Top-Down Object-Based User Interface Definition and Design Paradigms

ANDY HOLYER

School of Cognitive and Computing Science,
University of Sussex,
Falmer,
Brighton BN1 9QH
United Kingdom
Email: andyh@cogs.susx.ac.uk

Abstract: Currently, the main emphasis in User Interface design tools is on the low-level manipulation of Interaction Components, such as widgets. This paper proposes a design architecture which approaches User Interface development in a top-down manner, to avoid particular shortcomings in current design methodologies.

1.0 Introduction

Currently a great deal of attention is being devoted to producing tools to aid User Interface design - in particular the design of direct-manipulation interfaces, which are generally considered to be the hardest to produce [4]. The guiding philosophy behind many of the User Interface design tools currently being developed is to provide the sort of functionality found in direct-manipulation "drawing" packages within a system which produces a working User Interface. Combined with progress towards modularisation in User Interface components - for example "widgets" found in the X windowing system - this does offer a considerable reduction in the difficulty of producing a User Interface.

The vast majority of User Interface systems which currently exist implement interface elements procedurally - that is, a UI designer must produce a *program which, when run, produces the interface desired.* Hence a "widget drawing tool" such as those currently existing provides a level of insulation between the designer and this procedural level of User Interface description. Her task is further made easier by the provision of direct visual feedback while the User Interface is being constructed - thus making it much easier to get cosmetic details such as colour and layout right.

However this approach to aiding User Interface design is in conflict with current views on HCI, which emphasize examining the interface from the top downwards, and avoiding committing the design to any particular implementation for as long as possible. That these methods are desirable is not in question; however, a designer, using current User Interface development tools, must do all the top-level analysis by hand, and merely use the UIDT to help implement the final design which the other methods have produced. This paper suggests an architecture for a projected User Interface Design Tool which would allow a designer to perform a complete top-down design process within one unified tool.

2.0 Levels of description of a User Interface

There are a number of levels of description, of varying degrees of abstraction, which can be applied to a User Interface. It should be emphasised at this point that each of these descriptions can be considered complete in

itself - that is, any User Interface within reason may adequately be defined in, or designed using, any of the following formalisms. However, there are certain advantages in using high- rather than low-level formalisms; The resulting User Interface is less platform- and version-specific, and hence may be more easily ported onto other platforms or updated as new versions of the application are produced; more effort is required using a low-level formalism to make the resulting UI stylistically and functionally consistent.

As I mentioned above, almost all User Interfaces are, at the lowest level of description, eventually instantiated as imperative program code. By this I mean that at some point there exists a program which, when run, will effect the user interface. Note that this is not so in the case of, say a document produced by an object-orientated draw program, or of a document produced by a word processor, both objects in principle of comparable complexity to an interface we could conceive of generating with an "Interface Processor". Note also that declarative programming languages such as NeWs are still nonetheless programming languages, and there is still an abstraction inherent in their use - that is, there is still "code" which must be produced and then "run" to check the results. Note that until the adoption of Direct-Manipulation User Interfaces, there was virtually no theory of interaction at above this level.

The next level of description in a User Interface may be described as the "widget" level, since this is the level at which X11 widgets are declared. The interface is described in terms of a collection of primitive graphic objects which are slotted together (normally in a parent-child hierachy) to create the desired effect. Some explicit programming is still required: A system call (or more than one in some cases) is required to generate each instance of a widget, and further calls are necessary in order to create the dependencies between widgets and to define each widget's behaviour. In addition such matters as memory management must still be programmed explicitly. However, much of the UI's functionality is standardised - buttons have standard behaviours and are consistently animated when triggered, menus and scroll bars work in a consistent manner automatically. At first sight, this is a major advance over the "roll-your-own" approach - as indeed it is: however there are still major problems with designing UIs using this approach.

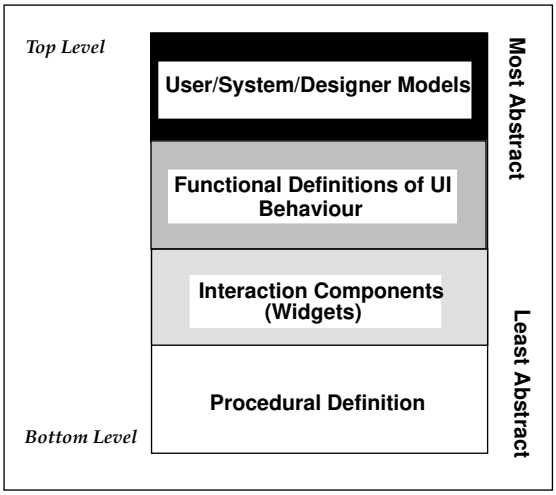

Fig. 1. Levels of Description of a User Interface

Most immediate of these is the sheer size of a widget-based toolkit, which means that in practice very few UI designers will learn to use any particular toolkit proficiently. In addition, the very power of the widget-sets reduces their flexibility, and, most significantly, makes it far harder to port a User Interface from one set to another. For example, the various X widget sets are to a large degree orthogonal to each other. By this I mean that for the more sophisticated and hence more powerful widgets in each widget set, there do not exist direct equivalents in competing widget sets. Hence a designer wishing to produce a UI on more than one widget set is faced by a choice; either do not use the more sophisticated widgets provided by a particular widget set, or potentially redesign the UI from scratch for each other widget set required. Obviously, neither of these approaches are to be desired.

These immediate difficulties are what most current User Interface Design Tools are meant to address. By allowing widgets to be moved around dynamically, they remove one abstraction which a designer would otherwise be required to make - that is transforming the desired form of the interface into a program which would produce that interface. However, they fail to address higher issues such as consistency of style and role integrity [5] and the correspondence between User and System Models.

It is generally accepted that the top level of abstraction of a User Interface is in Norman's concept of Models ([2],[6] and also Fig. 2.). Briefly, this posits that when a designer is producing a system, she internalises a model of how the system behaves (the "Designer Model"), which is what (in theory) she instantiates in the final system (the

"System Image"). The User, when he is using the system, similarly forms a model (the "User Model") of what *he* believes is happening in response to his actions, and acts accordingly. Problems occur when, as frequently happens, the User Model which emerges differs from the model from which the designer was working. Just to make things a little more interesting, it's surprisingly common for the System Image to differ (sometimes quite radically) from the Designer Model. Both Norman and Thimbleby give extensive examples of everyday systems in which this kind of design error occur (for example, see [3] and [7]).

3.0 Current theory on User Interface Design methods

From the theoretical principles stated above, as well as from existing engineering principles of structured design, a system has begun to be developed which recommends a methodology for structured User Interface design. The method specifies a series of stages which a design should undergo, beginning with the most abstract, and ending with the nearest level of abstraction to the actual User Interface itself. Note that there is no reason that this process is only applicable to the design of computer systems; most of the principles are equally applicable to the design of other devices, such as consumer appliances and vehicles (within the HCI literature, a great deal of attention tends to be devoted to the design of video recorders, microwave ovens and cars; possibly this reflects on the obsessions of HCI researchers).The suggested method is summarised in Fig. 3. I will briefly examine each stage involved in this process in order:

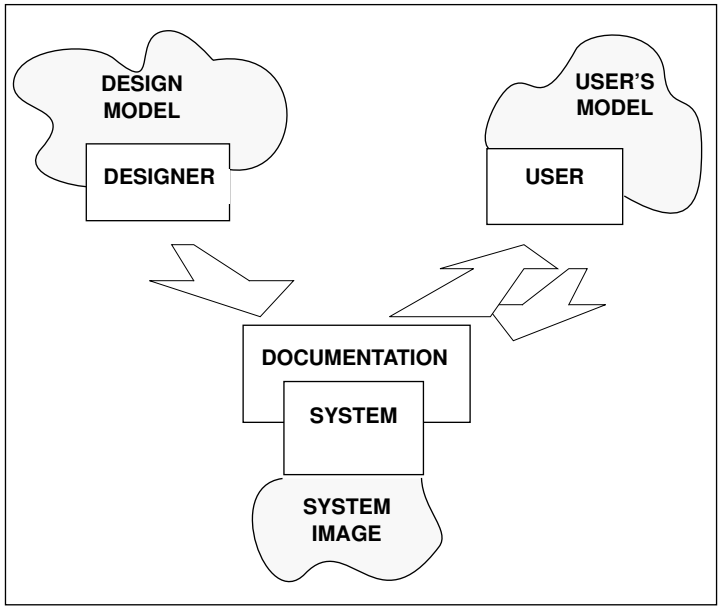

Fig. 2. Norman's "Three Models" (After Norman & Draper "User Centered System Design", 1986, p.46)

4.0 Specify the Problem

At this stage of development, the designer should definitely not be thinking about implementing the required task; The only attention of the design process should be on examining the problem as it exists currently - i.e. before the new tool is developed. In the case of a totally new application, the designer should pay attention to the way the application domain is currently handled by other means; in the case of an existing application for which a new interface is being developed, it is fruitful to examine user's qualitative experiences with preexisting systems. This process should yield a task description, which is of use in the next stage:

5.0 Derive Requirements

Taking the task description, the designer should develop a requirements specification. This specification should examine what a user should be able to do with the application, what operations should be made easy to do (an in some cases what it is desirable to stop the user from doing). This information is used in the next stage:

5.1 Produce Functional Design

At this stage in the design process, one finally begins to approach implementation issues. The functional specification should describe the things a user can do with an application, and give some indication of the mechanisms by which they may do each of them. However, the functional design should not look at actual implementation details; the same functional design should be implementable in a wide range of hardware environments, without alter-

ation. For example form one functional design, it should be able to produce a user interface implemented on any of several different windowing systems, on a character terminal, or on a custom-built console without modification.

5.2 Produce Specifications

Finally, given a functional analysis of out desired interface, we can begin to specify the interface in target-technology terms. Paradoxically from some points of view, this is the first time that we actually see what the interface will look like. It is only at this stage that decisions about the physical form of the interface are made - menu organisation, control panel layout, etc. At this point we are at last able to do what many designers think of first:

5.3 Design Interface

This is the stage in the design process normally considered when we think of "User Interface Design", and is the context which most widget-level construction tools are positioned.

6.0 The Problem

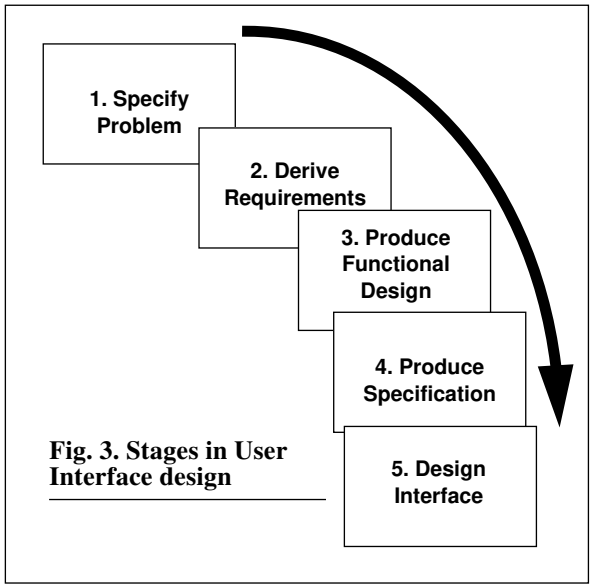

Fig. 3. Stages in User Interface design

It is clear from the above that various advantages accrue from designing a User Interface from a top-down approach. Firstly, the interface is likely to be more consistent; it is disconcerting to have certain features, for example an "Undo" button present on some dialogs and not on others - if the designer decides on the abstract functionality of each of her "dialogs", and then extends that to cover particular cases there is less chance that such stylistic clashes will occur.

In addition, many user interfaces may be very nicely engineered at a menus-and-buttons level, while exhibiting quite glaring flaws at higher levels of design. Norman's concept of the "gulf of execution" comes in to play here [2]; there are occasions when the user understands very well how to operate the interface on a simple level, but is unable to find out how to do a particular operation (e.g., "how do lay out my text evenly spaced down the page?"). This is sometimes the result of the user trying to do something the designer never expected the user to need to do, but this in itself is a sign that the User and Designer Models are badly out of sync; this could be avoided if the design process had begun at a higher level of abstraction.

A third advantage of high-level design is that the design remains flexible until much later in the design process than would otherwise be the case. Thomas Green has proposed the concept of the "viscosity" of a user interface [1]. This describes the effect whereby, as development task continues (whether it is developing a program, writing a document or designing a car), the further the development progresses, the more fixed the various elements of the design become; the temptation to keep a flawed design rather than waste resources and effort by abandoning it becomes stronger and stronger the more work is done on the design. Top-Level design keeps the design process at a high level for as long as possible, and thus similarly keeps the overall flexibility of the design as high as possible for as long as possible.

Other related advantages include the possibility of producing relatively system-independent designs (which may be ported to various target environments), and the fact that high-level design tends to encourage modularity (thus all control panels in an application would tend to have similar layout and pragmatics, for example).

Unfortunately the tools which currently exist to build user interfaces only aid the designer in step 5 of the diagram - actually constructing the interface. For the earlier (and very important) stages of design a computer system is of no more help than pencil and paper - and may in fact impede development by trammelling development in a similar direction as that of the computer system being used! The challenge, then, for the designer of a User Interface Design tool is to produce a tool which can make at least some progress upstream in the design process.

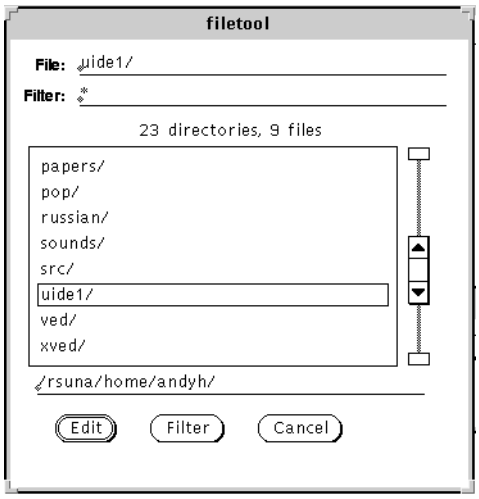

Fig. 4. The UIDE filetool

7.0 An attempt at a solution

The UIDE Project has so far developed a User Interface building tool which is restricted to the widget level; as part of the next stage in the project we are currently devoting considerable research effort into producing tools which allow the designer to work at the intermediate User Interface specification level shown in fig. 1; the level of functional description of a User Interface. This corresponds in the design process to boxes 3 and 4 in fig. 2. To handle this level of abstraction, we have posited a new sort of User Interface object; the Functional Interface Component, or FIC. An FIC is an element of a User Interface which is described in terms of its abstract function, stripped of any implementation details.

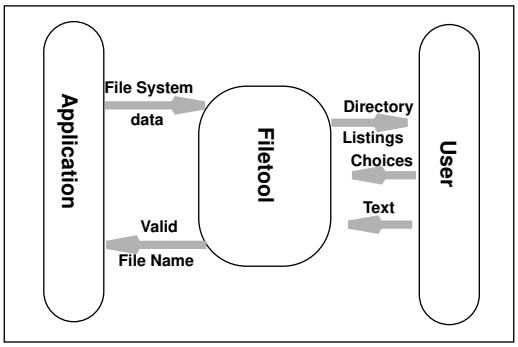

Fig. 5. The Filetool as an FIC

Working with Functional ICs is greatly aided by the use of an "object-oriented" approach to the design. This is the approach which UIDE takes (using the Poplog object-orientation system, "flavours"), although it should be emphasised that these principles are still applicable without the use of object-oriented approaches.

An FIC is purely an abstraction for the convenience of the designer; in a final interface a lower-level Interaction Component (or group of ICs) would form the instantiation of the FIC. This is a subtle point, and it is important to be clear on it. It is also true in fact of widgets; talking in strict terms there is no such "thing" as a "widget" - it is merely a convenient abstraction for a data structure eventually implemented through a set of calls to graphics primitives. However it has become so natural for designers to talk about interfaces at the widget level that it requires quite an effort of will to remember that one is only talking about procedural data, and not, say, an *actual* button that one could handle.

One consequence of this abstract nature of an FIC is that the mapping from FIC $\longrightarrow$ IC is a one-to-many mapping. That is, a collection of different Interface Components (possibly even from different User Interface environments) would all be subsumed at the functional level as one class of FIC.

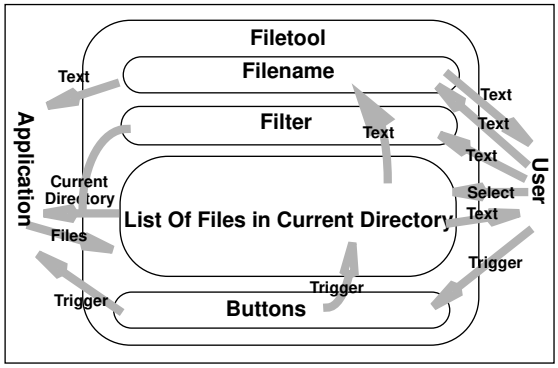

Fig. 6. The Filetool broken Down

Each FIC is considered to be an object which passes messages of particular types to and from other objects; within any particular User Interface there exists one (or more) objects called "The User" and one (or more) objects called "The Application". A complex and sophisticated FIC may be decomposed into a collection of other FICs, which themselves may be decomposed, and so on. Hence the entire User Interface may be considered one giant FIC.

For example, consider the File Selector shown in Fig. 3: It is possible to describe this tool in Functional terms as the single Functional IC as shown in Fig 4; this FIC may be decomposed as shown in Fig. 5 (assuming the existence of Application and User objects):

Each of the subcomponents of the filetool is itself an FIC. Notice that the FICs labelled “List of Files in current directory” and “buttons” themselves have a substructure, and could in turn be decomposed. Note also that the filetool shown here is instantiated under the OpenLook widget set; the FICs could equally well be instantiated under a different look and feel while retaining the same functionality. Hence the designer is able to concentrate on top-level description of the interface, without getting bogged down in implementation-specific details, thus remaining in line with the principles proposed earlier.

FICs may be considered to be organised in a hierachy arranged according to the type of information they send or receive. Research is still under way to produce a logical set of primary FIC classes which gracefully include all desired types of User Interface objects but it is clear that at the highest level of functional abstraction, there will not be many classes of FIC - probably less than 10. For example, Table 1 shows an (incomplete) enumeration, which results in 5 FIC taxa (additionally subdivided as to whether information flows from the User to the application or from the application to the user)

Another aspect of a User Interface which should be considered is the behavior of its components. An interface of any sophistication will inevitably have its own semantics independent of the application itself - certain buttons trigger certain menus, scrollbars alter the window shown by a container, resizing a viewport will entail changes in the settings of scrollbars etc. In addition, it is important for the designer that there is a uniform mechanism by which actions by the user on the User Interface are handled.

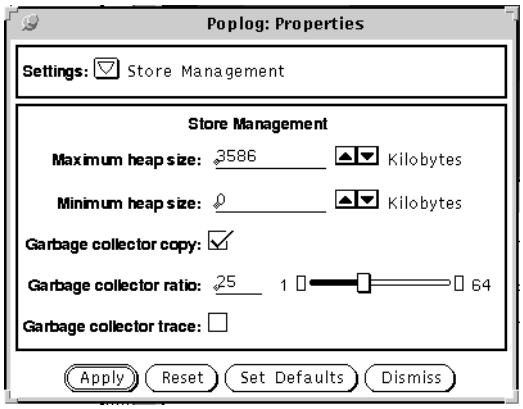

Fig. 7. The Poplog Properties Sheet

In a Widget-orientated User Interface, these are implemented as “events” which are applied to individual widgets; for example, when the user clicks on a button, a “select” event is sent to the button widget, causing some pre-defined behavior. Functional ICs also recognise evens; but whereas a widget’s event is intimately related to the widget itself, generally being specific to an individual widget-set for example, the “functional events” which an FIC will experience are expressed in terms of the abstract effect on the interface, and should not be specific to particular interaction components. These functional events are implemented as object-orientated messages;

For example, consider the control panel in Fig. 7.; two of the settings, “Garbage collector copy” and “Garbage collector trace” are simple boolean values which the user may set. It happens in this case that the designer has chosen to implement these selections using check boxes; however there are several other possible target implementations which may have been chosen - by use of a latching button, for example, or by a pair of radio buttons. In each of these cases the widget-dependent events produced by users making a choice would be different; however, each of these different target implementations would be an implementation of the same Functional IC - a “boolean selection” FIC - and would thus produce the same functional event (“true chosen” or “false chosen”) when a selection is made. Considering controls with more sophisticated semantics than a mere binary signal (such as the slider shown in the figure), the problem becomes much worse. The feedback produced by a Slider in any one widget is (in possibly some quite subtle ways) unlike that produced by the equivalent widget in any other widget set, and is in addition different from that produced by other ways of entering a numerical value (such as the two fields shown with a write-in field with up and down arrows). By viewing each of these fields as different instantiations of a single class of FIC, which merely supplies the application with a numeric value, it is possible to use one description regardless of the physical implementation details.

8.0 Conclusion and future plans

Top-Down interface definition offers the same sort of advantages for an interface designer as structured programming did for a high-level-language programmer when it was first introduced. What is required now are tools to enable a designer to use these methods without extra work; and theoretical work on the best ways to abstract the behaviour of user interface components in a natural and powerful way. My project's solution - largely in terms of dataflows - appears at this time to be a good one, but future research may reveal better approaches.

In the UIDE project, work is currently underway at the planning stage to refine the concept of Functional ICs and to produce a graceful infrastructure for their description. We are also looking at extending the paradigm to allow for direct manipulation interaction techniques - dragging with the mouse, rubber-band selection etc., and in producing the graphical paradigm by which a User Interface design process would itself become a direct manipulation operation.

There is obviously a long way to go before computerised tools exist to aid the designer in all areas of Interface design; however in our research we are at least attempting to go one step up the ladder.

9.0 Acknowledgements

This work is supported as part of the DTI/SERC project "User Interface Design Environment", Project No. IED/4/1/1577. Partners on this project are as follows:

The School of Cognitive and Computing Science, University of Sussex:	Dr. Phil Husbands, Mr. Ian Rogers Mr. Andy Holyer.
The School of Computer Science, University of Birmingham:	Prof. Aaron Sloman.
Integral Solutions Ltd.:	Dr Alan Montgomery, Dr Tom Khabaza, Mr. Julian Clinton, Mr. Ben Rabau.
British Maritime Technology Ltd.:	Dr. M. A. Lahoud, Dr. Jonathan. Cunningham.

10.0 References:

[1] T.R.G. Green, *"Describing information artifacts with cognitive dimensions and structure maps"* in *"Proceedings of HCI'91: Usability Now, Annual Conference of BCS Human-Computer Interaction Group"*, Cambridge University Press, 1991

[2] Donald A Norman, *"Cognitive Engineering"* in Norman and Draper, *"User Centered System Design"*, 1986 esp. pp. 45-48

[3] Donald A Norman, *"The Design of Everyday Things*" (formally published as *"The Psychology of Everyday Things"*), 1988, pp. 14-17

[4] Brad A. Myers, *"User Interface Tools: Introduction and Survey"*, IEEE Software, January 1989

[5] Harold Thimbleby, "*User Interface Design"* ACM Press 1990 p 130

[6] *ibid*, pp124-131

[7] Harold Thimbleby and Ian H. Witten, *User Modelling as Machine Identification: New Design Methods for HCI"*, University of Calgary Department. of Computer Science Research Report No. 91/436/20 (Especialy Example 2 on p.3)

Table 1: ICs Categorised by type of data flow

Direction:	Type of data carried	Potential ICs
$user \Rightarrow application$	Trigger	Button (Non-Latching)
$user \Rightarrow application$	Boolean	Button (Latching), Switch, Check Box
$user \Rightarrow application$	Choice (1 of n)	Radio Button, List, Menu (Pop-Up or PullDown)
$user \Rightarrow application$	Integer	Fill-In field, Increment/Decrement buttons, Sliders
$user \Rightarrow application$	Continuous value	Slider, (physical) dial, Mouse (controlling a pair of values, or ignoring one dimension), Joystick (ditto), TrackBall (ditto)

Table 1: ICs Categorised by type of data flow

Direction:	Type of data carried	Potential ICs
$user \Rightarrow application$	Text	*Various Sorts* of text fields (different manners of editing, scrolling, etc.) Handwriting/Speech Recognition?
$user \Leftarrow application$	Trigger	Bell, Screen Flash, Altered/flashing Icon...
$user \Leftarrow application$	Boolean	Highlighted Button, Altered Icon.
$user \Leftarrow application$	Choice (1 of n)	"Label" field, counter, gauge, colour coding...
$user \Leftarrow application$	Integer	Numeric Field, counter, bar-Chart, Histogram, graph, dial...
$user \Leftarrow application$	Real	As for Integer, with more detail; (also implementation differences)
$user \Leftarrow application$	Text	Static Text field, scrolling text field (scrolling left-right or up-down? with scroll-bars?)...