delete" action by a pair of scissors, which denotes the activity of cutting something.

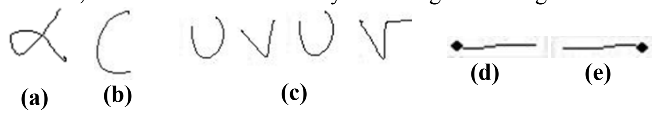


Figure 1. Some pen-based gestures for "delete" (a), "copy" (b), "u" and "v" letters (c), move forward (d), move back (e).

Learnability. Users sometimes forget gestures because they are numerous or because they are complex or not iconic. 90% and more participants held that pen-based gestures with visual meaningful related to commands are easy to be remembered and learned [5]. Another option suggested in [8] to increase the rememberability of the users was to represent a gesture as the first character of the command name. For instance, Fig. 1(b) depicts the "Copy" action because «C» stands for Copy, which is an alternative provided that shortcuts are available. If users spend their time for checking which pen-based gesture is convenient for executing which command in the manual, they will get bored soon.

Recognizability. Naive designers often create pen-based gestures that the computer viewed as similar and thus were difficult to recognize [15]. There is a trade-off between improving the gesture recognizability for the gesture recognizer and optimizing the recognizability for the end user. For instance, Fig. 1(c) compares two alternate designs for writing the "u" and "v" letters. The right one improves the system recognition rate of the system, but deteriorates the end user recognition (since the gesture looks like a square root), while the left one is the opposite: natural for the end user, hard for the system to differentiate.

Compatibility and coherence. Gestures [8] are also better learned and used when they are compatible and coherent. Gestures are best perceived when they are introduced in a uniformed and stable way. For instance, the gesture of Fig. 1(d) depicts "moving forward or right" since the gesture indicated the direction, which is natural to understand for the user, while Fig. 1(e) means the opposite direction.

2.2 Motivations for integration in UI development

In this subsection, some significant related work is reviewed by referring to some of their shortcomings.

Accessibility. Pen-based gesture recognition algorithms are usually made accessible as APIs, libraries, toolkits, platforms or procedural code. A recognition algorithm may become available either in one or many of these heterogeneous forms, thus making their integration in application development a permanent challenge. When a same algorithm is found in different sources, the source choice is even more challenging. Some platforms, like iGesture [26] and InkKit [21,25] offer several algorithms though, but they are more intended for recognition benchmarking.

Hard Algorithm Selection. Certain platforms help the designer to create a gesture set or to benchmark recognition algorithms in a general way. But, to our knowledge, none of them is able to drive the designer through the different steps of selecting a recognition algorithm that is appropriate for a particular gesture set, of fine-tuning its parameters in order to optimize its performance, and for integrating this fine-tuning in UI development.

Satin [3] is a toolkit created for making effective pen-based UIs easier. The two important facets are the integration of pen input with interpreters and the libraries for manipulating ink strokes. Built on top of Satin, *Denim* [6] consists in web site design tool aimed at the early stages of information, navigation, and interaction design. It may be used by designers to quickly sketch web pages, create links among them and interact with them at runtime. *Silk* [4] is an interactive tool that allows designers quickly prototyping a UI by sketching it with a stylus. GART [8] is a UI toolkit designed to enable the development of gesture-based applications. It provides an abstraction to machine learning algorithms suitable for modeling and recognizing different types of gestures. The toolkit also proposes support for the data collection and the training process. GT2k [19] provides a publicly available toolkit for developing gesture-based recognition components to be integrated in large systems. It provides capabilities for training models and allows for both real-time and off-line recognition.

Many gesture recognition algorithms exist and few comparative studies for comparison are published. *DataManager* [13] is a toolkit providing automatic evaluation of gesture recognition algorithms. *iGesture* [14] also helps developers selecting a suitable algorithm for specific needs, and additionally supports them to add new gesture recognition functionality to their application. *InkKit* [11] may be used to perform recognition benchmarking too. *Quill* [7] assists developers in the task of creating well-designed gesture sets, both in terms of computer recognizability and likelihood of confusion among gestures by people.

All these tools provide help during a specific step of the process leading to the creation of a gesture-based user interface. To our knowledge though, none of them is able to fully support all the steps of this process. Following section first outlines the different actors it involves, gives some insights of a system fostering their collaboration, and finally defines a method describing the different steps to be followed by the actors of the user interface creation with the help the system. The two next sections respectively present a model describing the system and give an implementation of this system.

Pen-based gesture recognition may have several finalities. Among them, many tools support the field of document manipulation in different ways. An example is PapierCraft [15], which is pen-gesture-based command system combining graphical and gestural modalities for paper-based interfaces. Fifth section shows through a concrete example how such a system could be easily created through the defined method, with the help of the system. Last section finally concludes and presents future work.

Lack of extensibility and flexibility. It is usually very hard to extend the current implement with another recognition algorithm or another variant of an existing algorithm. Some platforms, like iGesture [26] and InkKit [25], already hold several recognition algorithms with room for extension, but this task is more intended for highly-experienced developers. In addition, they do not help in selecting which algorithm or interpretation of an algorithm is most appropriate for a certain task.

Incorporation in UI development. Most of the time, the incorporation of pen-based gestures should be achieved in a programmatic way, by hand, with little or no support [14]. This is probably one of the most challenging shortcomings. There is no continuity between the design phase of pen-based gestures and the development phase of incorporating the algorithms in the final interactive application. Wilhelm *et al.* describe a trajectory-based approach that is applied to support device independent dynamic hand gesture recognition from pen devices. Feature extraction and recognition are performed on data recorded from the different devices and transformed them to a common basis (2D-space).

This paper is believed to be original in that it provides a method (and a software support) for integrating 2D pen-based gesture interaction in software development of Graphical User Interfaces straightforwardly, with device independence, extensibility, and flexibility.

3. METHOD

Creating a user interface is a complex task requiring many skills. In the perfect case, all these skills would belong to a single developer that would then be able to conduct the whole process of creating the user interface alone. In practice, it is rarely the case since it is difficult for a single developer to acquire all the needed knowledge for each sub-process involved in the user interface creation.

The usual process of creating a user interface can be divided in two distinct parts: during the *conception* phase, the engineer/architect analyzes the user preferences, environment parameters, and miscellaneous requirements, and elaborates the specifications of the interface with the help of the designer. Based on these specifications, the programmer actually codes the user interface during the *implementation phase*.

We are here interested by the specification of user interfaces including gesture or sketch (in the rest of the paper, we will refer to gesture or sketch as equivalent) manipulation. Taking such a feature into account would impact the two phases of the user interface creation. A gesture specialist helps the engineer/architect and the designer during the conception phase. The programmer codes the gesture mechanism based on the specifications during the implementation phase, or is helped by a second programmer specialist in this domain.

We thereby define two phases and four roles characterizing the creation of gesture-based user interfaces:

1. *Conception phase*: outputs a UI description.
    - The **engineer/architect** analyzes the different requirements elicited by the user, the environment, or any other input and identifies the different parts to be included in the user interface (widgets) as well as the behavior enabling interaction with the user and between the widgets. He is taking care of the ergonomics of the system.
    - The **designer** is in charge of the aesthetics of the user interface. His role is to choose the right layout parameters (size, color, font, etc.) for each part of the user interface, and follow aesthetics rules based on metrics such as density [18] or balance [9]. He helps the engineer/architect ensuring good ergonomics.
    - The **gesture specialist** is devoted to the recognition mechanism specification with its different parameters.
2. *Implementation phase*: based on the UI description elaborated in conception phase, outputs the final code of the UI.
    - The **programmer(s)** are the builders of the user interface. Based on the specifications of the conception phase, they actually code it. This includes the recognition mechanism, i.e. the algorithms and the gesture datasets.

The roles defined in the conception phase are well delimited, but in practice, they are more interfering and lead to a real cooperation. Additionally, if the conception and implementation phases remain independent, they will rarely be performed one time each. After the conception and the implementation, the generated user interface is tested, often leading to a conception refinement. The different phases are then repeated until convergence on a satisfying user interface. Our goal is to provide a system supporting the conception and implementation phases, enacting the interaction between the different actors.

At this point, we have considered providing a system that can be ***used*** by engineers/architects, designers, gesture specialists and programmers to create a gesture-based user interface. But how to tune such a system? What widgets, behavior, algorithms, elements of design, etc. should be proposed? We are not specialists and we do not have competences to make any assumptions on what is important or not. Furthermore, technologies are constantly evolving, how to update the system in order to reflect them?

The system should be flexible and extensible to be able to easily acquire new knowledge from skilled people. These people are engineers/architects, designers and gesture specialists, helped by programmers. Instead of being users of the system, they become ***feeders***.

At this level, the role of the programmer(s) is slightly different since it consists in programming the engines allowing the automatic generation of the user interface specification (i.e. every element of knowledge determined by engineers/architects, designers and gesture specialists) in final code, rather than directly programming the user interface by transforming the specifications "manually" (thanks to the direct mapping between the XML specifications and Java code provided by WindowBuilder, presented in Section 5). In this view, the role of the programmer(s) as user(s) of the system is not useful anymore and may then be suppressed for this level.

Around these different descriptions, we are now able to formalize a method enacting the creation of gesture-based user interfaces. For this purpose, we define three different roles acting at three different levels of interaction:

- The **Interface Users (IU)**: end users of the interface.
- The **System Users (SU)**: first group of engineer(s)/architect(s), designer(s) and gesture specialist(s) using the system in order to produce the user interface for the Interface Users.
- The **System Feeders (SF)**: second group of engineer(s)/architect(s), designer(s), gesture specialist(s) and programmer(s) feeding the system with knowledge allowing SU creating user interfaces.

The roles are not exclusive. People acting as System Feeders may be System Users (e.g., a gesture specialists that create an algorithm and then tune and include it in the user interface) or Systems Users may be Interface Users (e.g., engineer(s)/architect (s), designer (s) and gesture specialist(s) wanting to use the interface they have created with the system), etc.

The method is then defined in seven steps:

1. Interface Users define user interface requirements.
2. Based on the UI requirements, System Users define system requirements.
3. If system requirements not met: based on the system requirements, System Feeders feed the system.
4. Based on UI requirements, System Users use the system to produce the user interface.
5. If UI requirements not met: System Users refine system requirements, then go back to step 3.

6. Interface Users use the produced user interface.
7. If Interface Users not satisfied: Interface Users refine UI requirements, then go back to step 4.

When the interface users are satisfied, the method stops iterating. The whole process is illustrated in Figure 2.

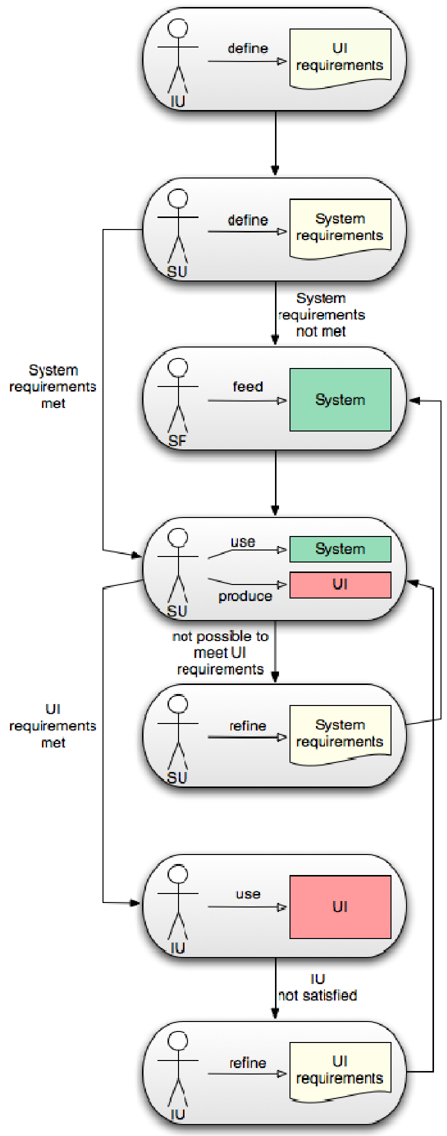


Figure 2. Method.

The main advantage of such a system is its ability to evolve through the different user interfaces creations. It may be viewed as a structured repository of gesture-based user interfaces knowledge. The more user interfaces are produced, the more the system is fed with knowledge, allowing any system users taking profit of all the experience acquired during previous work.

Next section presents the choices we made to model the system on which the method is relying.

4. MODEL

The Cameleon Reference Framework [1] defines the description of a user interface at different levels of abstraction: Tasks & Concepts, Abstract User Interface (AUI), Concrete User Interface (CUI) and Final User Interface (FUI). Each level is formalized by a User Interface Description Language (UIDL) representing the user interface for the specific level. Different mechanisms of forward and reverse engineering allow translating one level into another one.

The user interface description elaborated during the conception phase of the method proposed in previous section actually corresponds to the CUI UIDL. CUI is then transformed in FUI in the implementation phase. This is the modeling approach chosen to support the method.

The CUI UIDL we decided to adopt is Eclipse XML Window Toolkit (XWT) [22]. This language allows specifying user interfaces with XML representation by relying on Standard Widget Toolkit (SWT) [16], which is an open-source widget toolkit for Java. XWT provides a very interesting piece of work, including the description of an important set of SWT widgets as well as a tool supporting the XWT and graphical edition of user interface (see next section). The chosen FUI language is naturally Java since it is directly supported by XWT.

Although XWT constitutes a very useful basis, it is not sufficient to achieve the targeted method. SWT proposes many widgets with design and behavior features. As aforementioned, XWT provides an XML description of an important part of them, but this description only concerns the widgets themselves with their elements of design. The behavior part is not fully described by XWT: although it is possible to specify listening to events, launching an action provided some condition is not possible with XML syntax. Furthermore, no mechanism is devoted to gesture interaction. We then chose XWT as our core model, extended to meet our requirements.

Figure 3 depicts the model including the XWT package, with the behavior extension part (bottom block) and the sketch extension part (top-left block). The three next subsections respectively detail XWT, behavior and gesture parts.

4.1 XWT Package

Only a subset of the XWT model is represented since the whole model is too big and contains useless information at this level. The attributes are not shown for the same reasons.

The main principle of this model is that most entities are derived from the concept of *Widget*, which defines a set of methods and attributes. *Control* and *Scrollable* both add other methods and attributes to *Widget* to increase the available functionalities. These three classes are sub-classed by many concrete classes as *Button, Label,* etc. Among them, three important widgets will be the starting points for the behavior and gesture extensions:

- *Composite*: container of widgets.
- *Canvas*: special container defining a paintable zone.
- *Shell*: root container representing the user interface.

Additionally, *Color* entity is represented since it will be useful for the gesture block.

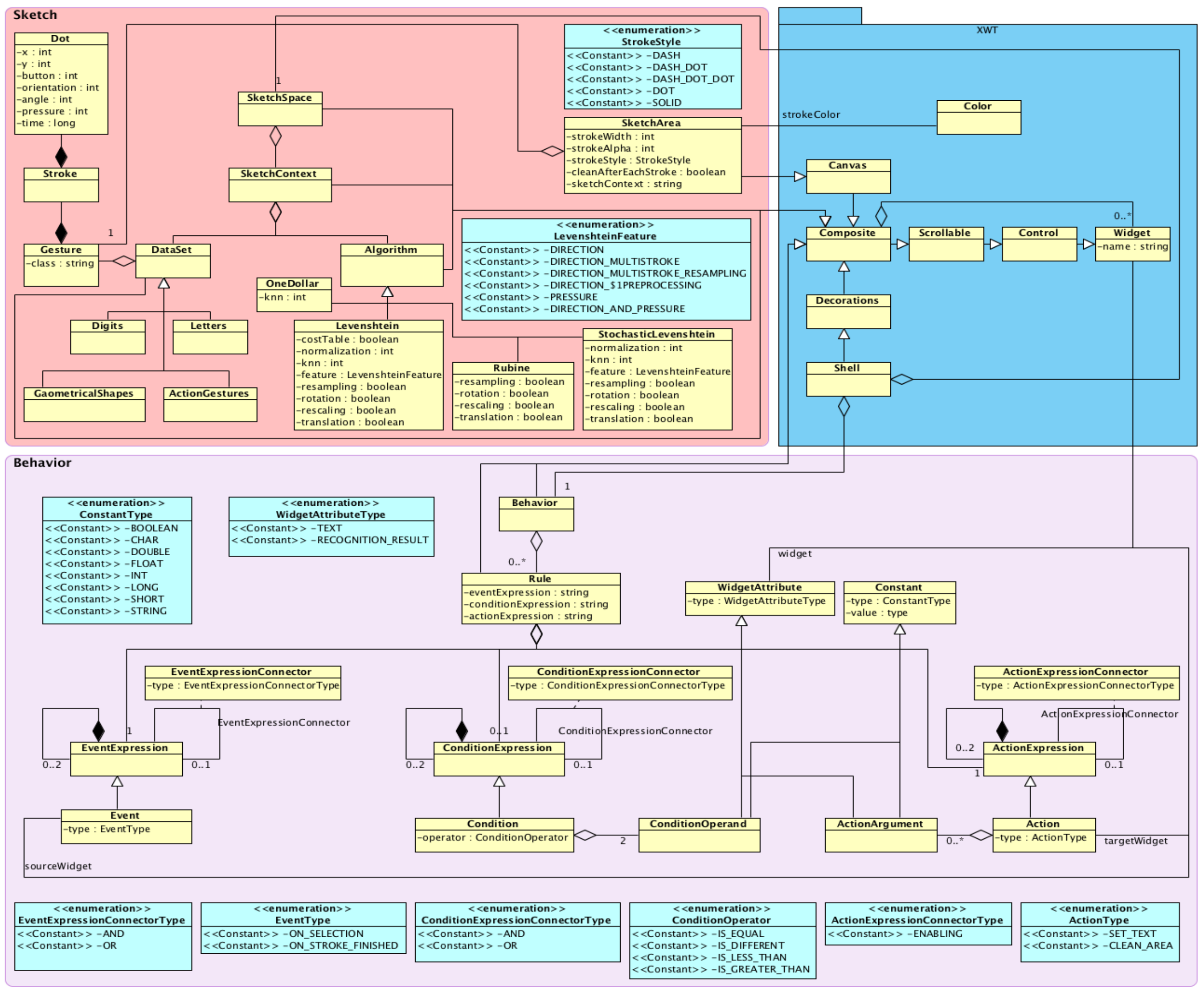


Figure 3. Conceptual model for defining graphical user interfaces and the integration of behavior and gestures.

4.2 Behavior Extension

The extension proposed for the behavior is articulated around two main entities:

- *Behavior*: main behavior entity enclosing all the needed information in a set of *Rule*. It is unique for a user interface and belongs directly to the *Shell.*
- *Rule*: represents an Event-Condition-Action rule (ECA rule). “eventExpression”, “conditionExpression” and “actionExpression” attributes are the string representations of the rules elements described in the followings.

An ECA rule in its basic form allows specifying an action to be launched when an event occur, under a certain condition. In order to add more expressivity, we considered event expressions, condition expressions and action expressions instead of events, conditions and actions. The corresponding entities in the model are *EventExpression, ConditionExpression* and *ActionExpression*.

An event expression is a set of events interconnected by connectors. Additionally, subsets of events may be interconnected with other subsets of events, or simple events. *EventExpression* is either a simple event, or a group of two *EventExpression* interconnected with an *EventExpressionConnector*, which allows modeling the described event expression. The simple event is modeled with *Event* entity, which has a source widget and an event type. There is no explicit constraint between the source widget and the type of event, but the programmer must take care of associating the possible combinations during the implementation phase.

The condition expressions and the action expressions follow, with different types of connectors, the pattern adopted for the event expressions. However, conditions and actions are different from events. A condition is represented by *Condition* entity, which is a couple of two operands linked through an operator. These operands may be either a constant (integer, string, etc.) or a widget attribute (represented by *WidgetAttribute* entity). An action is modeled with *Action* entity, which has a target widget on which the action applies, an action type and a list of arguments to be taken by the action type. There is no explicit constraint between the target widget, type of action and the arguments, but the programmer must take care of associating the possible combinations during the implementation phase.

Let's finally notice that both *Behavior* and *Rule* entities are subclassing *Composite* entity. With this way of modeling –strongly suggested by common practices in XWT and partly imposed by the tool (presented in next section) – the behavior and the rules are considered as being entirely part of the interface. It is usual in XWT, since it allows reusing already defined entities with useful attributes and functionalities. In this case, *Behavior* and *Rule* take profit of the *Composite* functionalities, but have a visibility set to false since they do not need graphical representation in the interface. We only chose these two elements to be explicitly part of the interface because we think specifying the event, condition and action expressions as strings is less constraining, more flexible and more straightforward.

4.3 Gesture Extension

Similarly to behavior extension, gesture extension relies on entities sub-classing *Composite*:

- *SketchSpace*: main sketch entity enclosing all the needed information in a set of *SketchContext*. It is unique for a user interface and belongs directly to the *Shell*.
- *SketchContext*: defines a sketch context gathering different algorithms and datasets.
- *DataSet*: collection of gestures, themselves composed of strokes, themselves composed of dots. Dots gather different information of position, time, and stylus state (if can be applied).
- *Algorithm*: represents the engine able to classify a new gesture, provided the datasets of the sketch context parent.
- *SketchArea*: paintable area linked to sketch context in order to provide recognition feature to the gesture being drawn on it. "sketchContext" attribute is the widget name of the sketch context it is linked to. "strokeWidth", "strokeAlpha" and "strokeStyle" provide control on the stroke aspect, and "cleanAfterEachStroke" indicates if the area must be cleaned each time a stroke is drawn.

As *Behavior* and *Rule*, *SketchSpace*, *SketchContext*, *DataSet*, and *Algorithm* are not intended to be visible in the FUI, while *SketchArea* is a visible component. The other entities are not part of the core description of the gesture extension and will be explained in the next subsection.

4.4 Feeding and Using the System

At this point, we have presented the entities of the core model describing the system aimed at supporting the method. In other words, without more specification this model is "naked", it describes a system ready to acquire any knowledge, but currently empty of knowledge. Increasing this knowledge is the role of the System Feeders. At model level, it corresponds in refining and/or increasing some specific entities in order to describe the features the system supports. The System Users will then use this model as a kind of documentation of the system. The way we expect the System Feeders to extend the model is different for each part.

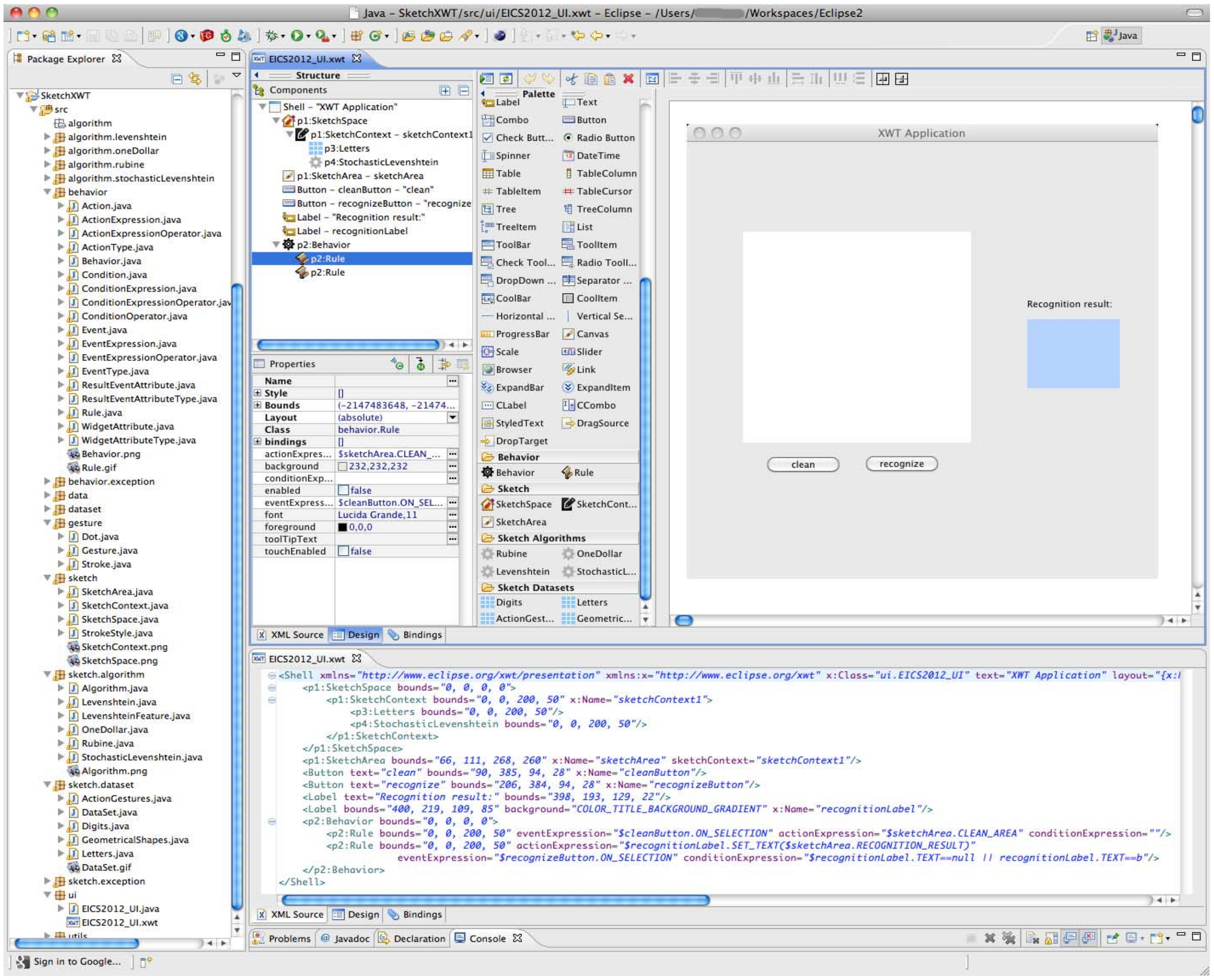
We think the behavior extension allows an advanced expressivity for many rules or composition of rules. However, the types of events, conditions, actions and widgets attributes are missing, as well as the connector types between expressions. For this part, *EventType*, *ConditionOperator*, *ActionType*, *WidgetAttributeType*, *EventExpressionConnectorType*, *ConditionExpressionConnectorType*, and *ActionExpressionConnectorType* are the enumerations to be increased in order to specify the possible events, conditions and actions, and the way they are connected. We already provide some possibilities for each enumeration. They are given as examples and must not be considered as mandatory attributes.

For the gesture extension, the way of adding new knowledge is different. The sketch contexts must actually contain sub-classes of *DataSet* and *Algorithm*, representing specific datasets composed of several *Gesture*, and specific algorithms with their options. Again, we provide some datasets and algorithms examples that we elaborated during a previous experiment [1].

The datasets we provide (digits, letters, action gestures and geometrical shapes) are built from gestures collected from 30 participants through a platform developed for this purpose. Algorithms we provide (Rubine [12], OneDollar [21], Levenshtein [5] and Stochastic Levenshtein [10]) and their extensions constitutes a set of well-known algorithms. They all have a different way of working and cover together many situations with different needs. They are not described here since it is beyond the scope of this paper.

In addition to the behavior and the gesture extensions, System Feeders may add other new functionalities. The XWT widgets may be extended in order to add any new component with a specific purpose. This new component may then be linked to the behavior extension or even the gesture extension if needed.

Next section shows the tool we developed in order to implement the system. It is described by the model presented in this section and supports the method.



Name	
Style	[]
Bounds	(-2147483648, -21474...
Layout	(absolute)
Class	behavior.Rule
bindings	[]
actionExpres...	$sketchArea.CLEAN_...
background	232,232,232
conditionExp...	
enabled	false
eventExpress...	$cleanButton.ON_SEL...
font	Lucida Grande,11
foreground	■ 0,0,0
toolTipText	
touchEnabled	false

```
<Shell xmlns="http://www.eclipse.org/xwt/presentation" xmlns:x="http://www.eclipse.org/xwt" x:Class="ui.EICS2012_UI" text="XWT Application" layout="{x:
    <p1:SketchSpace bounds="0, 0, 0, 0">
        <p1:SketchContext bounds="0, 0, 200, 50" x:Name="sketchContext1">
            <p3:Letters bounds="0, 0, 200, 50"/>
            <p4:StochasticLevenshtein bounds="0, 0, 200, 50"/>
        </p1:SketchContext>
    </p1:SketchSpace>
    <p1:SketchArea bounds="66, 111, 268, 260" x:Name="sketchArea" sketchContext="sketchContext1"/>
    <Button text="clean" bounds="90, 385, 94, 28" x:Name="cleanButton"/>
    <Button text="recognize" bounds="206, 384, 94, 28" x:Name="recognizeButton"/>
    <Label text="Recognition result:" bounds="398, 193, 129, 22"/>
    <Label bounds="400, 219, 109, 85" background="COLOR_TITLE_BACKGROUND_GRADIENT" x:Name="recognitionLabel"/>
    <p2:Behavior bounds="0, 0, 0, 0">
        <p2:Rule bounds="0, 0, 200, 50" eventExpression="$cleanButton.ON_SELECTION" actionExpression="$sketchArea.CLEAN_AREA" conditionExpression=""/>
        <p2:Rule bounds="0, 0, 200, 50" actionExpression="$recognitionLabel.SET_TEXT($sketchArea.RECOGNITION_RESULT)"
                eventExpression="$recognizeButton.ON_SELECTION" conditionExpression="$recognitionLabel.TEXT==null || recognitionLabel.TEXT==b"/>
    </p2:Behavior>
</Shell>
```

Figure 4. The software environment for integrating gestures.

5. TOOL

WindowBuilder [20] is a powerful and easy to use bi-directional Java GUI designer built as an Eclipse plug-in. It supports different toolkits, including SWT and its XML representation XWT. It may be viewed as a visual CUI editor outputting XWT user interface specifications. Moreover, it provides a Java rendering engine for the XWT package described in the previous section. We decided to choose this tool since it fits our requirements by providing a CUI editor and a Java rendering engine to produce the FUI from the CUI. Nevertheless, it must be improved in order to support the behavior and gesture extensions.

5.1 Using the Tool

Figure 4 shows a screenshot of the extended WindowBuilder environment (it may also be seen in action at the following address: http://dai.ly/GJduWS) The top-right part is the visual CUI editor allowing, through the widget palette, manipulating standard SWT widgets as well as behavior and gesture widgets, in order to build the description of a gesture-based user interface. A tree representing the interface is available along with a properties panel useful to specify any option or attribute of a widget. The bottom part of the tool shows the XWT sources of the specification of the interface.

For editing the user interface specifications, System Users have three choices: through the visual CUI editor, by editing the tree representation, or directly by updating the XWT description. In the three cases, the changes are reflected in the other representations.

To build an interface, System Users place widget elements in the visual editor, in the tree, or directly write them in the XWT specification. They define the behavior with one and only one *Behavior* widget in the *Shell* and add a set of *Rule* to it. Each *Rule* may be specified with event, condition and action expressions as String attributes, as described in the model.

The tool adds an interesting feature on the behavior by providing validation. It is used to throw an exception when the System Users specify a malformed or invalid event, condition or action expression. The validation may be turned off via an attribute in the *Behavior* widget in order to let the possibility of describing a different syntax than the one expected by the system. In this case, the running interface provided by WindowBuilder may not work, but in some cases it may be useful to create a specification based on a custom syntax. Concerning the gesture part, System Users are expected to place one and only one *SketchSpace* widget in the *Shell* and add a set of *SketchContext* to it. These *SketchContext* gather one or many *Algorithm* and *DataSet* that can be tuned with the different options described in the model. The *SketchArea* is the sketch element with a visible representation in the interface, allowing the user drawing on it and performing recognition on the gesture drawn. The *SketchArea* has many options, and must be linked to a *SketchContext* through one of them.

The specifications of the user interface are described in the *.xwt* file generated when a new XWT application is created. In order to actually produce the user interface, the System Users have to run the *.java* associated file, which will parse the CUI specifications and render the FUI in Java.

5.2 Feeding the Tool

The left column of the tool provides a view on the sources of the system implementation. Each entity described in the model has a class corresponding to its implementation.

System Feeders may add enumeration constants to *EventType*, *ConditionOperator*, *ActionType*, *WidgetAttributeType*, *EventExpressionConnectorType*, *ConditionExpressionConnectorType*, and *ActionExpressionConnectorType* classes in order to increase the behavior possibilities. The four first ones allow increasing respectively the set of events, conditions, actions and widget attributes, while System Feeders may add new connectors respectively between the events, the conditions and the actions through the three last ones.

The schema of the aforementioned validation feature is set in *EventType*, *ConditionOperator*, *ActionType* and *WidgetAttributeType* classes. Indeed, each constant of these enumerations can optionally take one or several arguments in its constructor, indicating the schema of validation:

- *EventType*: takes a table of *Class* objects as source specification in order to specify the allowed widget types from which the event may be issued.
- *ConditionOperator*: takes a first *Class* object as left operand specification and a second *Class* object as right operand specification to specify the allowed types than may be compared.
- *ActionType*: takes a *Class* object as target specification in order to specify the allowed widget type on which the action may be applied, and a table of *Class* objects as argument specification in order to specify the allowed types for each action argument.
- *WidgetAttributeType*: takes a table of *Class* objects as specification in order to specify the allowed widget types from which the widget attribute may be called.

For the constants of the four enumerations, no schema specification means no restriction. Validation exceptions are thrown if the System Users do not respect these specifications, use a malformed syntax, or try to use inexistent widgets, events, actions, etc.

Feeding the gesture recognition mechanism is done by providing new algorithms or datasets. A new algorithm is added by subclassing the *Algorithm* class and providing it all the wanted options with getters and setters. It is the way WindowBuilder works to show an attribute in the visual editor. New datasets are added by sub-classing *DataSet* class and providing option attributes with getters and setters. The new algorithms and datasets must then be added to the widget palette of the tool (simply by adding a new component to the palette through the contextual menu, and then selecting the newly created class).

At this point, no recompilation of the tool or any further action is required from the System Feeders: the System Users are able to use the new elements of behavior or gesture mechanism in the user interface specification. However, new elements are not implemented for working during the runtime yet. The rendering engines have still to be updated in consequence by the programmer(s) to reflect the new types of events, conditions, actions and widget attributes, the new types of connectors or the new algorithms and datasets. For the behavior part, the programmer(s) must edit the *EventExpression*, *ConditionExpression*, *ActionExpression*, *Event*, *Condition*, *Action* and *WidgetAttribute* classes. Each new feature (new types of event, condition, action, or new types of connections between them) must be handled in one specific method of one of these classes.

For the gesture part, the programmer(s) must modify each new sub-class of *Algorithm* and of *DataSet*. For each algorithm subclass, *train* and *recognize* methods must be implemented in order to train the algorithm and return the string class of the gesture to recognize. For each dataset sub-class, *dataset* method must be implemented in order to return set of *Gesture*. For the implementation we provide, all the logic allowing fulfilling the methods are contained in *algorithm* and *dataset* packages. All the used gestures are provided as text files and located in the *records* folder.

This section has given a description of the few pieces of code to update in order to increase the tool possibilities, the rest of the platform is expected to remain unchanged. It globally shows how the platform is articulated, but does not constitute an exhaustive documentation, which is present in the code.

Next section presents an example showing how the method, model and tool are used to build a pen gesture-based user interface.

6. EXAMPLE

The user interface chosen for the example target document manipulation, and more specifically document navigation. The goal is to show how the method, the model and the tool can be used in order to create a system such as PapierCraft [15], which allows using gesture as commands for manipulating a paper-based interface.

Additionally, at the beginning we assume that the system is empty of any knowledge, i.e. without sub-class of *DataSet* and *Algorithm*, and with empty enumerations in the behavior part. We will start from user interface requirements produced by Interface Users and show how the System Users with the help of the System Feeders will build the user interface by iterating through the different steps of the method.

The user interface for the example has been imagined sufficiently simple to avoid useless information but sufficiently expressive to highlight all the steps defined in the method (Figure 2).

- *Step 1 – IU define UI requirements*

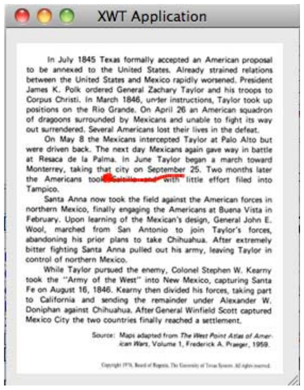
The user interface is a viewer for a document containing several pages. The viewer must have the capability of changing the current page to previous or next page by using gestures.

- *Step 2 – SU define system requirements*

The system must propose a component on which it is possible to specify a background corresponding to the current page of the document, and triggering the recognition of left and right action gestures as the actions for going to previous and next pages. The system requirements are not met then go to step 3.

- *Step 3 – SF feed the system*

Engineer(s)/architect(s), designer(s) and gesture specialist(s) update the CUI UIDL by creating a *ActionGestures* dataset and a *Stochastic Levenshtein* algorithm with options. They create a new *PageViewer* component on top of a sketch area allowing specifying different pages. They additionally update *EventType*, *ActionType*, and *WidgetAttributeType* by providing them respectively *ON_STROKE_FINISHED* (for monitoring the end of an action gesture on the document viewer); PREVIOUS_PAGE and NEXT_PAGE (for modifying the current page on document viewer); *RECOGNITION_RESULT* (for getting recognition result on the document viewer). They add also a *IS_EQUAL ConditionOperator*. The programmer(s) then fulfill the *dataset* method of *ActionGestures*, the *train* and *recognize* methods of *Stochastic Levenshtein*, the *perform* method of *Action* and the *getWidgetAttribute* of *WidgetAttribute* in order to implement the rendering engines reflecting the knowledge introduced in the CUI UIDL.



In July 1845 Texas formally accepted an American proposal to be annexed to the United States. Already strained relations between the United States and Mexico rapidly worsened. President James K. Polk ordered General Zachary Taylor and his troops to Corpus Christi. In March 1846, under instructions, Taylor took up positions on the Rio Grande. On April 26 an American squadron of dragoons surrounded by Mexicans and unable to fight its way out surrendered. Several Americans lost their lives in the defeat.

On May 8 the Mexicans intercepted Taylor at Palo Alto but were driven back. The next day Mexicans again gave way in battle at Resaca de la Palma. In June Taylor began a march toward Monterrey, taking that city on September 25. Two months later the Americans took Saltillo and with little effort filed into Tampico.

Santa Anna now took the field against the American forces in northern Mexico, finally engaging the Americans at Buena Vista in February. Upon learning of the Mexican's design, General John E. Wool, marched from San Antonio to join Taylor's forces, abandoning his prior plans to take Chihuahua. After extremely bitter fighting Santa Anna pulled out his army, leaving Taylor in control of northern Mexico.

While Taylor pursued the enemy, Colonel Stephen W. Kearny took the "Army of the West" into New Mexico, capturing Santa Fe on August 16, 1846. Kearny then divided his forces, taking part to California and sending the remainder under Alexander W. Doniphan against Chihuahua. After General Winfield Scott captured Mexico City the two countries finally reached a settlement.

Source: Maps adapted from *The West Point Atlas of American Wars*, Volume 1, Frederick A. Praeger, 1959.

Copyright 1976, Board of Regents, The University of Texas System. All rights reserved.

Figure 5. Example, first version.

- *Step 4 – SU use the system to produce the UI*

SU put the document viewer component in the UI and specify the different pages. For the behavior, they add a first rule triggered for the event expression "$documentViewer.ON_STROKE_ FINISHED" and, if the condition "$documentViewer. RECOGNITION_RESULT == LEFT", launching action expression "$documentViewer.PREVIOUS_PAGE". This rule express the the action for going back to the previous page. Similarly, they add a second rule for handling the possibility for going to next page. This second rule is triggered by event expression "$documentViewer.ON_STROKE_FINISHED" and, if the condition "$documentViewer.RECOGNITION_RESULT == RIGHT", launching action expression "$documentViewer.PREVIOUS_PAGE". The produced user interface is depicted in Figure 5 (the red line represents a right action gesture).

For the SU, the UI requirements are met then go to step 6.

- *Step 6 – IU use the UI*

IU are not satisfied with produced UI then go to step 7.

- *Step 7 – IU refine the UI requirements*

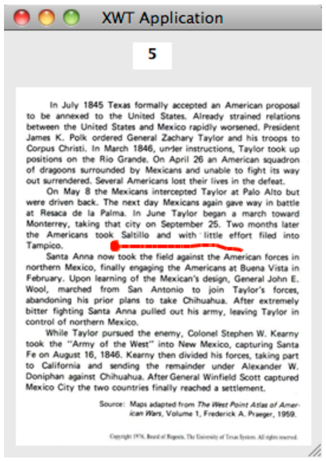
For the IU, even if the recognition is good, it is too slow and should be fastened. They also would like receiving a feedback indicating the current page they are watching.

- *Step 4 – SU use the system to produce the UI*

With the current system, SU are unable to create the user interface respecting the new UI requirements.

- *Step 5 – SU refine the system requirements*

The system needs to propose a faster algorithm. Additionally, a label is required with the possibility to specify a text, corresponding the page currently visualized.



In July 1845 Texas formally accepted an American proposal to be annexed to the United States. Already strained relations between the United States and Mexico rapidly worsened. President James K. Polk ordered General Zachary Taylor and his troops to Corpus Christi. In March 1846, under instructions, Taylor took up positions on the Rio Grande. On April 26 an American squadron of dragoons surrounded by Mexicans and unable to fight its way out surrendered. Several Americans lost their lives in the defeat.

On May 8 the Mexicans intercepted Taylor at Palo Alto but were driven back. The next day Mexicans again gave way in battle at Resaca de la Palma. In June Taylor began a march toward Monterrey, taking that city on September 25. Two months later the Americans took Saltillo and with little effort filed into Tampico.

Santa Anna now took the field against the American forces in northern Mexico, finally engaging the Americans at Buena Vista in February. Upon learning of the Mexican's design, General John E. Wool, marched from San Antonio to join Taylor's forces, abandoning his prior plans to take Chihuahua. After extremely bitter fighting Santa Anna pulled out his army, leaving Taylor in control of northern Mexico.

While Taylor pursued the enemy, Colonel Stephen W. Kearny took the "Army of the West" into New Mexico, capturing Santa Fe on August 16, 1846. Kearny then divided his forces, taking part to California and sending the remainder under Alexander W. Doniphan against Chihuahua. After General Winfield Scott captured Mexico City the two countries finally reached a settlement.

Source: Maps adapted from *The West Point Atlas of American Wars*, Volume 1, Frederick A. Praeger, 1959.

Copyright 1976, Board of Regents, The University of Texas System. All rights reserved.

Figure 6. Example, second version.

- *Step 3 – SF feed the system*

SF will add a label component to the CUI UIDL and implement the original *Levenshtein* algorithm (less good performances but less time consuming), and a *SET_TEXT ActionType* on the label component.

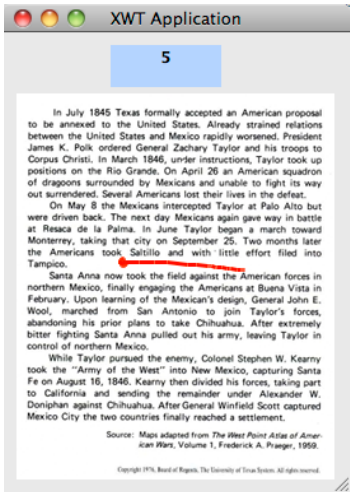
- *Step 4 – SU use the system to produce the UI*

SU are now able to add a label on top of the document viewer in order to indicate the current page. The behavior possibilities have been increased in order to allow linking the page viewer with the label. The generated UI is depicted in Figure 6.

For the SU, the UI requirements are met then go to step 6.

- *Step 6 – IU use the UI*

IU are not satisfied with produced UI then go to step 7.



In July 1845 Texas formally accepted an American proposal to be annexed to the United States. Already strained relations between the United States and Mexico rapidly worsened. President James K. Polk ordered General Zachary Taylor and his troops to Corpus Christi. In March 1846, under instructions, Taylor took up positions on the Rio Grande. On April 26 an American squadron of dragoons surrounded by Mexicans and unable to fight its way out surrendered. Several Americans lost their lives in the defeat.

On May 8 the Mexicans intercepted Taylor at Palo Alto but were driven back. The next day Mexicans again gave way in battle at Resaca de la Palma. In June Taylor began a march toward Monterrey, taking that city on September 25. Two months later the Americans took Saltillo and with little effort filed into Tampico.

Santa Anna now took the field against the American forces in northern Mexico, finally engaging the Americans at Buena Vista in February. Upon learning of the Mexican's design, General John E. Wool, marched from San Antonio to join Taylor's forces, abandoning his prior plans to take Chihuahua. After extremely bitter fighting Santa Anna pulled out his army, leaving Taylor in control of northern Mexico.

While Taylor pursued the enemy, Colonel Stephen W. Kearny took the "Army of the West" into New Mexico, capturing Santa Fe on August 16, 1846. Kearny then divided his forces, taking part to California and sending the remainder under Alexander W. Doniphan against Chihuahua. After General Winfield Scott captured Mexico City the two countries finally reached a settlement.

Source: Maps adapted from *The West Point Atlas of American Wars*, Volume 1, Frederick A. Praeger, 1959.

Copyright 1976, Board of Regents, The University of Texas System. All rights reserved.

Figure 7. Example, final version.

- *Step 7 – IU refine the UI requirements*

IU finally want the label in blue and bigger because they think it is more beautiful.

- *Step 4 – SU use the system to produce the UI*

System already allows addressing this new UI requirement. The color of the label is then changed and the produced user interface is proposed to the IU, which are finally satisfied. This user interface specification may be viewed in the tool screenshot (Figure 4), with the corresponding XWT. The final generated user interface is depicted in Figure 7.

7. CONCLUSION AND FUTURE WORK

In this paper, we proposed a method enacting the creation of gesture-based user interfaces. This method (Figure 2) defines seven steps through which, with the help of the system, different categories of actors will cooperate on the creation of the gesture-based user interface targeted by end users. The system is described by an extension of the XWT model (Figure 3) allowing supporting behavior and gesture missing features. For implementing the system, we relied on a extended version of WindowBuilder (Figure 4). Through a concrete example applied to document manipulation, previous section showed the process of cooperation between the different actors leading to the creation of a gesture-based user interface. If the considered user interface is simple, it is sufficient to highlight the different steps and the way achieving them. The same principles may be applied for more elaborated interfaces. Furthermore, although the method has been developed for handling the specific gesture feature, it may be generalized to take other features into account.

The big advantage of the described system is its ability to acquire and structure new knowledge. It may be viewed as a customizable gesture-based user interface factory: it is expected to evolve through the different creations of gesture-based user interfaces and be refined by System Feeders accordingly to the evolution of new technologies and needs. With time and experience, needs for updating the system will decrease, since it will contain the information needed for most user interfaces. This will lead to an improvement of the process of creating the gesture-based user interfaces. Indeed, the steps for feeding the system will be skipped, letting all the efforts focused on the direct conception of the user interfaces. When this level is reached, only the System Users and the Interface Users are needed to collaborate for producing the interface.

As a next step, we aim at improving the system in such a way that the Interface Users alone would be able to create the user interface. The idea is to rely on recommendation principles. An extension to the system would be developed in order to capture a series of rules defined by conception actors such as engineers/architects, designers and gesture specialists. It would add ”intelligence” to the system, being then able to drive the final users in the creation of their user interfaces, even without any knowledge in ergonomics, UI architecture, design or gesture mechanism.

Although the current implementation of the system only targets Java language and more specifically SWT toolkit, it underlies how the method can be concretized in a general way. Thanks to the separation of concerns between the conception and the implementation phases, the whole part of the system devoted to the conception can be kept whatever the platform targeted. However, in order to implement other languages and hence support other platforms, we aim at adding a translation engine from XWT to UsiXML [17] language. UsiXML is an XML-based UIDL supporting the different levels of abstraction of Cameleon Reference Framework. It defines rendering engines from its CUI to many FUI such as HTML5, Java, etc., and would allow our system outputting final user interfaces in different languages. Relying on UsiXML brings the advantage of reusing many already defined rendering engines, but these rendering engines are sometimes not sufficient to translate the XWT syntax increased by System Feeders engineers/architects, designers and gesture specialists. As UsiXML is extensible, the role of the System Feeders programmers would then become improving UsiXML rendering engines to support new features or languages fitting the increased XWT syntax. With this future integration, we hope even more increasing the System Users experience by avoiding them additional efforts when specifying a user interface for a new language since one specification would be automatically transformable in many languages.

8. REFERENCES

[1] Beuvens, F. and Vanderdonckt, J.: UsiGesture: an Environment for Integrating Pen-Based Interaction in User Interfaces. In Proc. of RCIS’12 , 339-350.

[2] Calvary, G. et al.: A Unifying Reference Framework for Multi-Target User Interfaces. Interacting with Computer 15, 3 (2003), 289-308.

[3] Hong, J. and Landay, J., Satin: a Toolkit for Informal Ink-based Applications. In UIST’00. ACM, 63–72.

[4] Landay, J. A.: SILK: Sketching Interfaces Like Krazy. In: CHI’96, ACM, NY (1996), 389-399.

[5] Levenshtein, V. I.: Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics Doklady 10, 8 (1966), 707–710.

[6] Lin, J., Newman, M., Hong, J., Landay, J.: DENIM: Finding a Tighter Fit Between Tools and Practice for Web Site Design. In: CHI’2000, 2 (1), 510-517.

[7] Long, A.C.J. quill: A Gesture Design Tool for Pen-based User Interfaces. PhD Thesis, Univ. of California at Berkeley, 2001.

[8] Lyons, K., Brashear, H., Westeyn, T., Kim, J.S., Starner, T.: GART: The Gesture and Activity Recognition Toolkit. In Proc. of HCI’07, LNCS, 4552, 718–727.

[9] Ngo, D. C. L., Byrne, J. G.: Another Look at a Model for Evaluating Interface Aesthetics, AMCS 11, 2 (2001), 515-535.

[10] Oncina, J., Sebban, M.: Learning stochastic edit distance: Application in handwritten character recognition. Pattern recognition 39 (2006), 1575–1587.

[11] Plimmer, B., Freeman, I.: A Toolkit Approach to Sketched Diagram Recognition. HCI, Lancaster, UK (2007), 205-213.

[12] Rubine, D.: Specifying gestures by example. SIGGRAPH Computer Graphics (1991).

[13] Schmieder, P., Plimmer, B., and Blagojevic, R. Automatic Evaluation of Sketch Recognizers. In: SBIM’09 ACM Press (2009), pp. 85–92.

[14] Signer, B., Kurmann, U., and Norrie, M. C. iGesture: A General Gesture Recognition Framework. ICDAR’2007, Los Alamitos (2007), 954-958.

[15] Liao, C., Guimbretière, F., and Hinckley, K.: PapierCraft: a command system for interactive paper.

[16] SWT: http://eclipse.org/swt/.

[17] UsiXML: http://www.usixml.org.

[18] Vanderdonckt, J.: Visual Design Methods In Interactive Applications. Mahwah : Lawrence Erlbaum Associates (2003), 187-203.

[19] Westeyn, T., Brashear, H., Atrash, A., Starner, T.: Georgia Tech Gesture Toolkit: Supporting Experiments in Gesture Recognition. Proc. of ICMI’03 (2003), 85-92.

[20] WindowBuilder: http://eclipse.org/windowbuilder/.

[21] Wobbrock, J. O., Wilson, A. D., Li Y.: Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes. In Proc. of UIST’07, 159– 168.

[22] XWT: http://wiki.eclipse.org/E4/XWT.