The Wireable User Interface

-

A graphical tool integration mechanism

Markus Kiser

Department of Computer Science
University of Zurich
Winterthurerstr. 190
CH-8057 Zürich
kiser@ifi.unizh.ch

August 94

1. Introduction

In this paper, we introduce the tool integration mechanism, called the *Wireable User Interface (WUI)*. In contrast to existing tool integration mechanisms, the WUI is primarily settled in the user interface component of tool environments, and therefore allows the creation of tool environments with a very powerful and consistent user interface.

The key idea behind tool environments is that of the whole being more valuable and efficient than the sum of its pieces. Tools are seen as basic building blocks that can be combined in new ways as needed by the circumstances. However, in order to make tools work together efficiently they must be integrated.

Ideally, in a tool environment, one wants to have a central database that can be accessed by all the tools of the environment, and that offers a consistent way for all the tools to store and retrieve their data. Moreover, tools should also have the possibility to access each other's data, if needed, and tools should be able to notify each other of changes made to the shared data, in order for all the tools to keep a consistent view of the data. In a tool environment, it is often also useful that tools have the ability to exchange their information automatically, without the intervention of the user. Finally, one also wants the user interfaces of the tools (and the user interface of the environment as such) to have a consistent look-and-feel. In order to separate all these different aspects, one usually identifies three dimensions that have to be integrated: Data Integration, Control Integration, and Presentation Integration [SCHE93].

There are a number of tool integration mechanisms known today. A good overview can be found in [MEYE91]. However, these tool integration mechanisms are mostly concerned with the data dimension, and/or with the control dimension of tool integration (the latter, in particular, is the case in the FIELD approach [REIS90] which suggests a message passing system for control among tools). They are not explicitly concerned with the aspects of integrating the user interfaces. Probably the most wide-spread approach that explicitly aims at aspects of the user interfaces is Tk, based on Tcl [OUST91]. However, Tcl and Tk are very low-level, in the sense that they don’t say anything about the architecture of tool environments, but only provide a means of letting tools and their user interfaces communicate.

There exists a large number of tool environments (or tool-environment-like applications). Some examples are TakeFive’s SNIFF, Harlequin’s LISPWorks, Symantec’s THINKPascal and THINKC, Frame Technology’s FrameMaker, or Microsoft’s Excel or Word, which run on workstations or PC’s. They mostly present themselves as a collection of tools that is very weakly structured. The available tools of such environments are often controlled by some form of manager, whose purpose is twofold: The manager allows users to control individual tools, by letting him or her start and stop tools (or instances of tools), and its main task is to let users control the environment as such, by starting and safely ending sessions with it. The problem with these environments is that for the user it is often not obvious which connections between tools really are available and what their meaning is. Furthermore, there often exist too many such potential connections between tools, which leads to confusion of the user, because the screen gets cluttered as new windows pop up and cover others that can no longer be directly accessed. In such environments, windows also often just disappear without warning, because they are no longer needed, or change their position because they are needed somewhere else. Another problem with this approach is that the connections between tools are fixed and cannot be changed. Depending on personal preferences, users can make use of them or not, but there is no way of adding new connections between tools where they appear useful, or of removing connections if they are no longer needed.

The Wireable User Interface (WUI) operates primarily on the level of the user interface of the tools and the tool environment as such. Therefore, it offers a clean and consistent way of integrating tools into a tool environment not only on the level of the data they share and on the level of control between tools, but also on the level of the user interface. The concept of the WUI is flexible enough to be applied in a wide range of tool environments. By letting the users ´wire´

their own configurations of tools, it allows users to define by themselves which tools should work together and how they should do it. Since the WUI also contains a configuration mechanism, users can save their ´wirings´ for later reuse, or they can make use of configurations that have been set up by other users. The latter, of course, is particularly convenient for less experienced users.

The concept of the WUI has first been designed as the tool integration mechanism for AEMES (An Extensible Maintenance Engineering System), whose aim is the design and implementation of a tool environment for software maintenance. The WUI approach has since been generalized, but the examples sketched here are taken from AEMES.

The remainder of this paper is organized as follows: In the next sections we present the basic concepts and mechanisms that are needed for the realization of the WUI. Section 3 shows how these mechanisms are used to achieve tool integration in the WUI. In section 4 we illustrate the ideas of the WUI by showing an example of how it is applied in the AEMES project. We close the paper with conclusions and an outlook on our future work.

2. Basic Tool Integration Concepts of the WUI

2.1. The Workbench Paradigm

The WUI is based on a tool environment paradigm that we call the *Workbench Paradigm*. The workbench paradigm is strongly based on the idea of a real-world workbench and therefore makes it easy for a user to get an intuitive understanding of the tool environment when working with it. In the real world, objects are brought to a workbench in order to inspect them or to perform manipulations on them (like, e.g. adding, removing or changing parts). A vice is used to position and hold the workpiece, and all the tools are laid out on the workbench and can be used on the object on the vice in almost any order and in various combinations. On the WUI, exactly the same is done with the virtual objects on a computer:

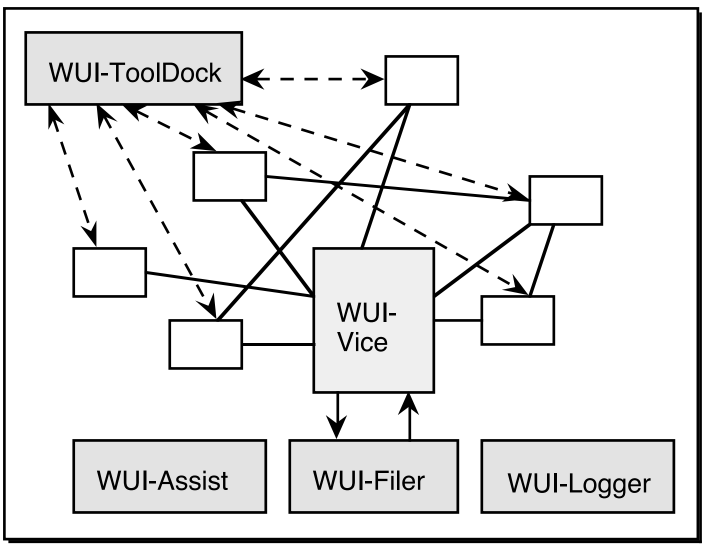

Fig. 1: Tools and Meta Tools in the WUI and how they are linked

As we can see in Fig. 1 on the left, there is a vice, called the *WUI-Vice*, in the centre of the workbench, and all the available tools are positioned around it in order to be quickly at hand. The object on the vice can be a graphical object, a spreadsheet, a document, or, in the case of the AEMES environment, it is the program that is currently maintained.

2.2. Tools of the WUI

The WUI-Vice, together with the other tools that are shaded in Fig. 1 belong to a class of tools, called the *Meta-Tools*, because they serve special purposes in the WUI. All the other tools are simply called *workbench tools*. The WUI-ToolDock plays the role of a manger for starting and ending sessions with tools, and also for configuring the working with tools. WUI-Assist is a user help and support system and WUI-Logger keeps track of all the actions on the workbench. The WUI-Filer is there to load 'objects' onto the WUI-Vice and to unload them after the work is done. Thus, it provides the door to the outside world.

2.2.1. *The WUI-Vice*

Objects on the WUI-Vice can generally be rather large. Therefore, the WUI-Vice has several features that allow users to quickly locate relevant portions of the object. In particular, the WUI-Vice is able to perform the following operations on the object it currently holds:

- position a cursor at a specific place on the object
- highlight parts of the object
- hide or show parts of the object

It is, however, very important to notice that the WUI-Vice never decides *by itself* where to place the cursor, which regions to highlight, or which ones to hide, respectively. This is either done directly by the user, or it is determined by the workbench tools, which tell the WUI-Vice what to show (or hide) and how to show it. In this respect, the role of the WUI-Vice is a *passive* one — the Vice Tool does only what other tools ask it to do or what the user tells it to do. This is very much like in our real-world analogy, where the vice is used to place and position an object, but other tools are needed to manipulate it. Like in the real world, where it can be useful to have more than one vice (or even slightly different vices) when one desires to work on more than one object at a time, there can be several instances of the WUI-Vice, too.

2.2.2. *The WUI-ToolDock*

The WUI-ToolDock is there for the users to configure the workbench tools. It plays the role of a manager for starting and ending sessions with tools and with the environment. But more important, it also allows the configuration of the workbench itself and thus permits a user to define which tools work together and in what way. Thus, it allows the user to define the *wiring* of the tools.

We will say more about the WUI-ToolDock and how it does the wiring later on.

2.2.3. *Other Tools of the WUI*

We have named the tools that are not greyed out in Fig. 1 the workbench tools. These tools can best be characterized as the tools of the environment that do the actual work, i.e. they pertain to the actual task of the tool environment. The aim of the WUI is the integration of these tools. We will say more about them in the next section and give some examples of such tools in section 4.

2.3. Identifiers and Coordinates

We have mentioned that an object that is put on the WUI-Vice can be displayed using several options, like hiding/showing, or highlighting parts of that object. In order to be able to select the appropriate parts, we must be able to identify them. We do this by first splitting the object into well-defined parts, and then by assigning unique identifiers to the obtained parts.

The splitting is in fact done by partitioning the object, and there is large freedom in how exactly this is done: if the object on the WUI-Vice is a graphical object, then the parts are the lines, arcs, rectangles, and circles, and even bitmaps. If the object is an ASCII-text, then either every single character can be considered a part, or every word and every punctuation mark can be seen as a part. But even every line could be taken as a part. In a document, where both, text and graphics, can appear, parts are a mixture of both, parts of text objects together with parts of graphical objects. In our maintenance environment, we split the program that is kept on the WUI-Vice into its characters, simply because it is the easiest way of partitioning the program (splitting into tokens could also be done, but this would require scanning – or even parsing – of the program).

There are three things that have to be taken care of when partitioning the object on the WUI-Vice: first, nothing that belongs to the object can be excluded from partitioning, second, parts may not overlap, and third, there has to be a consistent level of detail when partitioning the object. While the first two requirements are trivial, the third needs some explanation: What is considered a part of the object on the WUI-Vice is also of importance to the tools on the workbench, because the tools will share the same view of parts as the WUI-Vice does.

We now assign unique *identifiers* to all the parts by using the bijective function

$$\mathtt{f_{id}(partOfObj) = id}$$

whose domain are all the parts of the object on the WUI-Vice, and whose range consists of the identifiers that are assigned.

For most objects, also the ordering of the parts is of importance (e.g. in the character partitioning we have chosen in our maintenance environment, the ordering of the characters in the program is fixed and must be remembered, in order to preserve the program's integrity). For such objects, the ordering must be preserved in the identifiers, too. Thus, the following must hold

$$\mathtt{partOfObj < partOfObj' \Rightarrow f_{id}(partOfObj) < f_{id}(partOfObj') = id < id'}$$

for all the objects and identifiers. In this case, we will also talk about *coordinates*, instead of identifiers for the parts of the object.

Because we defined function f_{id} to be bijective, there exists an inverse function f_{id}^{-1} with

$$\mathtt{f_{id}^{-1}(f_{id}(partOfObj)) = partOfObj}$$

and

$$\mathtt{f_{id}(f_{id}^{-1}(id)) = id}$$

We additionally request f_{id}^{-1} to be computable reasonably efficient.

The following picture illustrates the concept of partitioning and identifying the resulting parts:

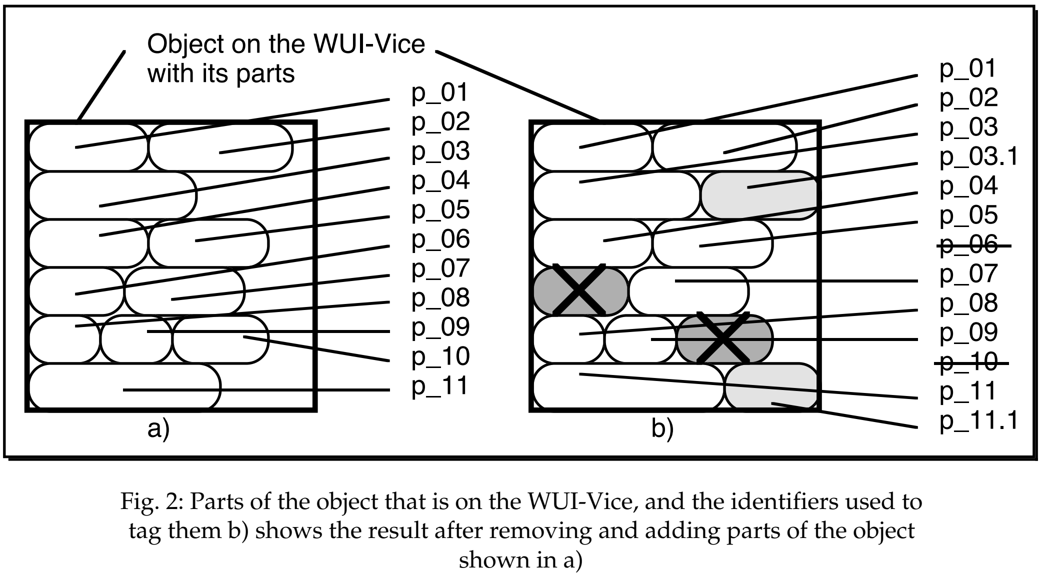

Fig. 2: Parts of the object that is on the WUI-Vice, and the identifiers used to tag them b) shows the result after removing and adding parts of the object shown in a)

We can see that in a) every part of the object on the WUI-Vice is uniquely identified and corresponding identifiers are properly ordered. This is still true in b). There, new parts that have been added obtain new identifiers that can be put into the appropriate place among the existing identifiers, so that the ordering is preserved. Parts that have been removed cause the corresponding identifiers of these parts to be no longer valid.

Note that there can also be sets and ranges of parts. A *set of parts* is just a loose collection of parts that can be designated by the collection of corresponding identifiers. A *range of parts* can only be given if the parts are ordered. The range is identified by two identifiers designating the borders of the range.

In our example of the maintenance environment, where individual characters are the parts of the object on the vice, we use a very simple identification function: we assign each character a *(l/c)* coordinate, indicating the line and column, respectively, of that character, and starting with coordinate *(1/1)* at the upper left corner, which also has the advantage that f_{id}, as well as f_{id}^{-1}, can be computed rather fast. If new parts are inserted into the program, new coordinates that preserve the ordering can be found easily, by, e.g., using two-level indicators for row and column, like *(l.l/c.c)*.

3. How do Tools work together?

After having explained the basic structure of the WUI environment (the workbench paradigm) and after having talked about the object on the WUI-Vice and the identifiers (or, in some cases, coordinates) assigned to the parts of it, we are now ready to explain how tools in the WUI work together.

3.1. Slots of tools

The reason we gave for introducing identifiers for parts of the object was to have a means to designate specific parts of the object, in order to be able to perform the various display functions the WUI-Vice offers. But the significance of identifiers goes much further, because they are at the same time the backbone of all communication among the tools on the workbench.

We have said earlier that the role of the WUI-Vice is a passive one, and that it is always the other tools on the workbench that tell the Vice what to do. They can do that, because the WUI-Vice has three so-called *slots*, namely, the locator slot, the selector slot, and the filter slot. All three slots are *input slots*, meaning that they receive identifiers (or sets of identifiers, or ranges of identifiers) from other tools, which are connected to these slots. The identifier at the locator slot tells the WUI-Vice where to position the cursor on the object it currently holds, whereas identifiers at the selector and filter slot specify which parts of the object should be highlighted, or hidden, respectively.

The WUI-Vice does also have an *output slot*. The output slot contains the identifier of the part of the object the user is pointing to at the moment, if the user is pointing at several parts this slot contains a range or a set of identifiers.

However, not only the WUI-Vice has input and output slots containing identifiers. Also, all the other tools (except for the other Meta-Tools) of the workbench may have an input and an output slot. Now the circle is closed: The WUI-Vice has an output slot, which may be connected to the input slots of the other tools, and their output slots can, in turn, be connected to one of the input slots of the WUI-Vice. But not only that, input and output slots of workbench tools may also be freely connected among each other.

By connecting input to output slots, a whole network of connections among tools can be realized. Thereby, we don't have to worry about the compatibility of the information the tools exchange, since all they ever exchange are identifiers or sets and ranges thereof. The one restriction is that all the tools share the same view of parts of the object on the WUI-Vice, and thereby, of the identifiers that are assigned to them. This also leads us to a new view of tools: Tools do not work directly on their input data, but on the interpretation of the identifiers (the parts of the object they designate) and they translate their results back to identifiers, that can be further passed to other tools. The following picture illustrates this:

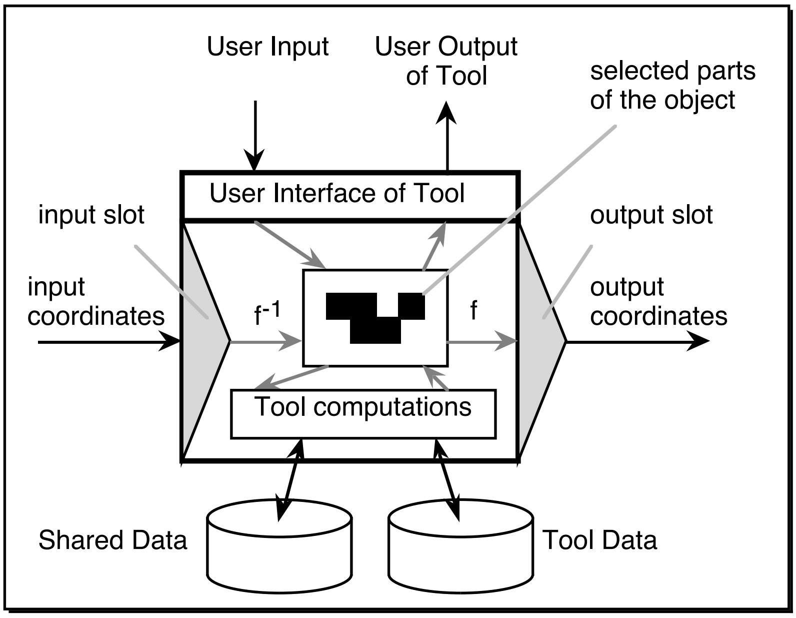

Fig. 3: A tool with its input and output slot. Note that the user interface of the tool may – through user settings – also influence the computations of the tool. This is not depicted here for simplicity and clarity of the drawing.

This illustration also shows, how the user works on the interpretation of the identifiers (thus, the parts of the object), but not on identifiers themselves. The identifiers are, in fact, never directly visible to the user.

For practical reasons, we distinguish between four classes of tools, based on the slots they provide. Tools with only an input slot are called single-input-no-output (SINO) tools, tools with only an output slots accordingly, no-input-single-output (NISO) tools, and tools with both, input and output slots, are called single-input-single-output (SISO) tools. Naturally, there can also be no-input-no-output (NINO) tools on the workbench, although these tools cannot be connected to other tools (and might therefore not be very meaningful in the environment). Note, that the WUI-Vice belongs to a special class of tools, three-input-single-output.

3.2. Wires

The process of how identifiers are propagated from output slots of tools to input slots is very simple: Whenever the identifiers on an output slot change, the tools which receive these identifiers at the input slots are notified of the change.

Theoretically all the tools of the WUI-workbench could be connected to all the others. In practice, this would lead to great confusion on the user interface. Therefore, we want to be able to determine which tool communicates with which of the others, and we do this by introducing *wires*. A wire between two tools A and B simply says, that the output slot of tool A is connected to the input slot of tool B.

There are certain conditions for establishing wires between tools, in order to keep the system clean and consistent:

- An output slot of a tool may not be wired to its input slot. Although one could think of meaningful examples of such wires ("feedback"), this kind of situation can be caught within the tool itself, which is also much better design.
- An output slot of a tool may be wired to several input slots of other tools, but multiple output slots may not be wired to the same input slot. This would lead to contradicting identifiers at the input slots, when several tools deliver coordinates.
- A distinction has to be made between three types of identifier input, namely, single identifiers, sets of identifiers, and ranges of identifiers. If, for a tool, one of the three types is not meaningful, such wires are prohibited. Note that there is usually a hierarchy of the three types: sets of identifiers '>' ranges of identifiers '>'

single identifiers, meaning that tools that can handle sets of identifiers can also handle ranges of identifiers, which can also handle single identifiers, but not vice-versa.
- It is considered good style if most of the wires lead from or to the WUI-Vice, in order to emphasize its special role.

The second and the third requirement can actually be overcome by introducing a set of tools that do not pertain to the actual task of the environment, but to the WUI itself. Examples of such tools are *switches* (which can take several inputs and have one output), or *selectors* (which can select individual identifiers out of a set or a range of identifiers), but it is beyond the scope of this paper to go into more detail here.

Wires can be either *manual* or *automatic*. Automatic wires, as the name implies, automatically forward the information from the output slot to the input slot they are connected to. Manual wires do the forwarding only on the immediate request of the user. If a manual wire contains new identifiers at its end (which is the input slot of a tool, actually), there is some visual feedback to the user that the display of the tool is no longer up do date but can be updated, since new identifiers arrived. The user can then choose to accept these new identifiers.

Wires can be changed between manual and automatic any time. Note that cycles in the configuration of the tools are not a problem, as long as not all the wires in the cycle are automatic. Wires that go through the WUI-Vice are generally not a problem, since the input slots and the output slot of the WUI-Vice are not directly connected, as the output slot contains the identifiers of where the *user* directly put the cursor.

3.3. WUI-ToolDock and WUI-Logger

The wiring of the tools is done in the WUI-ToolDock that we have introduced in the previous section. The WUI-ToolDock is also responsible for enforcing all restrictions given above. Furthermore, the ToolDock allows to define different configurations of tools and wires, and the fast switching between such configurations. The WUI-ToolDock should also contain a small plan of the current screen layout (positions of tools and wires, including their status), so that users can easily and quickly remind themselves of the current screen configuration and the status.

The WUI-Logger keeps track of all the identifiers that have been exchanged between the various tools over the wires. The information kept can be used to undo operations of the user or to replay earlier actions of the user any time.

4. An Example

Let us now take up the example of the software maintenance environment we have already mentioned earlier in this paper and show how the concept of the WUI is used for this environment.

4.1. Some Tools

We have said that there exist four different classes of workbench tools, depending on the existence of input and output slots. Two examples of SINO tools in our maintenance environment are a *where_belongs* tool and a *what_is* tool. The where_belongs tool takes the coordinate at the input slot, determines the corresponding part, and tells the user where that part belongs to, i.e. to which module, to which procedure, and to which program, etc., so that the user quickly sees, where in the program he or she is. The what_is tool looks at the part and tells the user if it is a keyword, a variable, a procedure name, or whatever.

A typical example of a NISO tool is a *cross-referencer*. It presents the user a list of all symbol classes, the symbols contained in each class, and for each symbol, also its occurrences. The user may then select a particular occurrence or a list of occurrences, whose coordinates are, in turn, laid to the output slot of the cross-referencer. Another NISO tool quite similar to the cross-referencer is a *program browser*.

Examples of a SISO tool are the *where_from* and *where_defined* tool. They interpret their input slot and display some information regarding the data at the input slot. The where_from tool then puts to its output slot the coordinates of all the locations where we could have reached the location indicated at the input slot from. The where_defined tool gives at its output slot the coordinates of where the part at the input slots has been defined. Of course, not all the parts of the program have been defined somewhere (like, e.g. keywords), which also shows, that there is not always a meaningful interpretation for every possible coordinate at the input slot of a tool. In this case, the user has to be informed of what happened by some visual feedback of the tool (like greying out the tool window) and no coordinates are put to the output slot. Good design of the tools clearly seeks to avoid that this occurs.

Examples of NINO tools can easily be found, like a tool that shows e.g., the creation and modification date of the object on the WUI-Vice, or its size. These tools are the least interesting ones in the environment, since they cannot be wired to others, but they can be meaningful nevertheless.

These are just some examples of tools. Of course, there can be a huge number of other tools.

4.2. Working with the WUI

A simple example for wiring is to wire the cross-referencer from above to the WUI-Vice. Here, the user has a choice: if the cross-referencer is wired to the locator slot, the user can quickly jump around the program. If the cross-referencer is wired to the selector slot, the relevant positions of the program can easily be found since they are highlighted. By wiring the cross-referencer to the filter slot, relevant portions of the program on the WUI-Vice appear close together.

A more complicated example for wiring: The user wires the where_defined and the what_is tool to the output slot of the vice. At the same time, the user wires the output slot of the where_defined tool to the selector slot of the WUI-Vice. We assume, that all wires are set to automatic. Now, whenever the user points to a location on the object on the WUI-Vice, he or she quickly sees what is there, and where it is defined.

There are many more examples for wiring, and wiring can go through many more tools. Depending on the preferences of the individual user, there can be many or few, short or long wires. Crucial for a user's motivation to use wires is the careful design of tools, which can be wired in many meaningful ways. But careful design of tools has always been important for tool environments.

5. Changes in the WUI

So far, we have only discussed the tools of the WUI-environment in terms of the retrieval and information functions they perform, and have not mentioned the aspect of making changes to the object on the WUI-Vice. There are, of course, a number of ways to deal with changes in the WUI. Due to the limited space of this paper, we will only present one possibility here, of which we think that it is a simple, yet powerful solution.

The basic idea of this approach is that changes to the object on the WUI-Vice can potentially be made in any tool on the workbench, but that the changes are only propagated when they are finished and the user commits the change to the system. When the user commits the change, the changed data is written back to the database that keeps a representation of the object on the

WUI-Vice. After that, all the other tools obtain a notification from the changing tool telling them that they are no longer working with the most up-to-date version of the data. This is illustrated in the following figure:

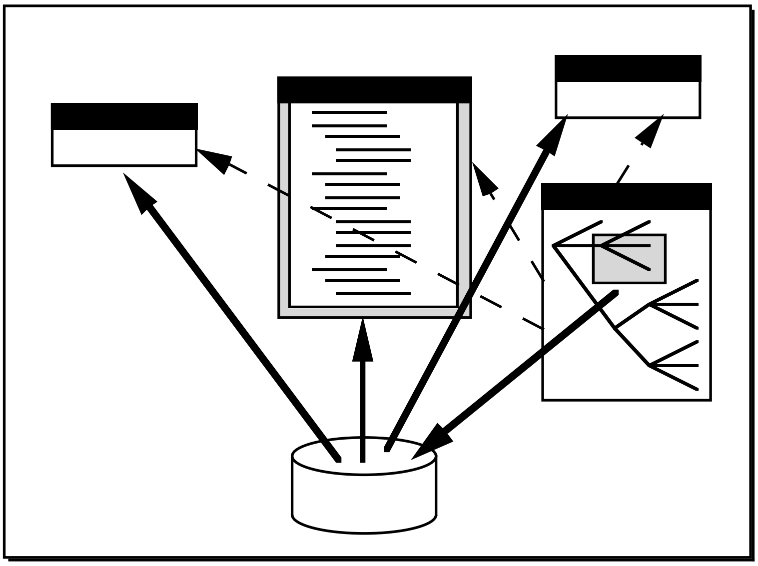

Fig. 4: A changes mechanism for the WUI

The change notification mechanism itself can be very simple, just informing the tools that they no longer display and work with the latest version of the data, but it could also give the tools some more information, i.e. by letting them know which coordinates/identifiers are no longer valid and which ones have been added newly. The latter, of course, would more easily allow the incremental update of the information of the tools.

Once a tools gets a change-notification, it is up to the individual tool how to respond to it. A tool can immediately update its information, or it can wait until the user wants it to update it, in which case the user should be given some visible feedback, telling that the data the tool is working with is not up-to-date.

6. Conclusions and Future Work

The Wireable User Interface presented in this paper is a powerful concept for integrating tools with graphical user interfaces into a clean and consistent environment. It is general enough to be used for a wide range of different tool environments, and it is open in the sense that new tools can easily be added to the environment without having to re-implement the already existing tools. Large parts of the WUI are independent of the particular purpose of the environment and can thus be reused.

The WUI is currently being implemented in C++, based on the application framework ET++ [GW90] for using and testing the concepts presented here in the software maintenance project AEMES. Tools in the WUI consist of one main window, whose frame contains (visual or textual) information about the wiring, the current status of the window, and the slots of the tool. This calls for a basic WUI-Tool-Window class, that, on the one hand, implements this window frame, and, on the other hand, processes the information that tools send and receive at their slots. Ideally, the Meta-Tools and all the user interfaces of the workbench tools are then compiled into a single User Interface Application, which is implemented as a client to a server, which implements all the functionality of the workbench tools.

One major aspect of the WUI could not be addressed in this paper, due to the limited space available. We have mostly been talking about one instance of the WUI-Vice (or even different variants of the WUI-Vice), and have also not discussed instances of workbench tools in general.

Having various instances of workbench tools has implications on the wiring, and, more dramatically, allowing various instances of the WUI-Vice potentially allows wiring between different objects that are on different vices. All these aspects are currently being further investigated.

6. Literature

GW90 E. Gamma, A. Weinand, “ET++: A Portable C++ Class Library for UNIX(TM) Environment” OOPSLA’90 Tutorial notes.

MEYE91 Scott Meyers, Difficulties in Integrating Multiview Development Systems, IEEE Software, Vol. 8, No. 1 Jan 91, p. 49-57

OUST91 John K. Ousterhout, “An X11 Toolkit Based on the Tcl Language”, Proceedings USENIX Winter Conference 91

REIS90 Steven P. Reiss, “Interacting with the FIELD environment”, Software Practice and Experience, Vol. 20(S1) Jun 90, p. 89-115

SCHE93 Dick Schefström, "System Development Environments: Contemporary Concepts", in "Tool Integration - Environments and Frameworks", by D. Schefström and G. van den Broek, Wiley Series in Software Based Systems, 1993