Towards a Universal Toolkit Model for Structures

Prasun Dewan

Department of Computer Science, University of North Carolina
Chapel Hill, NC 27516, U.S.A
dewan@cs.unc.edu

Abstract. Model-based toolkit widgets have the potential for (i) increasing *automation* and (ii) making it easy to *substitute* a user-interface with another one. Current toolkits, however, have focused only on the automation benefit as they do not allow different kinds of widgets to share a common model. Inspired by programming languages, operating systems and database systems that support a single data structure, we present here an interface that can serve as a model for not only the homogeneous model-based structured-widgets identified so far – tables and trees – but also several heterogeneous structured-widgets such as forms, tabbed panes, and multi-level browsers. We identify an architecture that allows this model to be added to an existing toolkit by automatically creating adapters between it and existing widget-specific models. We present several full examples to illustrate how such a model can increase both the automation and substitutability of the toolkit. We show that our approach retains model purity and, in comparison to current toolkits, does not increase the effort to create existing model-aware widgets.

Keywords: tree, table, form, tab, browser, hashtable, vector, sequence, toolkit, model view controller, user interface management system

1 Introduction

User-interface toolkits strongly influence the nature of a user-interface and its implementation. Programmers tend to incorporate components into a user-interface that are easy to implement. For example, programmers use the buttons directly supported by a toolkit rather than define their own buttons using the underlying graphics and windows package. Moreover, the implementation of the user-interface typically follows the architecture directly supported by the toolkit. For example, in the early versions of the Java AWT toolkit, programmers attached semantics to widgets by creating subclasses of these widgets that trapped appropriate events such as button presses. As the newer version of AWT supports delegation, programmers now associate callbacks with these widgets.

One of the major recent advances in toolkits is support for model-aware widgets, that is, widgets that understand the interface of the semantic or model object being manipulated by them. Model-aware widgets have the potential for (i) increasing *automation* and (ii) making it easy to *substitute* a user-interface with another one. Current toolkits, however, have focused only on the automation benefit as they do not allow different kinds of widgets to share a common model. For example, in Java's Swing toolkit, the JTable and JTree model-aware widgets understand different kinds

of models. As a result, it is not possible to display the model of a `JTable` widget as a tree, and vice versa.

Therefore a data structure that serves as a universal model for different widgets is an attractive idea. It is not possible to develop such a model for all possible widgets as some widget models assume fundamentally different semantics. For example, the model of a slider must be a numeric value and not, for example, a string or a list. In this paper, we show that is possible, however, to develop a universal model for all existing structured model-unaware widgets and several new structured components such as browsers for which no appropriate model interface has been defined so far. Thus, such a universal structured model increases both the automation and substitutability of the toolkit. It increases automation as it directly supports user-interface components such as browsers that have to be manually composed today. It increases substitutability as it allows the model to be displayed using any of the existing and new model-aware structured-widgets.

In the rest of the paper, we expand on this idea. We first show the relationship between the MVC (Model-View-Controller) architecture [1] and model-aware widgets. Once this relationship is understood, then the substitutability limitation of current toolkits becomes apparent. We then present requirements of a universal structured-model. Next we take a top-down approach to identifying such a model based on the work done in programming languages, operating systems and database systems that support a single data structure. We then do a bottom-up analysis of this model by exploring how it could be attached to existing and new structured user-interface components, extending it as necessary. We end with conclusions and directions for future work.

2   MVC and Toolkit Widgets

The MVC framework, as presented in [1], requires the semantics of a user-interface to be encapsulated in a model, the input processing to be performed by one or more controllers, and the display to be defined by one or more views. In response to an input command, a controller executes a method to write the state of the model, which sends notifications to the views, which, in turn, read appropriate model state, and update the display.

One issue not explicitly addressed by MVC, or any other paper with which we are familiar, is: what is the relationship between MVC and toolkits? The architecture could be implemented (i) from scratch, without using a toolkit, (ii) using model-unaware widgets, or (iii) using model-aware widgets. As (i) does not inform toolkit design – the focus of this paper – let us ignore this approach. To contrast (ii) and (iii), we must precisely distinguish between model-aware and model-unaware widgets.

A model-unaware widget talks to its client in a syntax-centric language. It defines calls allowing the widget client to set its state in display-specific terms, and sends notifications to the client informing it about changes to the state, again in display-specific terms. For example, a model-unaware text-box displaying a Boolean value talks to its client in terms of the text it displays. It defines calls that allow the client to set the text and sends notifications informing the client about changes to the text. A

model-aware widget, on the other hand, talks to its clients in a semantics-centric language. It receives notifications regarding changes to the client state in model-based terms, and converts these changes to appropriate changes to the display. When the display changes, the widget calls methods in the client to directly update its state. For example, a model-aware text-box displaying a Boolean value would talk to its client in terms of the Boolean it displays. When the user edits the string, it directly updates the Boolean, and conversely, it responds to a notification by automatically converting the Boolean to a string.

Given model-unaware widgets, Figure 1(a) shows how the user-interface *should* be implemented and Figure 1(b) shows how it *can* be implemented. In Figure 1(a), the view translates a model notification into an operation on the widget; and the controller translates a widget notification to a call in the model. In Figure 1(b), the widget client is a monolithic application that performs semantics, input and output tasks. Often, programmers follow the architecture directly supported by a toolkit, which in this case means that the architecture shown in Figure 1(b) is used, resulting in a spaghetti of callbacks [2] mixed with semantics.
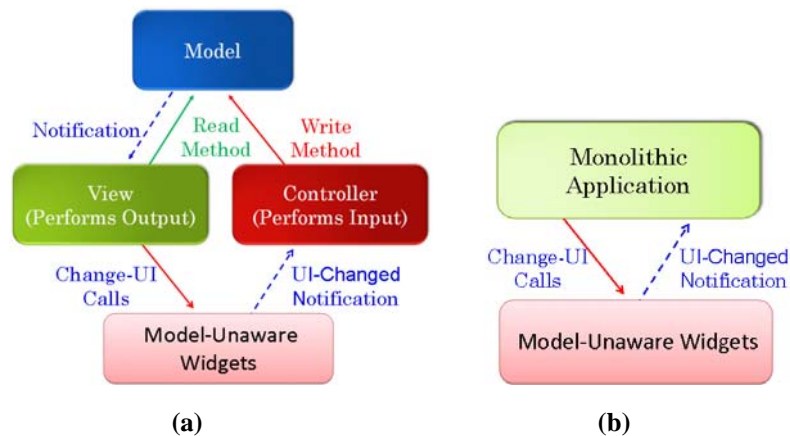

(a)   **(b)**

**Figure 1** Using model-unaware widgets with (a) and without (b) MVC

This problem does not, of course, occur with model-aware widgets. These widgets do not directly support the MVC architecture. Instead, they support a model-editor architecture (called subject-view in [3]), in which the editor combines the functionality of a view and controller, receiving notifications from the model and calling both read and write methods in the model. A model-aware widget is essentially an editor automatically implemented by the toolkit that is based on some model interface. As it is based on an interface rather than a class, it can be reused for any model class that implements the interface, as shown in Figure 2(a). It is this model substitutability that increases the automation of the toolkit – for all models displayed using the widget, no UI code needs to be written.

Model substitutability was not an advertised advantage of the original MVC framework, which, as mentioned earlier, did not address toolkits or automation. This substitutability is the dual of the UI/editor substitutability for which the MVC architecture was actually created, which is shown in Figure 2(b). Given a model, it is

possible to attach multiple editors to it, concurrently or at different times. Attaching a new editor to a model does not require changes to the model or other editors – the only requirement is that the editor understand the model interface. Thus, given a model displayed as a bar-chart, adding an editor that displays it as a pie-chart does not require changes to the model or the existing editor.

While toolkits have made an important advance to the MVC architecture by using it for automation, as designed currently, they have done so by sacrificing the original advantage of the architecture. The reason is that different editors supported by a toolkit assume different model interfaces. For example, the tree and table widgets in Swing assume different models. As a result, it is not possible to display the same model as a tree and/or a table. It is possible to display a tree or table model using a programmer-defined user-interface, but that involves sacrificing automation. The Windows/Forms toolkit has a similar problem. As our implementation is based on Java, we shall focus only on the Java Swing toolkit in the remainder of the paper.

What is needed, then, is a technique that combines both kinds of substitutabilities, which is shown in Figure 2(c). Here, a toolkit-provided editor can be attached to instances of multiple model classes. In addition, a model can be attached to instances of multiple editor classes. In the next section, we describe what this means in more depth.
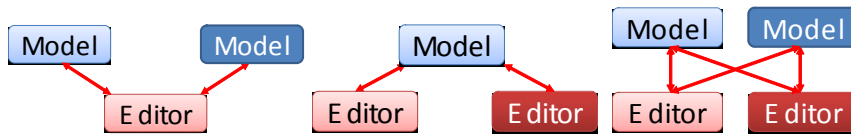
(a) Toolkit Model Substitutability (b) MVC UI Substitutability (c) Model/UI Substitutability
Figure 2 Three forms of substitutability possible with model-aware widgets

3 Requirements

To remove the limitations of previous work mentioned above, we need a new toolkit design that meets the following requirements:

1. *Reduced model set:* The current set of models should be replaced with a smaller set of models.
2. *Same or increased model-aware widget sets:* The set of model-aware widgets automatically supported by the toolkit should not be reduced.
3. *Same or decreased programming effort:* It should not be harder to create models and bind them to existing editors.
4. *Model purity:* The models must have only semantic state.

It is important to meet all of these requirements. It is easy to meet the first requirement by, for instance, simply eliminating the table model from Swing. However, this approach does not meet the second requirement, as the set of model-aware widgets is also reduced. It is easy to meet both requirements by requiring a model to implement the interfaces of multiple existing model-aware widgets. For instance, combining the model interfaces defined by the tree and table widgets

reduces the set of model interfaces, but requires programmers using the interface to implement both sets of methods, instead of only one of the sets, which does not meet the third requirement. Existing "models" in toolkits sometimes have user-interface information. For example, the `JTable` model indicates the label to be used as a column name. Therefore, we have put the fourth requirement to ensure the purity of models. It is possible to meet the first three requirements to different degrees depending on the extent to which the (1) model set is reduced, (2) set of model-aware widgets is increased, and (3) programming effort is changed. In the following sections, we present an approach that meets these requirements and evaluate it based on the above metrics.

4   Top-Down Identification of a Universal Structured Model

The ideal approach to meeting the above requirements is to define a universal model for all widgets. However, as mentioned before, it is not possible to develop such a model as there are widget models with fundamentally different semantics. Thus, we must set our sights lower and aim simply for a reduced model set rather than a single model.

There are well known techniques for reducing the model set in existing toolkits. Previous work has shown how a model can be mapped to multiple *unstructured-widgets* [4, 5], that is, widgets displaying a single editable atomic value. In particular, a discrete number can be mapped to a slider or textbox, an enumeration can be mapped to combobox or textbox, and a Boolean can be mapped to a textbox, combobox, or checkbox. These techniques are gradually being implemented in existing toolkits. However, there has been no work for mapping a model to multiple *structured-widgets* such as tables and trees, which display composite (non-atomic) values. Therefore, we will focus only on such widgets in this paper.

Can we define a single universal model for all model-aware structured-widgets supported so far? If so, can it also be bound to other user-interface components that are not automatically supported by existing toolkits? These are the two questions we address in this paper. While they have not been addressed before in the user-interface arena, analogous questions have been posed in other fields such as database management systems, operating systems, programming languages, and integrated systems.

Research in database management systems has tried to determine if a single data structure can be used to store all data that must be searched. A practical answer has been the relational model [6]. Similarly, research in operating systems has tried to determine if a single data structure can be used to store all persistent data, and a practical answer has been the Unix "file", which models devices, sockets, text files, binary files, and directories. Research in programming language has tried to answer an even more complex question: can a single structured object be used for all computation? The answer in Lisp (and later functional languages such as ML) is an ordered list, and in Snobol (and later string processing languages such as Python) a hashtable. Designers of EZ [7] have proposed using a nested hashtable as the only structured object in a programming language that is integrated with the underlying

operating system. For example, a directory is simply a persistent table, and changing to sub directory, *sd*, corresponds to looking up the table value associated with key *sd*.

Of course, the reduced abstraction set is not without limitations. Therefore, object-oriented database management systems have been proposed as alternatives to traditional relational systems; IBM has supported structured files in its operating system (an idea that was supposed to be extended by the Longhorn Microsoft operating system); and object-oriented languages are preferred today to Lisp and Snobol. It is for this reason that we have added the other three requirements in addition to the requirement of a reduced model set. If we meet all four requirements, we improve the state of the art without introducing any limitations.

We mention the research in other fields to motivate a top-down search for a universal structured model that is based on data structures that have been found to be sufficient for defining a variety of semantic state, which is the kind of state managed by a model. The alternative is a bottom-up approach in which we try to generalize models of existing structured-widgets. As the nature of the models should be independent of the nature of user-interfaces, the result of the top-down approach seems more likely to last in the long-run. In particular, as it is not based on specific user-interfaces, it should make it possible to automatically support new kinds of structured-widgets. On the other hand, this approach does not distinguish between displayed and internal semantic state. The second approach can identify aspects of displayed semantic state not captured by existing display-agnostic data models.

For these reasons, we take an approach in which we: (1) first use the top-down approach of creating an interface that models the universal semantics structures proposed in other fields; (2) and then take the bottom-up approach of generalizing this interface to connect it to existing model-aware widgets.

The first step above requires an interface that combines elements of relations, nested hastables, and lists. A relation is simply a set of tuples, where each tuple is a record. Thus, we can reduce the above goal to supporting records, un-ordered sets, ordered lists, and nested hashtables.

As we are developing a Java-based tool, let us start with an interface containing a subset of the methods implemented by the Java Hashtable class:

```
public interface UniversalTable <KeyType, ElementType>{
    public Object put(KeyType key, ElementType value);
    public Object get(KeyType key);
    public Object remove(KeyType key);
    public Enumeration elements();
    public Enumeration keys();
}
```

This interface completely models a hashtable because it has methods to (a) associate an element with a key, (b) determine the element associated with a key, and (c) remove a key along with the associated element. The interface is parameterized by the types of the keys and elements. As the element types can themselves be tables, this interface also models nested hashtables of the kind supported by EZ. The last two methods in the interface seem to have been added by Java for purely convenience reasons – they make it possible to treat a hashtable as a pair of collections accessed using CLU-like iterators [8]. However, as we show below, they also allow the interface to model records, ordered lists, and sets.

A record is simply a table with a fixed number of keys. Thus, a record implementation of this interface simply initializes the table with the fixed number of keys and does not let keys to be added or deleted. This is illustrated in the following class, which defines a subset of the contents of an email message-header:

```java
//simulating a record whose fields are not ordered
public class AMessage implements UniversalTable<String, String> {
 Hashtable<String, String>  contents = new Hashtable();
 public final static String SUBJECT = "Subject";
 public final static String SENDER = "Sender";
 public final static String DATE = "Date";
 public AMessage (String theSubject, String theSender, String theDate){
    put(SENDER, theSender);
    put(SUBJECT, theSubject);
    put(DATE, theDate);}
 public Enumeration keys() {return contents.keys();}
 public Enumeration elements()  {return contents.elements();}
 public String get (String key) {return contents.get(key);}
 public Object put(String key, String val) {
   if (contents.get(key) != null)return contents.put(key, val);
   else return null; // record keys are fixed
 }
 public String remove (String key) {return null;}
}
```

The above class defines a record consisting of three fields named "Subject", "Sender" and "Date", and defines a constructor that initializes the value of these fields.

The two iterator-based methods can be used to model an ordered list. The return type, Enumeration, of these methods, is given below:

```java
public interface Enumeration{
   public boolean hasMoreElements();
   public Object nextElement();
}
```

As we see above, this type defines an order on the elements to which it provides access. Thus, the keys() and elements() methods of our universal table can be used to define an order on the keys and elements, respectively, in the table. The class, AMessageList, given on the next page, illustrates this concept. Like the previous example, this class stores the mapping between keys and elements in an instance of the Java Hashtable class. However, unlike the previous class, it does not return these values in the order returned by the underlying Hashtable. Instead, it uses two vectors, one for keys and another for elements, to keep track of the order in which these values are added to the table, and returns them in this order. If a key is associated with a new element, then the new element takes the position of the old element associated with the key. When a key is removed, the key and the associated element are removed from the vectors storing them. As this code is somewhat complicated, we have incorporated it in a generic list class that is parameterized by the key and element type and implements UniversalTable. As a client may wish to insert rather than append components, we add another put method to the universal table interface that takes the position of the key and element pair as an additional argument:

```
    public Object put(KeyType key, ElementType value, int pos);
```

A set can be more simply modeled by overriding the put method to not replace the value associated with a key. Thus, we have been able to use a single interface to simulate four important structures: nested hashtables, records, ordered lists, and sets. Interestingly, we have done so by using a subset of the methods of an existing class – the Java Hashtable.

```java
// simulating an ordered list
public class AMessageList
        implements UniversalTable<String, AMessage>{
    Hashtable<String, AMessage> contents = new Hashtable();
    Vector<String> orderedKeys = new Vector();
    Vector orderedElements = new Vector();

    public Enumeration keys() {
        return orderedKeys.elements();
    }
    public Enumeration elements()  {
        return orderedElements.elements();
    }
    public AMessage get (String key) {
        return  contents.get(key);
    }
    public AMessage put (String key, AMessage value) {
        AMessage oldElement = contents.get(key);
        AMessage retVal = contents.put(key, value);
        if (oldElement == null) {
            orderedKeys.addElement(key);
            orderedElements.addElement(value);
        } else {
            int keyIndex = orderedKeys.indexOf(key);
            orderedElements.setElementAt(value, keyIndex);
        }

        return retVal;
    }
    public AMessage remove (String key) {
        int keyIndex = orderedKeys.indexOf(key);
        if (keyIndex != - 1) {
            orderedKeys.remove(keyIndex);
            orderedElements.remove(keyIndex);
        }
        return contents.remove(key);
    }
}
```

Finally, to make our universal table a model that can notify editors/views and other observers, we add the following methods to `UniversalTable`:

```
public void addUniversalTableListener(UniversalTableListener l);
public void removeUniversalTableListener(UniversalListener l);
```

A listener of the table is informed about keys being put and removed:

```java
    public interface UniversalTableListener {
        public void keyPut(Object key, Object value);
        public void keyRemoved(Object key);
    }
```

5 Binding Universal Model to Structured-Widgets

Let us now take the bottom-up approach of determining if instances of the universal table can serve as models of two existing Swing structured model-aware widgets: `JTree` and `JTable`?

Let us first consider `JTree`, which has several requirements:

1. Its model must be decomposable into a tree,
2. Both the internal and leaf nodes should have data items associated with them.
3. The node data items should be editable, that is, it should be possible to add and remove children of composite tree nodes, and modify the data items of all nodes.

To meet requirement 1, we must be able to decompose an instance of a universal table into component objects. The instance can be decomposed into its (a) key objects, (b) element objects, and (c) key and element objects (Figure 3).

We provide a special call that can be used by the programmer to make this choice for a specific application class, as shown below:

```
ObjectEditor.setChildren(AMessageList.class, ELEMENTS_ONLY);
ObjectEditor.setChildren(AMessageList.class, KEYS_ONLY);
ObjectEditor.setChildren(AMessageList.class, KEYS_AND_ELEMENTS);
```
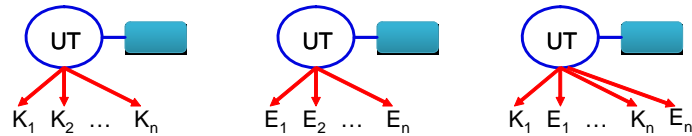

Figure 3 Three alternative approaches to decomposing a universal table

These calls tell the toolkit to decompose instances of `AMessageList` into its elements, keys, or keys and elements. If a key or element is also a universal table, then it too can be decomposed in any of the three ways. In the case of `AMessageList`, each element is an instance of `AMessage`, which implements `UniversalTable`. Therefore, it too can be decomposed into sub-objects. Figure 4 shows the decompositions defined by the following calls:

```
ObjectEditor.setChildren(AMessageList.class, ELEMENTS_ONLY);
ObjectEditor.setChildren(AMessage.class, ELEMENTS_ONLY);
ObjectEditor.setChildren(AFolder.class, KEYS_ONLY);
```
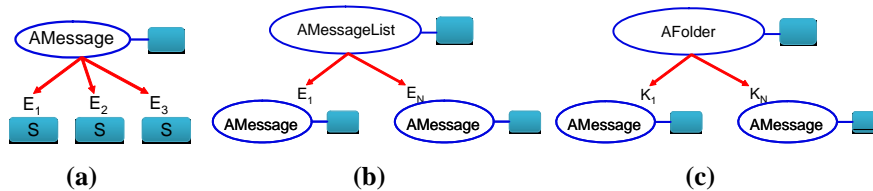

(a)   **(b)**   **(c)**

Figure 4 Decomposing three example universal tables into components

Here, `AFolder` is a universal table with keys of type `AMessage` and elements of type `String`, mapping message-headers to the corresponding message texts:

```
public class Folder implements UniversalTable<AMessage, String>
```

Thus, `AFolder` and `AMessageList` are duals of each other in that the key type of one is the element type of the other. In Figure 4, an empty box is attached to an internal node to denote its data item, and a box with label S is used to denote a leaf node of type `String`.

By default, a table is decomposed into its elements. A programmer can define the default decomposition for all universal tables by using the following call:

```
ObjectEditor.setDefaultHashtableChildren(KEYS_ONLY);
```

Let us now consider the second requirement of associating the tree nodes with data items. We could simply use the approach used by `JTree` of assuming that the `toString()` method of a tree node defines the value. However, to support form user-interfaces, we use a more complex approach described by the following routines:

```
Object getTreeDataItem(node) {
    if (getLabel() != "")
        if (node is leaf)
                return getLabel(node) + ":" + node.toString()
        else // node is element
                return getLabel(node)
    else // label = ""
        return node
String getLabel (node) {
    if node is labelled and label is defined
        return label
    else if (node is labelled and node is element) // label not defined for element
        return getTreeDataItem( key associated with element).toString()
    else // label not defined for key
        return ""
```

This algorithm is motivated and illustrated by the tree displays of `AMessage`, `AMessageList`, and `AFolder` shown in Figure 5.
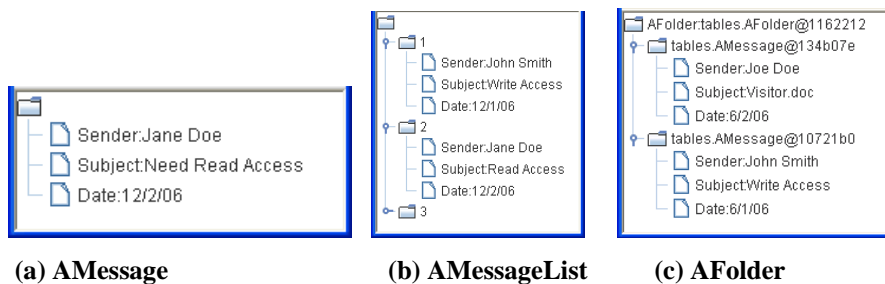

(a) AMessage   **(b) AMessageList**   **(c) AFolder**

Figure 5 Associating model items with tree nodes

In Figure 5(c), none of the classes has overridden the `toString()` method, while in Figures 5(a) and 5(b), `AMessage` and `AMessageList` have overridden this method to return the null string. In all cases, the labeled attribute is true and the default label is the null string. In Figure 5(b), the data items associated with the `AMessage` elements are their keys: "1", "2" and "3". In Figure 5(c), the data items associated with the

`AMessage` keys are the values returned by their `toString()` methods. ObjectEditor provides routines to set the values of the labeled and label attributes. For example, the following call says that, by default, the value of the labeled attribute is false:

```
ObjectEditor.setDefaultLabelled(false);
```

Similarly, the following call says that the value of the labeled attribute for instances of type `AMessage` is true:

```
ObjectEditor.setLabelled(AMessage, true);
```

The exact algorithm for determining the data item of a node can be expected to evolve – what is important here is that it depends on a programmer-specified label and takes into account whether the node is a key, element, leaf, or composite node.

Now consider the requirement of allowing nodes to be editable. Inspired by Java's `MutableTreeNode` class, we add the following method to `UniversalTable` to allow its data items to be changed:

```
public void setUserObject(Object newVal);
```

The following code shows what happens when a node's data item is changed:

```
Object edit(node, newValue) {
    if node is composite
        node.setUserObject(newValue)
    else if node is key // leaf key
        parent_of_node.put (newValue, parent_of_node.get(old key));
        parent_of_node.remove (oldKey);
    else // leaf element
        parent_of_node.put (key_of_node, newValue);
```

Editing the data item of a composite node results in the `setUserObject()` method to be called on the node with the new value. Editing the data item of a leaf element results in the key associated with the element to be bound to the new value. Thus, in Figure 5(a), changing "Jane Doe" to "Jane M. Doe" results in the "Sender" key to be associated with "Jane M. Doe". Editing the data item of a leaf key results in the element associated with the old key to be associated with the new key. Thus, in Figure 5(b), changing the key "1" to "One" associates the first message with "One" instead of "1".

The following code shows what happens when a new node is inserted into a composite node at position index:

```
insert (parent, child, index)
    if (keysOnly(parent))
        parent.put (child, node.defaultElement(child), index);
    else if (elementsOnly(parent))
        parent.put (node.defaultKey(child), child, index);
    else if (keysAndElements(parent))
        if (isKey(parent, child, index) or index == size ) // inserting before key or at end
            parent.put (child, node.defaultElement(child), index/2);
        else // inserting before element
            parent.put (node.defaultKey(child), child, (index – 1)/2);
```

Based on the position of the inserted element and how the parent of the inserted element has been decomposed, the code determines if a key or element is to be

inserted, and calls methods in the parent to determine the default key or element to serve as the new child. The `isKey()` method determines if the new node is a key based on the insertion position. The code assumes two new methods in the universal table interface:

```
public KeyType defaultKey(ElementType element);
public ElementType defaultValue(KeyType key);
```

These two methods are needed only because the universal table constrains the types of its key and elements. If it were to accept any object as a key or element, the toolkit could simply create a new object as a default key or object:

```
new Object()
```

The operation to remove a node is simpler.

```
remove (parent, child)
    if isKey (child) parent.remove (child) else parent.remove (key of child)
```

Finally, ObjectEditor provides a way to specify that a universal table should be displayed as a tree:

```
edit (UniversalTable model, JTree treeWidget);
```

This operation displays the model in `treeWidget`. Here, the programmer explicitly creates the tree widget, setting its parameters such as preferred size as desired. We also provide the operation:

```
treeEdit (UniversalTable model)
```

which creates the tree widget with default parameters. Sometimes a whole class of objects must be displayed using a particular kind of widget, so the following operation is also provided:

```
setWidget (Class universalTableClass, Class widgetClass)
```

This call tells the toolkit to always display an instance of `universalTableClass` using an instance of `widgetClass`.

Thus, we have met all of the requirements imposed on us by the Swing tree widget. Let us consider now the Swing table widget. This widget needs the following information: (1) a two dimensional array of elements to be displayed; (2) the most specific class of the elements of each column; (3) the names of the columns; and (4) whether an element is editable.

The first requirement can be met by a non-nested or nested universal table. A one-level universal table (that is a universal table whose children are leaf elements) is considered a table with a single row or column based on whether its **alignment** is **horizontal** or **vertical**, respectively. A two-level universal table (that is a table whose children are one-level tables) decomposed as keys only (elements only) is straightforwardly mapped to a table in which a row is created for each key (element) of the table consisting of the components of the key (element). A universal table decomposed as keys and elements whose keys are leaf values and elements are 1-level universal tables is decomposed into a table in which a row is created for each key of the object consisting of the key and children of the corresponding element. Currently, we do not map other universal tables to table widgets. The second requirement above is met by returning the class of the default element/key depending on how the table

has been decomposed into children. As column names can sometimes be automatically derived from the semantics of the model, but should not be defined explicitly by the model, we use the following algorithm for determining them:

```
getColumnName(root, columnNum)
        if numRows (root) > 0 return firstRow(root).column(columnNum).getLabel();
        else return "";
```

If the matrix is not empty, it then uses the `getLabel()` operation defined earlier to return the label of a particular column in the first row. Recall that the operation returns a value based on the key of an element and the label attribute of the element. To meet the last requirement of `JTable`, we provide the following methods inspired by the Swing `JTableModel` class:

```
public boolean isEditableKey(KeyType key);
public boolean isEditableElement(ElementType element);
public boolean isEditableUserObject();
```

Figure 6(a) illustrates our schemes for meeting the requirements above using an instance of a `AMessageList`. Here, `AMessageList` is decomposed into keys and elements, `AMessage` is decomposed into elements, the keys of `AMessageList` are not labeled, and the elements of `AMessage` are labeled but have no explicit label set by the programmer. As a result, each row consists of the atomic String key, and the atomic elements of `AMessage`; and the keys of the elements of `AMessage` are used as column names but not displayed in each row. As in the tree widget case, we provide routines to bind a table widget to a model.
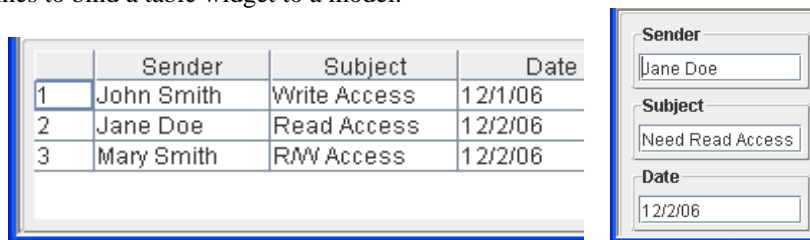

Figure 6 Table and Form Displays

The fact that a universal table models a record implies that we can also support forms, as these have been previously created automatically from database records [9]. However, database records (tuples) are flat. As universal tables are nested, we can create hierarchical forms. In fact, we can embed tables and trees in forms. Figure 7 shows a table embedded in a form. Here, we assume `AFolder` is decomposed into its keys, and `AMessage` is decomposed into keys. The algorithm for creating a form is:

```
displayForm (node) {
    panel = new Panel
    setLabel (panel, getLabel(node)) // can put label in the border, add a label widget, ….
    for each child of node
        childPanel = display (child)
        add (panel, childPanel)
    return panel
```

The operation `display(node)` returns a component based on the widget associated with the type of `node`. For a universal table, the widget is a form, tree, tabbed pane, or table. For an atomic type, it is an atomic widget such as a slider, combo-box, text-box or checkbox. The algorithm leaves the layout of children in a parent panel as implementation defined. In [10], we define a parameterized scheme for arranging form items in a grid.

Tabbed panes are similarly implemented:

```
displayTabbedPane (node) {
    tabbedPane = new tabbed pane;
    for each child of node
        childPanel = display (child)
        add(tabbedPane, childPanel, getLabel(child))
    return panel
```

Figure 7(b) shows the tabbed display for folder displayed in 7(a).

Universal tables are ideal for creating browsers, which are common-place, but have not been automatically supported by any user-interface tool. To create a browser, the ObjectEditor provides the following call:

```
edit (UniversalTable model, Container[] containers);
```

If the array, `containers`, is of size n, this call creates an n-level browser. A browser always decomposes a universal table into its keys. The top-level model is displayed in `container[0]`. When a key is selected in `container[i]`, it displays the associated element in `container[i+1]`, where $0 <= i < n$. Figure 8 illustrates this scheme. Here, a three-level browser has been requested, and the top-level model is an instance of the class `AnAccount`, whose keys are strings and elements are of type `AFolder`:

```
public class AnAccount implements UniversalTable <String, AFolder>
```

`AnAccount` has been bound to a tree widget, and `AFolder` to a table widget. The container array passed to the edit routine above consists of the left, top-right, and bottom-right windows, in that order. The toolkit shows the two `String` keys of the top-level model in the first container. Selecting the first `String` key in this container results in the associated folder element being displayed in the second container. Selecting one of the `AMessage` keys of this folder results in the associated `String` element to be displayed in the third container.
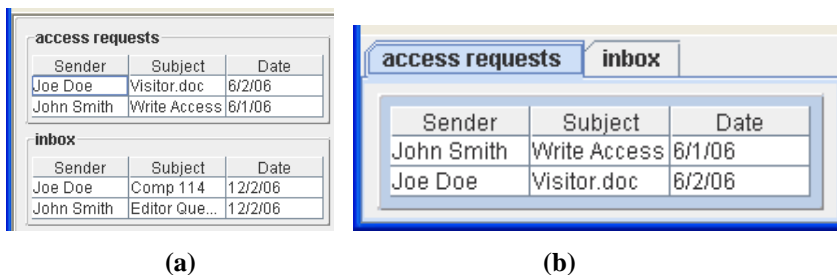

(a)   **(b)**

Figure 7 Nested Form and Tabbed Panes

Like tables and trees, tabs, forms and browsers are structured model-aware widgets in that they are composed of components that are bound to children of the model. However, in the former, the nature of the automatically generated child components is fixed by the designer of the widget, while this is not the case in the latter. For example, a browser pane can consist of a table, tree, form, textbox or any other component to which a model is bound. The algorithms we have given above are independent of the exact widget bound to a model child. Support for such heterogeneous model-aware widget-structures is a fundamentally new direction for toolkits, but is consistent with the notion of supporting model-aware widgets. Some existing structured-widgets such as `JTable` do allow programmer-defined widgets to be embedded in a widget-structure, but the embedded widgets are not themselves model-aware widgets automatically supported by the toolkit. For example, a `JTable` or `JTree` cannot be automatically embedded in a `JTable`.
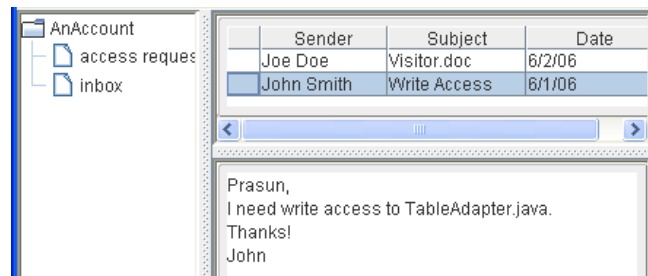

Figure 8 A Three-Level Browser

Thus, we have described an approach that allows a single model to be bound to both existing and new user-interface components. There are many ways of implementing it. From a practical point of view, it should be possible to layer it on top of an existing toolkit without requiring re-implementation of existing model-aware widgets. This, in turn, requires adapters between the universal table models and the existing toolkit models. We could require a separate adapter for each existing toolkit model. For example, we could define separate adapters for tree and table models. However, we take a more complicated and perhaps less modular approach in which we define a universal adapter that can support both existing and new widgets. This adapter understands the universal table interface, and implements the interfaces of the models of the Swing tree and table widget. This approach allows us to create a single adapter tree that can be dynamically bound to multiple widgets concurrently (Figure 9). The following algorithm describes the nature of the model structure, and how it is created:

```
UniversalAdapter createUniversalAdapter (Object model)
    if (model is UniveralTable)
        UniversalAdapter modelAdapter = new StructureAdapter(model);
        for each key, element of model
            UniversalAdapter keyAdapter = createUniversalAdapter(key)
            UniversalAdapter elementAdapter = createUniversalAdapter(element)
            keyAdapter.setParent(modelAdapter);
            elementAdapter.setParent(modelAdapter);
            modelAdapter.setKeyElement(keyAdapter, elementAdapter);
```

else return new LeafAdapter(model);

Unlike the model structure, the adapter structure includes back links from children to parents, which are required by the model of the Swing tree widget. These links also allow us to find the key associated with an element, which is needed to label the latter. Programmers can determine the universal adapter bound to a model, and retrieve information kept by it such as the parent adapter, children, and currently bound widget. Thus, they don't have to manually keep such book-keeping information.
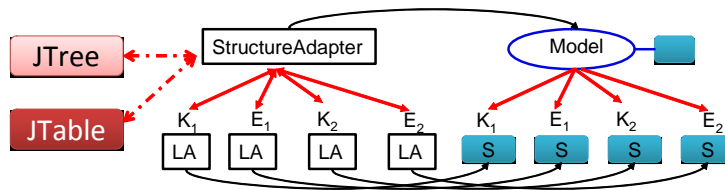

Figure 9 Implementation architecture (LA = LeafAdapter)

Figure 10 illustrates the use of universal adapters to simultaneously display a model using all structured-widgets supported by the toolkit. The model is an instance of `AnAccessRequest` with three fixed `String` keys, "File," "Rights," and "Message", which are associated with elements of type `String`, `String`, and `AMessage`, respectively.
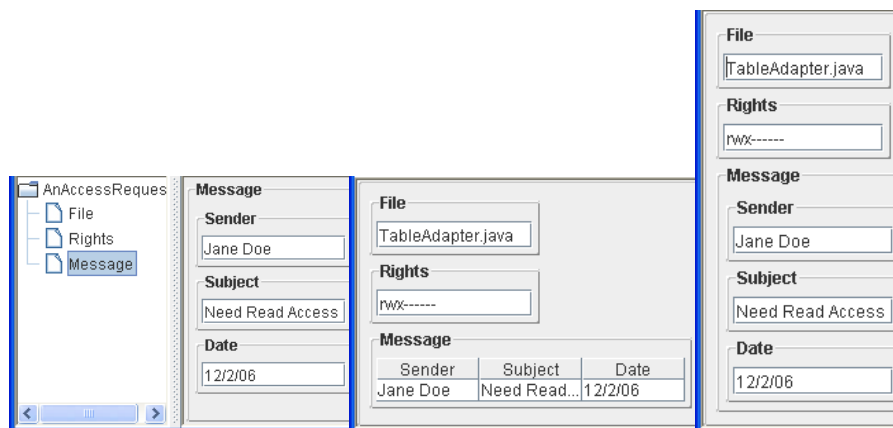

Figure 10 Simultaneously displaying a nested record using all structured widgets

6   Discussion

We have described above the interface of a model object, and techniques for automatically binding it to both existing and new model-aware structured widgets.

Thus, in comparison to existing user-interface toolkits, we simultaneously support a reduced model set and expanded model-aware widget set. Determining if we meet the other two requirements presented in Section 3 requires more analysis.

We went through (a) first a top-down phase in which we derived the interface of the universal table from well-established display-agnostic semantic structures, and (b) then a bottom-up phase in which we added additional methods to the interface needed by existing widgets. These methods do not increase the functionality of the model – their main purpose is to provide information the user-interface needs. For example, the user-interface needs to know the default key or element that should be added when the user executes the insert command. Similarly, it needs to know which keys and elements should be editable so that it can prevent the user from editing its visual representation.

Did the second phase compromise model purity? The answer, we argue, is no. The MVC architecture requires that the model be unaware of details of specific user-interfaces, so that these details can be changed without modifying the model. It is aware, however, that it will have one or more user interfaces – it allows views to be attached to it and sends notifications to them. The methods we have added play a similar role. The code in them also serves the same purpose as assertions. Assertions describe the behavior of an object to programmers, and prevent many mistakes. The additional methods we added in the bottom-up phase describe the behavior of an object to other objects – in particular the user-interface objects – and prevent mistakes. Consider the `isEditable()` methods. If a key or element is not editable, the model will not change it in the `put` method. However, an external object such as an editor would have to try to indirectly learn this behavior from repeated calls to the method. The `isEditable()` methods make this behavior explicit. Similarly, the methods returning the default key/element make the most specific class of the key/element apparent, and prevent additions of components of the wrong type. Just as notifications are now also used by non user-interface objects, we can expect these additional methods to have more general uses in the future.

Consider now programming effort. Mostly, our model does not require programmers to expose any information that is not also required by models of Swing. One exception is the information about editability of table data and components. While the Swing table model requires this information, the tree model does not. As this information not only increases the user-interface functionality but, in the long term, can be expected to prevent mistakes, we can say it does not significantly increase the programming cost. On the other hand, Swing requires tree nodes to keep track of their parent, and indicate if they are leaf nodes. If programmers are not careful, a forward (child) link can easily become inconsistent with a back (parent) link, leading to significant debugging effort. Such links are kept by our implementation but not the models. In addition, our approach uses keys as default labels of elements, which works in several user-interfaces such as the ones shown here. Thus, in some respects, our approach reduces the programming effort required to create models of even existing model-aware widgets. In summary, our approach meets the programming effort requirement.

This is not to say that our design has created the best user-interface tool today. There is limited abstraction flexibility in that all models of a widget must implement the same toolkit-defined interface. In addition, programmers must manually determine

the widget to be bound to a model, and set label and other user-interface attributes of these widgets. These are also limitations of existing toolkits. However, certain user-interface management systems (UIMS) such as [10-13] provide higher abstraction and automation. For these tools, our approach provides a method for increasing portability and reducing programming cost. We described above a simple approach for converting between the universal tables and existing models. If such code is added for each toolkit, then by layering on top of the universal table, a UIMS becomes portable and does not have to worry about implementing the new model-aware user-interface components supported by the universal table. We are planning to use this approach in a UIMS we are implementing as part of the ObjectEditor software[10]. For example, the properties of an object defined through getters and setters will be mapped to record fields, and then, using the scheme described above, to keys and elements of a universal table, which acts a proxy between the object and the widget. The interface of such an object would be programmer-defined and, hence, not constrained to a universal table. Thus, this approach assumes that a structured widget is linked in a chain to two models: a toolkit-defined proxy-model and a client-defined real-model. A UIMS can automatically translate between the events and operations of the two models, making the programmer oblivious of the toolkit-defined model. It is also possible to use this proxy-based approach in a manually-created user-interface – but the programmer would have to be responsible for translating between the two models. By reducing the number of toolkit-defined models, our approach reduces the number of translators that have to be written in the proxy-based approach.
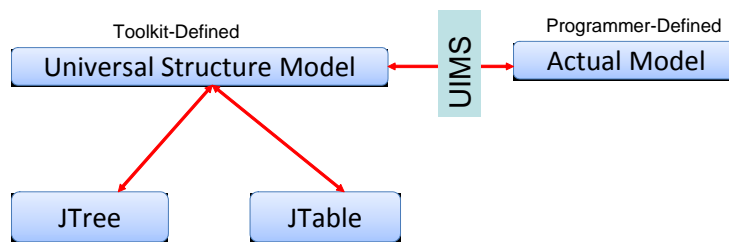

Figure 11 Interfacing with a UIMS to Support Programmer-Defined Types

To conclude, at the most abstract level, our message is that a toolkit should support both model and editor substitutability. At the next-level are the requirements of reduced model set, same or increased model-aware widget set, same or decreased programming effort, and model purity. The universal table interface and methods for mapping it to sequences, sets, records and nested tables and binding it to tables, trees, forms, tabbed panes, and browsers provide one approach to meeting these requirements. More work is required to extend and refine the requirements and approach, use and evaluate the approach, and incorporate it in higher-level tools.

References

1. Krasner, G.E. and S.T. Pope, *A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80.* Journal of Object-Oriented Programming, August/September 1988 **1**(3): p. 26-49.
2. Myers, B.A. *Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-Backs*. ACM *UIST'91*. Nov. 11-13, 1991.
3. Linton, M.A., J.M. Vlissides, and P.R. Calder. *Composing User Interfaces with InterViews*. in *IEEE Computer*. February 1989.
4. Dewan, P. *A Tour of the Suite User Interface Software*. in *Proceedings of the 3rd ACM UIST'90* October 1990.
5. Olsen, D.R., *User Interface Management Systems: Models and Algorithms*. 1992, San Mateo, CA: Morgan Kaufmann.
6. Codd, E., *A Relational Model for Large Shared Data Banks.* Comm. ACM, 1970 **13**(6).
7. Fraser, C.W. and D.R. Hanson, *A High-Level Programming and Command Language.* Sigplan Notices: Proc. of the Sigplan '83 Symp. on Prog. Lang. Issues in Software Systems, June 1983. **18**(6): p. 212-219.
8. Liskov, B., *Abstraction Mechanisms in CLU*. CACM, August 1977. **20(8)**: p. 564-576.
9. Rowe, L.A. and K.A. Shoens. *A Form Application Development System*. in *Proceedings of the ACM-SIGMOD International Conference on the Management of Data*. 1982.
10. Omojokun, O. and P. Dewan. *Automatic Generation of Device User Interfaces?* in *IEEE Conference on Pervasive Computing and Communication (PerCom)*. 2007.
11. Nichols, J., B.A. Myers, and B. Rothrock. *UNIFORM: Automatically Generating Consistent Remote Control User Interfaces*. in *Proceedings of CHI'2006*. 2006.
12. Paterno, F., C. Manicini, and S. Meniconi. *ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models*. in *INTERACT. 1997*. 1997.
13. Gajos, K. and D.S. Weld. *SUPPLE: Automatically Generating User Interfaces*. in *IUI*. 04.

Questions

Yves Vandriessche

Question: How do you finally handle the atomic objects?

Answer: We don't, there are a lot of ways to handle this and they keep being reinvented every day.

Remi Bastide

Question: Most modern dynamic languages, e.g. Javascript, use the dictionary as the basic data structure and programmers tend to have their API towards using dictionaries. This conflicts your arguments.

Answer: Most of these string-based languages actually come from SNOBOL.

Morten Harning:

Question: Would it not be obvious to handle interface to user defined Java classes by treating objects not implementing Universal Table interface as Universal Tables by interpreting setters and getters as keys in a Universal Table.

Answer: Absolutely. This is actually what we started doing, by only relying on Java naming conventions ended up being too messy.