Comparing Drag-and-Drop Implementations

Maxime Collomb, Mountaz Hascoët

Comparing *drag-and-drop* implementations

Maxime Collomb, Mountaz Hascoët

LIRMM, UMR 5506 du CNRS
161 rue Ada
34 392 Montpellier Cedex
Tel: (33) 4 67 41 86 56 - Fax: (33) 4 67 41 85 00
{*collomb, mountaz*}*@lirmm.fr*

Abstract

In this paper, we present the *drag-and-drop* as it is implemented in four systems: MacOS/Carbon, Windows/OLE, X-Window/Motif and Java/Swing. We use decomposition in five steps to analyze each implementation and compare them. These steps are: initialization, beginning of the interaction, *drag*, *drop*, and finalization. We also compare data structures used by each system.

Key words: drag-and-drop, interaction model

1 Introduction

In this paper, we will present the *drag-and-drop* as it is implemented in four systems: MacOS/Carbon, Windows/OLE, X-Window/Motif and Java/Swing. Then, we will discuss the common parts and differences of these implementations.

First, we will describe the establishment of the model used to describe each implementation of the *drag-and-drop*.

2 Establishment of the model

A *drag-and-drop* implementation contains two complementary aspects: (1) a complex event model ensuring *drag-and-drop* feedback and control by user and (2) a data transfer protocol and implementation. These two aspects are present in the four systems presented here.

Event models usually introduce three actors: the source (the component from where the object is moved), the system (toolkit and/or windowing system) and the target (the component on which the object is moved).

The *drag-and-drop* mechanism can be decomposed in five steps [6]:

- Initialization that is made once when creating the source and target components;
- Beginning of the interaction (detection of a pointer movement while a mouse button is maintained pressed). After which the source component ask for a *drag-and-drop* operation;
- *Drag* that consists of a pointer displacement, mouse button still pressed, while notifying potential targets so they can accept or refuse the object;
- *Drop* that takes place when the mouse button is released and consists of sending the data from the source to the target;
- Finalization that is done when freeing resources used by source and target components.

The transferred data can be of multiple formats. For example, when dragging a picture from a web page to a photo editing software, the picture should be copied and ready for editing but when dragging to an *html* editor, the picture url should be copied to the editor. This example illustrates the necessity of transferring the data under different formats. It can be noted that the same kind of multi-format transfer is used for *copy-paste* operations. Some editor even proposes a *special paste* option to select the format of data to paste.

In the next sections, we will detail this mechanism for four different systems.

3 MacOS/Carbon

The MacOS X system from Apple has two graphical toolkits: Carbon, inherited from previous versions of Mac OS, and Cocoa, inherited from NextStep. The description of *drag-and-drop* mechanism on Cocoa can be found in [1] and do not present originality compared to other systems presented here.

Thus, we will focus on Carbon toolkit [2] (figure 1) which is still widely used (e.g. iTunes). In carbon, *drag-and-drop* facilities are provided through the *DragManager*. *DragManager* is a module part of the *Human Interface Toolbox* implemented in Carbon.

3.1 Initialization

Before any interaction, the *drag-and-drop* system has to be initialized. The source component declares itself to the *DragManager* as a *drag-and-drop* source by calling *InstallWindowEventHandler()*. The target component declare itself as a potential target for *drag-and-drop* by calling *InstallTrackingHandler()* followed by *InstallReceiveHandler()*.

3.2 Drag detection

When a drag gesture is recognized by the system, it notifies the source component through the *dragHandler()* method. Then, the source component will ask for a *drag-and-drop* operation by calling *TrackDrag()* with a *DragReference* object containing the data to transfer in parameter. *TrackDrag()* method will terminate when the whole *drag-and-drop* operation will be completed.

3.3 Dragging

While the user is dragging the object, the source component is not notified of every events and the target component is notified of entering, moving and exiting events (*trackingHandler()*). Notifications contain a parameter of type *DragReference* (the data) so the target can know the type(s) of the data (thanks to the *getFlavorFlags()* method). If the target can manage at least one type of data provided by the *DragReference*, it can ask the system for a highlight of itself while the dragged object is within the component thanks to the methods *ShowDragHilite()* and *HideDragHilite()*.

3.4 Dropping

When the user drops the object, the target is notified by *receiveHandler()* and then gets the data by calling *getFlavorData()* on the *DragReference* object. The source component knows that *drag-and-drop* operation is finished thanks to the termination of the *TrackDrag()* method.

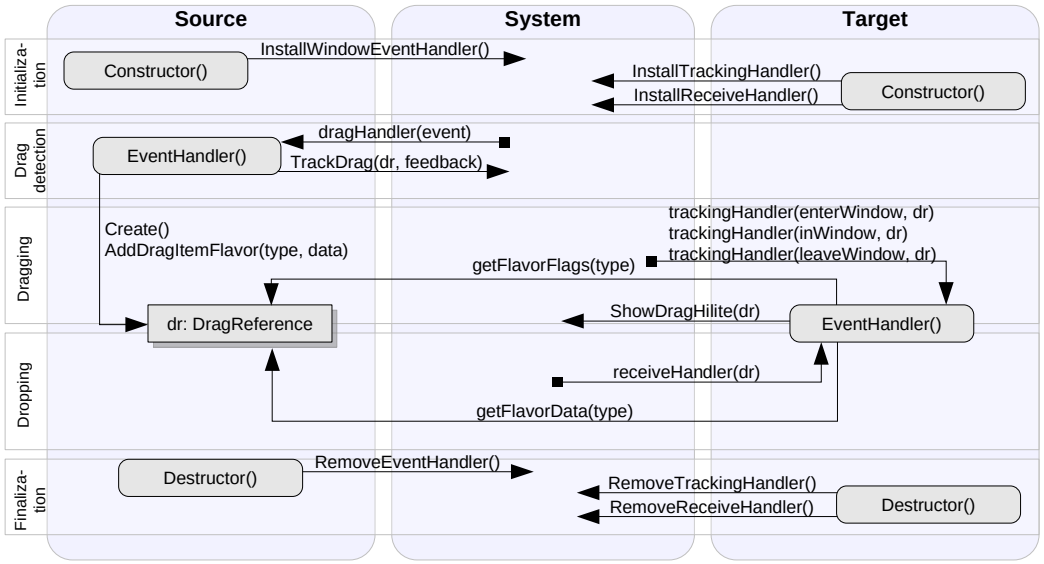

Fig. 1. *Drag-and-drop* mechanism under Carbon.

3.5 Finalization

In order to remove the *drag-and-drop* support, source component has to call *removeEventHandler()* and target component should call *RemoveTracking-Handler()* followed by *RemoveReceiveHandler()*.

3.6 Data Structures

A *DragReference* object contains the data under different formats. Each piece of data is associated to an identifying of type (for example, *TEXT* refers to the plain text format). The data can be given by the source component at the beginning of the operation or, if needed, only the data types are given (so the target can accept or reject the data) in addition of a callback routine that will be called by the *DragManager* when the data will be needed (only if the target accepts the data).

3.7 Conclusion

The MacOS/Carbon *drag-and-drop* mechanism features a *DragManager* that ensure a consistent behavior through the different applications that use *drag-and-drop*. It's the only implementation in this paper that features such a component.

4 X-Window/Motif

In the Motif toolkit (figure 2), the *drag-and-drop* model is based on the inter-application communication protocol provided by *XToolkit Intrinsic*. It is also based on the *Uniform Transfer Model* introduced in the version 2.0 of Motif [3].

4.1 Initialization

To become a source for a *drag-and-drop* operation, a component has to be of type *DragSource* thus recognize *ButtonPress()* events for the middle button. On the other side, to become a potential target, a component must call the *XmDropSiteRegister()* method.

4.2 Drag detection

The drag gesture detection is made by the source component when notified of *mouseEvent()*. Then, it ask the system for a *drag-and-drop* operation by calling *XmDragStart()* specifying a *DragContext* parameter containing the transferable data. *DragContext* will be notified of advancement of the operation.

4.3 Dragging

While object is dragged, the *DragContext* is notified via the callbacks *XmNdragMotionCallback()*, *XmNdropSiteLeaveCallback()* and *XmNlevelEnterCallback()*. The target component is notified via the optional *XmNdragProc()* method.

The data type match between what is dragged and what accepts the target is done by the system which is able to know the transferred data (method *XmNexportTargets()* of the *DragContext*) and the data types accepted by the target (methods *XmNimportTargets()* of the target)

At this step, there are two kinds of visual feedbacks: The target component can manage a *drag-under effect* by providing a *XmNdragProc()* callback. And the source component can manage a *drag-over effect* via a *DragIcon* object that is able to customize the cursor to reflect the action in progress (move, copy, etc.), type of the transferred data, state of the transfer, etc.

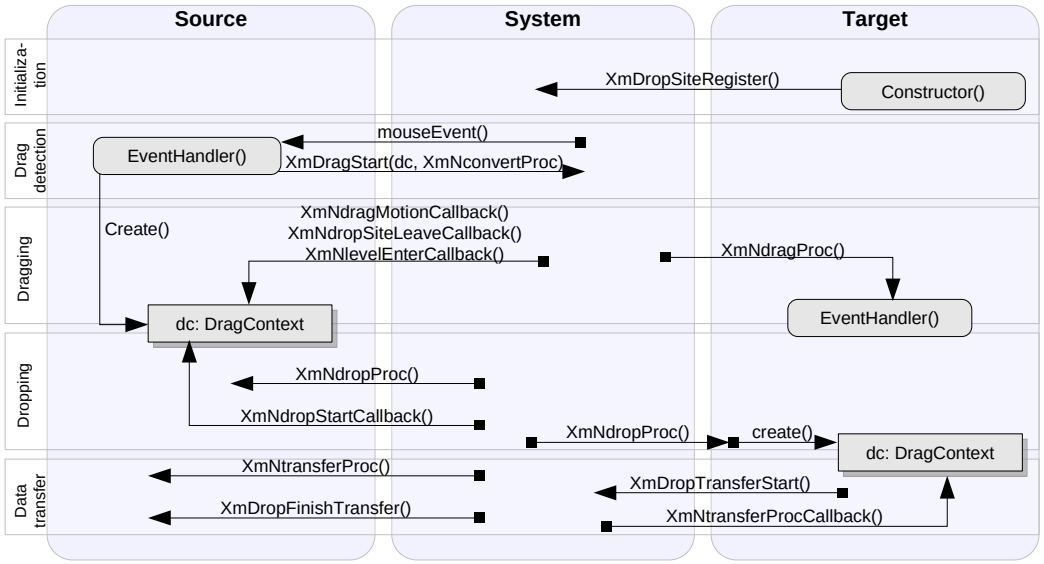

Fig. 2. *Drag-and-drop* mechanism under Motif.

4.4 Dropping

When the object is dropped, the *XmNdropProc()* callback of the target component is called. It executes the method *XmDropTransfertStart()* which creates an object of type *DropTransfert* that will manage the data transfer and determine the issue of the operation (success or failure).

During this step, visual effects cannot be customized. If the operation is a success, the dragged object and the target combine but if it is a failure, then the dragged object returns to its initial position to notify the user that data has not been transferred.

4.5 Finalization

Motif does not provide any finalization routines.

4.6 Data structures

Developer has different possibilities to implement the data transfer under Motif. Data can be transferred using the *X-Selection* mechanism (via the *X-Server*). This mechanism requires the data to be prepared at the beginning of the operation.

A second possibility is to use UTM (*Uniform Transfer Protocol*) that is de-

signed to unify the data transfers in Motif 2.0 whatever is the context (*drag-and-drop*, *copy-paste*, etc.). This protocol is based on a set of callbacks on both source and target of the transfer that allows them to determine the optimal format to use for the transfer and then to accomplish it.

The asset of this protocol is to be very general. But, on the downside, it introduces a complex mechanism of callbacks that implies an important work for the developer. This complexity is partially hidden in some widgets that do a part of the work, but implementing *drag-and-drop* mechanism for *new* widgets puts the developer in the front of this complexity.

4.7 Conclusion

Motif toolkit offers the most complex *drag-and-drop* mechanism. It is also original by some aspects: for example, the drag gesture detection is made by the source component by analyzing the mouse events. This leads developer to re-create the same *gesture-recognition-code* for each widget. Furthermore, behavior can be inconsistent between different widgets.

Another originality of the Motif *drag-and-drop* mechanism is to make the *system* determine which targets are compatible with the dragged data. This takes part in rendering this protocol complex with a great amount of callbacks.

5 Windows/OLE

The first *drag-and-drop* mechanism in Microsoft Windows was very simple and consisted in managing the message *WM_DROPFILE*. This mechanism was only allowing files to be dragged. It has been overridden with the advent of OLE (*Object Linking and Embedding*) that manages both *copy-paste* and *drag-and-drop* mechanisms. We will discuss here of the OLE *drag-and-drop* mechanism [7,5] (figure 3).

5.1 Initialization

Chronologically, the first action that will permit the user to drag and drop objects is the registration of the target component as a potential target for a *drag-and-drop* operation. To do so, interface *IDropTarget* has to be implemented and system routine *RegisterDragDrop()* has to be called. On the source component side, OLE support must be activated by executing *OleInitialize()* system routine.

5.2 Drag detection

The drag gesture has to be recognized by the source component (it is only notified of basic mouse events). Then, in order to initiate the operation, it calls the *DoDragDrop()* method specifying an *IDropSource* and an *IDataObject* as parameters. The execution of *DoDragDrop()* is synchronous: it ends when the whole *drag-and-drop* operation finishes.

5.3 Dragging

While the object is dragged, the source component is involved through the *IDropSource* interface via *GiveFeedBack()* and *QueryContinueDrag()* methods that provide the possibility to interrupt the operation at any time to the source component. The target component is notified of events through three methods: *DragEnter()*, *DragOver()* and *DragLeave()* when the cursor respectively enters, moves over and leaves the component. When cursor is entering, one parameter of the *DragEnter()* method is the *IDataObject* interface. So the *IDropTarget* can ask the *IDataObject* for the type of data that is dragged through the method *QueryGetData()*.

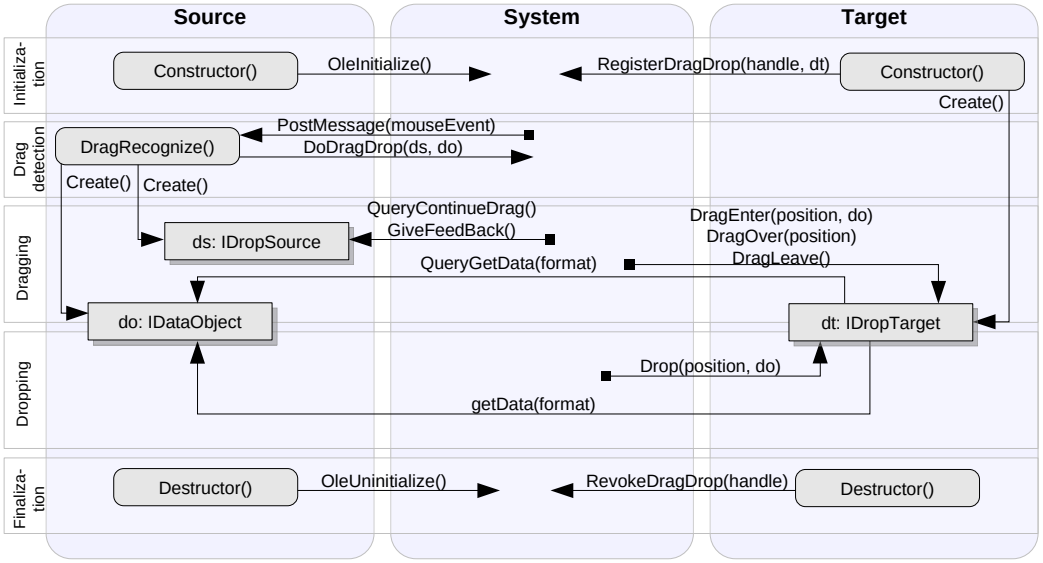

Fig. 3. *Drag-and-drop* mechanism under OLE.

5.4 Dropping

When the user releases the object, the target component is notified via the *Drop()* method with the *IDataObject* as parameter. Next, it can ask the

IDataObject for the content of the dragged object thanks to the *GetData()* method.

5.5 Finalization

If the source component does not need anymore to initiate *drag-and-drop* operations, it has to remove OLE support with the *OleUninitialize()* method. On the target component side, unregistering as a potential target can be done by calling *RemoveDragDrop()*.

5.6 Data structures

In OLE, data are encapsulated in an *IDataObject* interface, which can hold data into different formats. This object is created by the source component at the beginning of the operation and contains a set of methods that are called by the target to determine which kind of data is available or to get the data in a given format.

The data can be generated at the beginning or at the end of the operation depending on the implementation of this object since *IDataObject* is simply an interface.

5.7 Conclusion

The drawback of this protocol has already been observed in Motif: the source component is responsible of the drag gesture detection. This can lead to inconsistent behaviors.

6 Java/Swing

The model event of Java is based on the principle of delegation, event are managed by specialized objects [4]. The package *java.awt.dnd* contains a set of classes/interfaces that allows managing of *drag-and-drop* operations (figure 4). Although it is an AWT package, it is used in Swing applications too.

6.1 Initialization

When constructing the source component, an instance of a class implementing the *DragGestureListener* interface has to be created and associated to the component with the *CreateGestureRecognizer()* static method of the *DragSource* class. On the other side, the target component must register itself as a potential target by initializing his *dropTarget* property (with the setter *setDropTarget()*) and by creating an instance of a class that implements the *DropTargetListener* interface.

6.2 Drag detection

The system is responsible of detecting drag gestures. It notifies the source component via the *dragGestureRecognized()* method of the *DragGestureListener*. The source component then decides to begin a *drag-and-drop* operation by calling the *startDrag()* method of the *DragGestureEvent* received as parameter. *StartDrag()* claims two parameters: the data and a *DragSourceListener* that will be notified of the advancement of the *drag-and-drop* operation.

6.3 Dragging

While the dragged object is moving, both source and target component are notified by *dragEnter()*, *dragOver()*, *dragExit()* and *dropActionChanged()* methods but with different parameters: when the cursor enters a new target, it is notified of the transferred data so it can watch at the data type and accept or reject it. After the target has accepted or rejected the data (it is still dragged - no real transfer has occurred yet), the source component is notified by the *dragEnter()* method including the acceptance state of the target component. In a general way, source component is always notified after the target component.

6.4 Dropping

As the object is dropped, target is notified by the *drop()* method. It then accepts or rejects the drop, accesses the data and if data transfer worked, finally declares the operation as a success. Then, the source component is notified by the *dragDropEnd()* method.

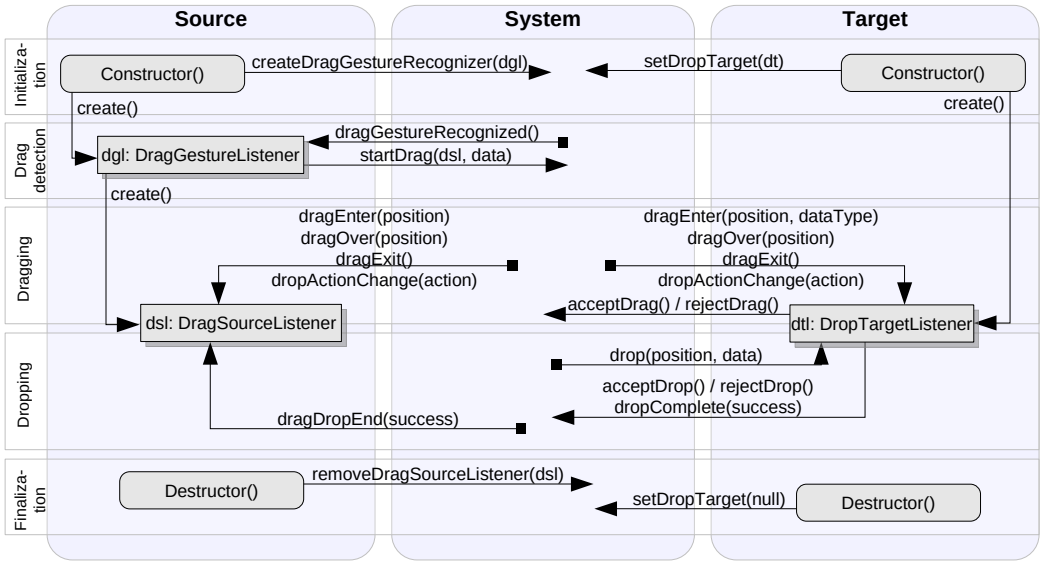

Fig. 4. *Drag-and-drop* mechanism under Swing.

6.5 Finalization

To stop managing *drag-and-drop* operations, source component calls the *removeDragSourceListener(listener)* of the *DragSource* class and the target component changes his *dropTarget* property with *setDropTarget(null)*.

6.6 Data structures

The data object must be an instance of a class implementing the *Transferable* interface. A minimal set of class implementing *Transferable* is provided by Java to support basic formats (plain text, file list, image). These predefined classes need the data to be given at the beginning of the operation.

6.7 Conclusion

The Swing mechanism (inherited from AWT) is multi-platform and therefore introduces some limitations. Its main drawback is to need the data (object of the transfer) to be generated at the beginning of the operation. It's regrettable, since *late-generated-data* are supported for data coming from non-java applications. That's why *acceptDrop()* has to be called before accessing data coming from external application.

Furthermore, Swing *drag-and-drop* mechanism is quite complicated from the

developer point of view: no less than eighteen classes and interfaces are involved in the *drag-and-drop* process.

7 Discussion

7.1 Differences: Dragging and dropping steps

Despite the similar general architecture illustrated by the five steps representation, the most notable differences are found in the core of the system during the *dragging* and *dropping* steps. Thus, during the *dragging* step, three kinds of events (or messages) are received: entering, moving over and leaving a target. These events are not generated in all systems: Swing notifies both source and target components of these three events while OLE and Carbon only notify the target and Motif only notifies the source component. This leads to a *drag-under/drag-over* behavior that, even if reproducible on other systems, is specific to Motif.

Another difference that appears between systems is the management of actions and cancellation. Action can be *copy*, *move*, etc. and is selected through *ctrl*, *shift* and *alt* modifier keys. But, while Carbon manages the entire action process, Swing selects the action in function of modifiers and notifies the source and target components. On the other side, Motif and OLE let the source component choose the action to take in function of the modifiers. That leads to a more flexible system but also to inconsistent behaviors of *drag-and-drop* actions across applications. About the cancellation of the operation (generally by pressing the *escape* key), the same difference appears: Swing and Carbon manage it and Motif and OLE let the source component manage it.

During the *dropping* step, on all exposed systems, the target component is notified first, accepts or refuses the *drop*, and after the source component is notified of the end of the *drag-and-drop*. However, the data transfer does not present a similar scheme on all systems: it is integrated in the *drop* notification on Carbon, OLE and Swing and is a specific step on Motif. The data transfer can be done according to two approaches: *early-generated* or *late-generated*. *Early-generated* data is known at the beginning of the operation while *late-generated* data is build at the end of the operation according to the type of data that requires the target. All systems enable these two approaches though it is more complex to use *late-generated* data on Swing since it requires writing specific classes for data management.

7.2 Shared points: initialization, drag detection and finalization

Two elements are found in all systems and are almost identical: the initialization and the finalization. In all cases, potential sources have to be prepared for *drag-and-drop* notifications and potential targets have to register themselves to the system. And, except Motif, all systems provide an unregistering procedure for targets that get destroyed or become inactive.

Another step that is abstractly identical on the different systems is the *drag detection* step. Depending on the system, the analysis of the mouse events is more or less done by the system: for example, on OLE, the source component has to analyze basic mouse events to detect a drag gesture while the work is already done on Swing and the source component is directly notified by a *DragGestureRecognized()* method. This difference is not of major order but beg the question of consistency. However, at this point, it's always to the source component to ask for beginning the *drag-and-drop* operation.

It is noticed that *drag-and-drop* is ruled by an abstractly similar protocol from one system to another. These similarities are due to the constraints imposed by the interaction technique itself.

8 Conclusion

Based on the description of the four *drag-and-drop* systems described in this paper, we can see that all the implementations can be described using five functional steps. However, these implementations present deep dissimilarities and any attempt of homogenization is very tedious. As attests the example of Java/Swing, a multi-platform system introduces limitations to keep the compatibility with each supported system.

9 Acknowledgments

We would like to thank Renaud Blanch for his precious help on existing implementations analysis, and for his skills on MacOS/Carbon.

References

[1] Apple developer connection. Drag and drop. *http://developer.apple.com/documentation/cocoa/conceptual/draganddrop/*.

[2] Apple developer connection. Drag manager programmers guide. *http://developer.apple.com/documentation/carbon/conceptual/dragmgrprogrammersguide/dragmgrprogrammersguide*.

[3] Fountain Anthony, Huxtable Jeremy, Ferguson Paula, and Heller Dan. Motif Programming Manual, 2nd edition, volume 6A. *OReilly Associates*, 2002.

[4] Gene De Lisa. How to drag and drop with java 2. *http://www.javaworld.com/javaworld/jw-03-1999/jw-03-dragndrop.html*.

[5] James Brown. Ole drag and drop from scratch. *http://www.catch22.net/tuts/dragdrop.asp*.

[6] M. Hascoët, M. Collomb, and R. Blanch. Evolution du drag-and-drop : du modèle dinteraction classique aux surfaces multi-supports. *revue I3, 4(2)*, 2004.

[7] Microsoft. The microsoft developer network. *http://msdn.microsoft.com/*.