GUI Migration across Heterogeneous Java Profiles

Candy Wong, Hao-hua Chu, Masaji Katagiri
DoCoMo Communications Laboratories USA, Inc.
Seamless Experience Environment Laboratory
181 Metro Drive, Suit 300
San Jose, CA 95110 USA
+1 408 573 1050
{wong, haochu, katagiri}@docomolabs-usa.com

ABSTRACT

Existing cross-platform graphical user interface (GUI) development tools do not support migrate-able GUIs as they do not consider any runtime concern, such as running state transformations. To address this problem, we introduce Scalable Graphical User Interface (SGUI). It allows GUI developers to construct a platform-independent GUI that can be migrated across heterogeneous Java profiles. In this paper, we will focus on two major problems in supporting migrate-able GUIs. First is layout and widget transformation, which describes how to layout a presentation after a GUI is migrated from one platform to another. Second is running state and event handling transformations, which describes how to transform running states and event handlings when a presentation is changed after a migration.

KEYWORDS

User interface development tool, multi-platform user interface, GUI migration, heterogeneous devices, Java

INTRODUCTION

Nowadays, mobile devices are so popular that many end-users own multiple mobile devices. To better assist an end-user's mobility, he/she would use a device which best suits his/her situation. For example, when an end-user is at home, he/she plays a PC game with his/her Notebook PC. However, in the middle of the game, if he/she has to go out to meet someone, he/she could continue the game with his/her Pocket PC while he/she is on a bus.

Based on this scenario, we found that there is a need to provide an application continuity for end-users. One of the solutions is to allow a runtime application migration. The migration involves the application logic migration and the user interface migration. This paper mainly discusses the user interface migration. Details of the application logic migration can be found in [1].

The goal of our work, called SGUI, is to provide a framework for developers to build a platform-independent GUI that can be migrated across heterogeneous Java profiles. The considered profiles include Java 2 Standard Edition (J2SE) [2] which is targeted for thick clients (e.g., Notebook PCs), Java 2 Micro Edition (J2ME) Personal Profile [3] which is targeted for thin clients (e.g., Pocket PCs), and J2ME DoJa Profile [4] which is targeted for very thin clients (e.g., DoCoMo 503i cell phones).

Existing cross-platform UI development tools, including XIML [5], UIML [6], XWeb [7], and Unified User Interface [8], mainly consider design-time problems (e.g., the screen size problem). These approaches can be categorized into two types: the autonomous approach and the automatic approach. The autonomous approach generates presentations autonomously by requiring every single layout details from developers. For example, UIML and Unified User Interface require developers to specify layout on each platform. However, the specifications require developers to have thorough knowledge of each platform's constraints. The automatic approach generates presentations automatically by leaving almost no layout controls to developers. For example, XIML and XWeb generate presentations automatically without many involvements from developers. Yet, developers are not allowed to specify their expectations.

In this paper, we propose a new layout algorithm which constitutes a middle ground between the two approaches. The algorithm makes use of a layout specification (e.g., a widget's x-y coordinates) of a single platform and a set of transformation rules. A platform is referred to a device that supports at least one Java profile. A transformation rule is referred to a rule that is composed of a widget transformation, a running state transformation, and an event handling transformation. SGUI provides generic transformation rules and developers can modify the rules into application-specific rules.

We also propose running state and event handling transformations which are not considered by existing cross-platform development tools. Running state and event handling transformations are critical elements for GUI migrations in providing consistent experience for end-users. For example, after a migration, end-users should be able to review the state changes that were made before a migration, and they should be able to trigger the application features that were available before a migration.

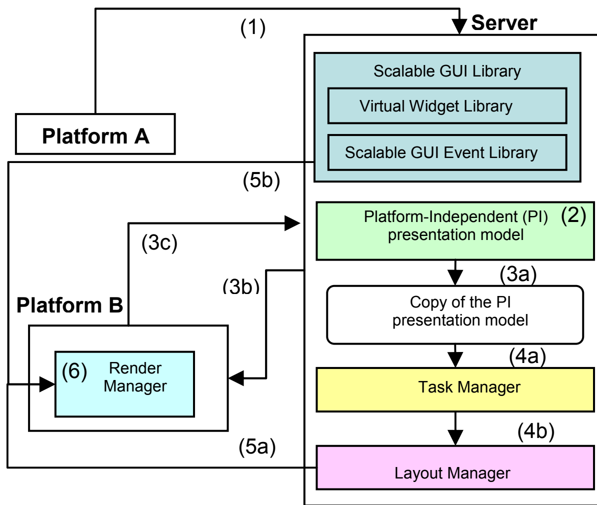

Figure 1: Overall Architecture

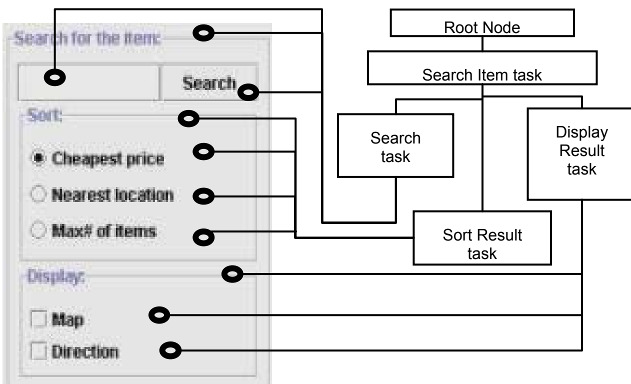

Figure 2: PI Presentation Model and the expected J2SE SWING GUI (that is generated from the PI Presentation Model) of a Searching Shopping Item application.

The rest of this paper is organized as follows. The next section shows the overall architecture of a GUI migration. The following sections describe the layout and the widget transformation, the running state and the event handling transformations, and the conclusion. Related work is described in the appropriate sections.

This paper focuses on the graphical UI, as it is the most widely accepted modality. Perceptual/cognitive UIs, such as changing presentations/modalities at different end-users' contexts, are out of scope of this paper.

OVERALL ARCHITECTURE

Figure 1 shows the overall architecture of a GUI migration. A GUI migration involves two major parties: a server and a client platform. The server generates a final presentation and the client platform renders it.

Before GUI migration

The server is composed of the following components: a SGUI library, a platform-independent (PI) presentation model, a task manager, and a layout manager. They are stored in the server before any GUI migration.

Since different Java Profiles support different GUI libraries and events, the scalable GUI library employs a mapping technique to abstract out their differences. The scalable GUI library contains two sub-modules: the *virtual widget library* and the *scalable GUI event library*. The virtual widget library contains a set of GUI widgets similar to Java SWING library's widgets, which is the richest set of Java library widgets among all Java profiles. A virtual widget library is implemented for each Java profile. For example, on a J2ME DoJa profile, a virtual widget library is implemented to describe mappings between virtual widgets and DoJa library widgets.

Similarly, the scalable GUI event library provides mappings between scalable GUI events and Java profile-specific GUI events. Even though an event can only be generated by a particular input method (e.g., a soft key event can only be generated by a soft key) and different platforms support different input methods (e.g., a PC supports a mouse, but a cell phone supports a keypad), Scalable GUI events allow an application to handle events in any considered Java profile. For example, a scalable GUI *action* event associated with a button press can be generated from a mouse click on a PC, a tap from stylus on a Pocket PC, or a select-key press on a cell phone.

Using the scalable GUI library, developers can prepare a PI presentation model at the design-time. Figure 2 depicts a PI presentation model for a search item application. The model has a tree-like structure. The root node, which represents an entire application, occupies the top of the tree. Child nodes of the root node are task nodes representing different end-users' tasks. Each task node can be further divided into sub-task nodes, sub-sub-task nodes, and so forth, until the leaf nodes are represented by virtual widgets. For example, the Search task node shown in Figure 2 is divided into 3 sub-task nodes, and they are represented by a "Search for the item:" virtual label, a "Search" virtual button, and a virtual textfield. Similar ways of groupings are also mentioned in [5].

Unlike the approach in [5], our model allows developers to provide hints on each task node. These hints are important and compulsory for automatically generating high quality presentations. The hints specify: (1) a detail layout for each node based on the Java Grid Bag Layout Constraint, which is the most flexible layout constraint among all Java profiles, (2) a *task preference*, which is implemented as an array of Booleans in which each array index represents a platform, describes whether a task is suitable for a particular platform or not, (3) a *priority*, which is implemented as an integer, denotes the desired layout sequence of each widget, (4) a *split-ability*, which is implemented as a Boolean, indicates whether the widgets can be spread over multiple pages or not, and (5) an *importance*, which is implemented as a Boolean,

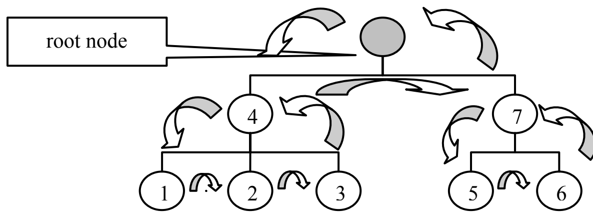

Figure 3: Layout algorithm, where node *v* has children virtual widgets that are not shown in this figure. The number of each node indicates the node's layout priority.

shows whether a widget is core or optional. Core widgets are defined as the most frequently used widgets [12], or widgets in performing a major task. The rest are defined as optional widgets. For example, the Search Item task in Figure 2 has 3 sub-tasks. In order to perform a Search Item task, an end-user must enter the item's name and press a button to initiate the searching process. However, an end-user does not require specifying the sorting preferences and the display options, as they are just enhancements for displaying the search result. Thus, widgets associated with the Search task are core and the remaining widgets are optional.

The hints are used by the task manager and the layout manager; the task manager uses the task preference and the layout manger uses the rest. The task preference is described in the GUI migration section. The layout specification, the priority, and the split-ability are described in the layout algorithm section. The importance is described in the widget transformation section.

GUI Migration

Figure 1 also shows the process of a GUI migration from platform A to platform B. In step 1, platform A serializes states of all widgets on its platform-specific (PS) presentation model (the model that is customized for platform A from the PI presentation model, by the task manager and the layout manager), and sends a migration request along with its PS presentation model to a server. In step 2, the server updates the state of the server's PI presentation model with the state of platform A's PS presentation model. The update is done through the running state transformation that is described later. In step 3, the server creates a new copy of its PI presentation model and probes platform B for its capabilities. In step 4, after getting platform B's capabilities, the task manager trims off unnecessary task nodes from the copy to form a PS presentation model. From it, the layout manager generates a presentation. If transformations rules are involved in the presentation generation, the layout manager will further customize the PS presentation model so that it will include the transformed widgets. In step 5, the server then sends the presentation, the PS presentation model, and the required portion of the SGUI library to platform B. In step 6, platform B's render manager displays the presentation.

LAYOUT AND WIDGET TRANSFORMATION

At runtime, the layout algorithm and the widget transformation are used to generate PS presentations for various screen sizes. There are three major requirements: (1) the generated presentations should have reasonably high qualities with the minimum help from developers, (2) the layout algorithm has to be simple in order to minimize any presentation generation delay, and (3) as some platforms do not support scrolling and scrolling normally degrades the GUI usability, scrolling should not be used to display an entire presentation.

In the past, there are many proposed methods in describing how to layout widgets for various screen sizes. However, some techniques, such as the one proposed in [9], involve high computation complexities. Some approaches require too much information from developers. For example, Humanoid [10] asks developers many layout-related questions before generating a final presentation. Among all proposed methods, T_EX [11] is the most promising method in formatting 2-dimensional box-like GUI widgets [7, 9]. T_EX allows each widget to report its desired size for the positioning.

Fortunately, Java has default layout managers that are similar to T_EX. Java also allows developers to specify the desired location of each widget through a set of predefined layout constraints. However, each set of layout constraints can only generate a single presentation. In order to meet our first requirement, we propose to only require one set of layout specification for a single platform, which can generate multiplatform presentations. The specification is the same as the Grid Bag Layout Constraint. Developers can specify the layout according to the presentation that consumes the largest dimension such as a presentation for a PC. Our layout algorithm will try to follow the specification as closely as possible.

When the specification of some widgets violates the screen size, we use Flow Layout to position those widgets. We choose Flow Layout because it is simple and it involves minimum computation while maintaining a reasonable presentation [12]. Since Flow Layout itself may not provide a high quality layout, we only apply it on violating widgets, and we keep on applying Grid Bag Layout on non-violating widgets. If the presentation is still larger than the screen size, we will apply transformation rules for meeting the third requirement.

Layout Algorithm

Figure 3 shows a more detail layout algorithm. It starts from the root node (the first current node).

1. If the current node has unprocessed direct child nodes
    - Find the unprocessed direct child node that has the highest priority, and set that node as the current node. Repeat step 1.

    Else
    - Proceed to step 2.

Type of Widget Transformation	Original Widget(s)	Transformed Widget(s)
One2One	List	Drop-down box
One2Multiple	Table	Lists, Drop-down boxes
MultipleSameClass2One	Radio Button	List, Drop-down box
MultipleSameClass2One	Text Fields	Text Field
MultipleSameClass2One	Labels	Drop-down box
Multiple2Multiple	A set of Label and Text Field pairs	A Drop-down box and a Text Field

Table 1: Sample transformation rules

Figure 4: Sample Muliple2Muliple transformation

2. Process the current node and mark it as *processed*. Proceed to step 3.
3. If the current node is a root node
    - Terminate the algorithm.

   Else
    - Set the current node's parent node as the new current node.
    - Repeat step 1.

The procedures of processing a node are as follows. Denote the optimum size of a page as the size of a screen.

1. Virtual widgets associated with the node are placed on a page according to Grid Bag Layout Constraints. The precise size of the page is then calculated.
2. If the page is bigger than the screen
    - We apply Flow Layout to all direct virtual widgets of the node.
    - If the page is still too big
        - If the node is split-able
            - We open a new page to place the extra widgets if the node is split-able.

          Else
            - Transformation rules will be applied on some/all widgets under the node. Place all direct widgets under the node with Flow Layout.
            - If transformation rules fail
                - The algorithm will leave the page as it is (i.e., some widgets will not be shown). Developers can find out the cause of the problem from the final presentation.

              Else
                - Exit the procedures.

      Else
        - Proceed to step 3.

   Else
    - Proceed to step 3.
3. The page is stored as a virtual widget associated with the node. Exit the procedures.

Widget Transformation

Widget Transformation is the transformation from one (composite) widget to another (composite) widget that consumes less space. A composite widget is a widget that is composed of several widgets. The goal of the widget transformation is to avoid applying any scrolling feature, by reducing the size of the presentation. To achieve this goal, we have to address two major issues: (1) which widget(s) we should transform, and (2) how we should transform the widget(s).

In order to solve the first problem, we retrieve the widget that triggers transformation rules. If the widget is a singular widget, we will apply a transformation rule on that widget only. If it is a composite widget, we will divide all widgets inside the composite widget into 2 groups (based on the *importance* hint): (1) the core widgets and (2) the optional widgets. We apply transformation rules on optional widgets first, as transforming a widget into a more compact widget degrades the GUI usability [12]. If there is no optional widget or the size reduction is insufficient, transformation rules will be applied on core widgets.

To solve the second problem, our model tries to find the best suitable transformation rule from a set of rules in a particular manner. Here, the suitable rule means the rule that provides the required size reduction. The transformation rules are categorized into 4 types: (1) *One-to-One* defines transformations of a single widget to another single widget, (2) *One-to-Multiple* defines transformations of a single widget to multiple widgets, (3) *MultipleSameClass-to-One* defines transformations of multiple widgets that belong to the same class (e.g., a set of radio buttons) to a single widget, and (4) *Multiple-to-Multiple* defines transformations of multiple widgets that belong to different classes to another set of multiple widgets. Each Multiple-to-Multiple rule is composed of a set of MultipleSameClass-to-One rules and a set of classic relationships introduced by [12] (e.g., in Figure 4, through the MultipleSameClass-to-One rules, the set of labels is transformed into a drop-down box and the textfields are transformed into one textfield. These two transformations are linked together with a Form-Filling relationship. Form-Filling describes the relationship of widgets that are tied together for filling in end-users' personal info). Other sample rules are shown in Table 1.

To select a rule, our model firstly prioritizes the rules according to their types. Since Multiple-to-Multiple rules can change the overall presentation drastically and can

```
private class FormFillingTransformation {

 /* widget transformation */
  1.    construct a new virtual drop-down box (DD1)
  2.    copy the virtual "Item Name::", "Brand Name:", …etc labels' properties
        (e.g., font size) to DD1
  3.    construct a virtual textfield (TF1)
  4.    copy the virtual "Item Name:", "Brand Name", …etc. textfields'
        properties to TF1

  /* running state transformation */
  1.    Record the original virtual label-textfield pairs into a Java hash table
        (shown in Figure 6)
  2.    Add a scalable Selection event listener (from our event library) to DD1,
        and specify the action. So that when an end-user selects an item from
        DD1, e.g., "Item Name:" the corresponding value, e.g., "Bottle Water",
        can be retrieved from the hash table.
  3.    Display the value onto TF1

  /* event handling transformation */
  //originally, developers specify that the value of a "Item Name", or "Brand
  //Name", or …etc. textfields will be automatically updated when an end-user
  //hit a ENTER or equivalent key

  //After a migration, when a ENTER key event is received,
  1.    Get the value on TF1, say "Coke"
  2.    Get the currently selected item on DD1, say "Item Name"
  3.    Find the title of the selected item in the hash table and update the
        corresponding value.
  4.    Create an event with "Item Name" as the source of the event and
        "Coke" as the new value.
  5.    Send the event to the original virtual "Item Name" textfield to
        synchronize the states of the original and transformed textfields.
}
```

Figure 5: Pseudo-codes of Figure 4's transformation.

Item Name	Bottle Water
Brand Name	Alhambra
Quality	4
Expected Price/Unit	$0.75

Figure 6: Hash table that is used in Figure 5.

highly degrade the GUI usability, the Multiple-To-Multiple rules have the lowest priorities. For other rules, our model filters them first before doing any prioritization. That is, our model ignores rules whose original widgets require input methods that are not supported by the transformed widgets. For example, a developer specifies a J2ME DoJa button to be interacted with a mouse-in event (i.e., when a mouse-arrow points, not clicks, to a button, an action will be invoked). The button cannot be transformed to a softkey, as a softkey does not support a mouse-in event. For the remaining rules, our model prioritizes them according to a set of space reduction parameters. Our model sets the rule that provides the less space reduction to have the highest priority, as a less compact widget usually has a higher GUI usability than a more compact widget [12].

The space reduction parameters include: width reduction ratio, height reduction ratio, and dimension reduction ratio. The importance of these parameters is dynamically changing according to the condition that triggers transformation rules. That is, when a transformation rule is triggered by a widget that is too wide (or tall), the width (or height) reduction ratio is the most important parameter; the height (or width) reduction ratio is the second, and the dimension reduction ratio is ignored as its result is covered by the width and height reduction ratios already. When the rule is triggered by a widget that is both too width and too tall, the dimension reduction ratio is the most important one, and the rest are ignored because of the similar reason.

Since we are dealing with Java-platforms, we choose to specify the rules in Java classes. Each rule is represented by one Java class. Inside a class, running state and event handling transformations are also specified. Sample pseudo-code is shown in Figure 5.

RUNNING STATE AND EVENT HANDLING TRANSFORMATIONS

Other critical ingredients for GUI migration are running state and event handling transformations.

Running State Transformation

It is very likely that the migrated presentation employs a different set of widgets from the original presentation. There is a need to map running states between the original presentation's widgets and the migrated presentation's widgets. This requirement leads to a question of "how to interchange running states among various widgets?"

To answer this question, we can make use of the virtual widget library, which is mentioned in the overall architecture section, to map the running state of a PS widget to the running state of its generic virtual widget. To realize this mapping, prior to a migration, the running state of a PS widget and that of its corresponding virtual widget are synchronized. After the migration, the running state of the virtual widget can be retrieved and presented on the migrated PS widget. We can employ similar processes when transformation rules are applied, by mapping running states of the original virtual widgets and that of the migrated virtual widgets. As the state mapping is unique to each transformation rule, each rule has to provide a method of the running state transformation. Figure 5 shows pseudo-codes of Figure 4's running state transformations. In the code, the states of all original textfields are stored in a Java hash table as shown in Figure 6. The transformed textfield displays these states one at a time, based on the state of the drop-down box.

Event Handling Transformation

Event handling transformation is the most important part for providing the same level of GUI interaction after GUI migrations. For example, assuming Figure 4 shows a GUI migration from a J2SE SWING platform to a J2ME PersonalJava AWT platform. Before the migration, entering an item name on a SWING textfield generates a SWING event, which triggers an action of storing the item name into a database. After the migration, when the

SWING textfield transforms to an AWT textfield, the same action should be able to be triggered even the AWT textfield only supports AWT event.

To meet this requirement, we use the scalable GUI event library that is described in the overall architecture section. Using the previous example, our model abstract both the SWING event and the AWT event to a generic virtual event; the associated action is triggered when the virtual event is received. We can employ similar abstractions when transformation rules are applied. As the event mapping is unique to each transformation rule, each rule has to provide a method of the event handling transformation. Pseudo-codes of Figure 4's event handling transformations are shown in Figure 5. In the code, when an event is generated from the transformed textfield, our system pretends to be the corresponding original textfield and sends out an event for triggering the action.

DISCUSSION

There are 3 important strengths in this project. Firstly, we introduced the GUI migration concept, which is not explored by existing cross-platform GUI approaches. Secondly, we introduced the use of transformations in fitting a page, which is bigger than the screen size, onto the screen. Existing approaches mainly employ scrolling to solve this screen size problem. Thirdly, we provided flexibility for developers to build application-specific transformation rules. We found that this customization is required, as there are many ways for transforming one widget to another but only developers know which way is the best for their applications.

However, the current customization process is tedious and complicated. Developers are required to build each transformation rule by constructing a Java class. Inside the class, developers have to write code in specifying how to transform a widget, its running state, and its event handling. Another weakness is about the presentation customization. Currently, developers can customize the presentation by customizing the PI presentation model, but this customization is indirect and hard to manipulate.

CONCLUSION & FUTURE WORK

In this paper, we described the techniques for GUI migrations. To enable GUI migrations, we introduced our layout and widget transformation for generating various presentations. We also proposed our running state and event handling transformations, which can preserve the running states and end-users' interaction after migrations.

We would like to improve the two discussed weaknesses in the future. For the transformation rule customization, we can provide a graphical tool that can automatically generate transformation rules once developers provide the type of transformation rules, the original version of a scalable widget, and the transformed version of a scalable widget. For the presentation customization, we can provide a drag-and-drop interface for developers to manipulate widgets on the final presentation. To realize the interface, we need to record the position and the size of each widget on the final presentation.

REFERENCES

1. Chu, H., Song, H., Wong, C., and Kurakake, S., "Seamless Applications over Roam System", *UbiTools'01* (Part of *ACM UbiComp'01*), September 2001, *http://choices.cs.uiuc.edu/UbiTools01/.*
2. Sun Microsystems, "Java™ 2 Platform, Standard Edition White Paper", June 2000.
3. Sun Microsystems, "PersonalJava™ Technology – White Paper", August 1998.
4. NTT DoCoMo, Inc., "i-mode Java Content Developer's Guide", May 2001.
5. Eisentein, J., Vanderdonckt, J., and Puerta, A., "Applying Model-Based Techniques to the Development of UIs for Mobile Computers", *Pro. of ACM IUI'01*, January 2001, pp. 69-76.
6. Harmonia, Inc., "User Interface Markup Language (UIML) Draft Specification", January 2000.
7. Olsen, D., Jefferies, S., Nielsen, T., Moyes, W., Fredrickson, P., "Cross-modal Interaction using XWeb", *Proc. of ACM UIST'00, November 2000.*
8. Stephanidis, C., Savidis, A., and Akoumianakis, D., Tutorial on "Universally accessible UIs: The unified user interface development". *Tutorial in ACM CHI'2001, 31 March - 5 April 2001.*
9. Masui, T., "Evolutionary learning of graph layout constraints from examples", *Proc. of ACM UIST'94*, November 1994, pp.103 – 108.
10. Szekely, P., Luo, P., and Neches, R., "Facilitating the Exploration of Interface Design Alternatives: The HUMANOID Model of Interface Design", *Proceedings ACM CHI'92*, May, 1992, pp. 507-514.
11. Linton, A., Vlissides, M., and Calder, R., "Composing User Interfaces with InterViews", IEEE Computer, 22(2), February 1989.
12. J. Vanderdonckt, "Knowledge-Based Systems for Automated User Interface Generation: The TRIDENT Experience", *Technical Report RP-95-010, Fac. Univ. de N-D de la Paix, Inst. D'Informatique*, Namur, 1995.
13. D. Thevenin, and Joelle Coutaz, "Plasticity of User Interfaces: Framework and Research Agenda", *Proc. of IFIP Interact'99*, 30 August – 3 September 1999.
14. F. Bodart, J. Vanderdonckt, "On the Problem of Selecting Interaction Objects", *Proc. of HCI'94*, August 1994, pp. 163-178.