The Universe of Discourse—Rational GUI Design with Visible Context

Andrew U. Frank

TU Wien, Department of Geoinformation
Gusshausstrasse 27-29/E127.1
A-1040 Vienna, Austria
frank@geoinfo.tuwien.ac.at

for course "Haskell practice" SS 09
possibly journal article for "visual languages"
GZ 4753, words 3671, svn 3818

1 Introduction

Design of a Graphical User Interface is mostly limited to a description of operations for the user and information provided to the user. Coherence is at best achieved by guidelines. The many possible sequences of actions of users and their information needs are not considered in the design and are, therefore, often resulting in errors. Translation to program code is again difficult and effectively many of the artefacts of coding becomes confusingly visible to the user. The resulting code is hard to read and expensive to maintain.

Evers, Achten, and Kuper (2005) have pointed out that the current difficulties of coding GUI result from graphics API interfaces that are too low in the level of abstraction. They show that the low level of abstraction in the API can be recognized as single design decisions result in repeated and distributed code, with the potential to inconsistency and problems while changing the code. The optimal structure for a functional API for graphics is an open research question; the wxHaskell team has for intermediate level of abstraction to allow experimentation to find the "correct" abstractions. TV [], Fruit [], and wxGeneric [] are current experiments using wxHaskell; Fudget (Hallgren et al. 1995) are older proposals. Lindstroem (2008) and others have identified composability as a crucial property of conceptual GUI elements in Fudgets (Hallgren et al. 1995). Different proposals address different types of interactions. The very impressive TV [Elliot] approach uses a pipe concept to connect GUI units using arrows [], leading to a GUI style which combines the input and output of units, which provide data processing and graphical representation; inputs by the user propagate: the program reacts to user actions. This style of reactive GUI follows

Shneiderman's principle of immediate feedback to user actions (Shneiderman 1997). Elliot uses it mostly to create graphical (artistic) processes. I try here to show how a similar approach can be used in an administrative application.

The interaction of a user with a program can be compared with a conversation between humans (Nardi 1993). The analogy produces a rational framework for the design of a GUI. The analogy with conversations stresses the importance of "reactiv": the immediate reaction of my partner in a conversation—even if it is only a silent nod with head—is crucial. Novel is here primarily the explicit introduction of a universe of discourse for the conversation. The explicitly represented Universe of Discourse (UoD) is where the user's input is deposited, detected by the program and the reactions of the program and the reactions of the program result in changes of the UoD, which are then posted to the screen. The categorical viewpoint used leads to a formal definition of transparency of a GUI, which I think is novel and useful. The resulting code is better structured and smaller than previous programs (at least for the test case).

The contribution is structured as follows: section 2 explores the analogy between human conversation and human computer interaction and introduces the universe of discourse (UoD). Section 3 formalizes the idea, investigating primarily mappings, using mathematical category theory. Section 4 introduces the case study and section 5 discusses the resulting code.

2 Conversations

2.1 Conversational Implicature and Relevance Theory

Grice in his seminal work on conversational implicatures gave maxims of conversation to stress the necessary cooperation between speaker and listener (Grice 1989). His maxims presuppose an agreement between speakers about the code used to express meaning. Sperber and Wilson's relevance theory (2004) expanding on Grice's „principle of relevance" argues that the context, including common sense knowledge, clarifies the encoding in vocabulary terms. In actual conversations the participants adjust their terminology rapidly to a small set of terms with meaning fixed for the duration of the conversation and possibly future conversations between the same partners.

A person in a conversation picks up the terms with the meaning used by the other. Many experiments with children have demonstrated well developed abilities of humans to acquire new words and meaning for a reviewer see (Tomasello 2003). In adult conversation one may encounter explicit statements like "Oh, I see, you mean a person if you say PAX", to confirm an agreement on terminology. Teaching consists to a large degree of transferring terminology with intended meaning to the students; conversations with (public) agencies feel often rigid when the partner in the conversation insists on his terminology and is not willing to adapt to mine.

The meaning of the terminology in a conversation is communicated with implicit reference to the context of the conversation. The context of a conversation

is crucial for understanding. The approach followed here is to make the context of the conversation between human and computer explicit in the program and visible for the user and the programmer.

The experience with first Apple Macintosh and later Windows often shows that programs that can be seen and explored, i.e., where the context is visible at all times, can be used without explicit learning and reading of user manuals. Features that are not visible require expensive learning efforts. It is important that the context of a conversation is limited and therewith limiting the interpretation of the words used to refer to objects. The restriction to the context makes it possible that the structures expressed syntactically and semantically in the utterances are mapable despite initial discordances in the meanings.

2.2 What Is the Shared Context of a Human Computer Interaction?

In analogy to natural conversations I see the human and the computer as having a conversation with a shared context. The shared context consists of the objects that can be referenced in the conversation and the operation applicable to change them or to change the relations between them. The interface screen must show the current state of objects and the operations applicable; the now standard "desktop view" with menus demonstrates the success of this rule, but the rigidity of the computer program forces all adaptations on the user. The context must be limited to "real world objects" already meaningful to the user and not forcing her to learn about internal aspects of the program. It should not include aspects internal to the program, but not a reflection of real world facts.

3 The Transparent GUI

A user manipulates a dataset in a computer through an interface. The dataset stands for some computer external reality, perhaps a model of a complex process, but perhaps as mundane as a letter, later printed. This will be termed model. The intention of the user is (Figure 1) to see and change the model she perceives by function p and changes with actions a.

The user is forced to act through the GUI program (Figure 2). Thus the intended functions p and a are:

$p = q \cdot f$

$a = c \cdot b$.

A GUI is well-designed if it gives the user the impression that she manipulates directly the objects of interest. This means the GUI is transparent to the user and the user can concentrate on his work and is not disturbed by artifacts of the GUI.

The user intends action a to change state of the model m0 to state m1

$m1 = o\ m0$.

The user intension

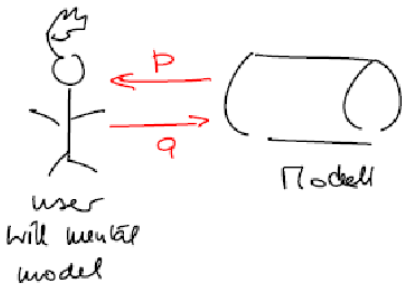

Figure 1: The user's intention is to manipulate a model of some part of reality

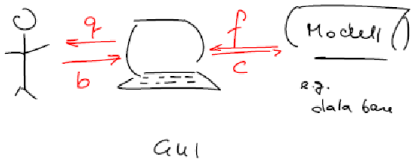

Figure 2: The user is forced to manipulate the model through the GUI

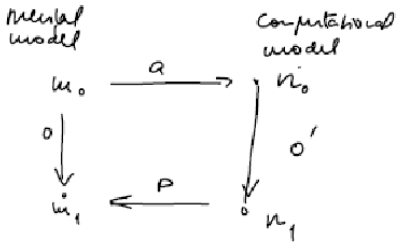

Figure 3:

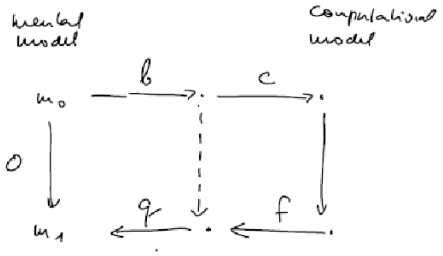

Figure 4:

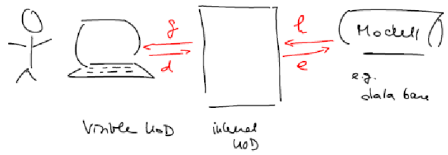

Figure 5: The UoD represents the shared, interpreted context

becomes

This gives the Transparency Condition, which is formally defined as commutativity in Figure 4: A GUI (a = c · b, p = f · q) is transparent if:

m1 = o (mo)
no = (c · b) m
n1 = o' (no)
m1 = (f · q) n1
m1 = (f · q · o · c · b) mo.
or
o = f · q · o · c · b.

3.1 The Internal Representation of Context

The model is usually more extensive than what the present user focuses on in the conversation. The objects and operation currently in focus will be termed Universe of Discourse of the conversation (UoD). This UoD is the part of the model that is in focus during the interaction. The computer makes the context visible to the user on the screen (or other HCI device) and expects user inputs regarding the object and operation from the user (Figure 5).

Comparing Figure 5 with Figure 2 indicates where difficulties in the programming of the interaction occurs: Figure 2 shows the ordinary situation; the program manages the relation between the visible screen and the stored model. Actions of the user must be interpreted to determine which operations to apply to the model; this interpretation may fail. For example with a click on the screen the user intends to select one object he sees on the screen but the program may find other (non-visible ones) in the model. Which one is meant by the user?

Figure 2 shows how the introduction of a UoD breaks the relation f between objects visible to the user and the model in two relations g and h, reducing a complex relation to two simpler functions.

3.2 Analysis of the Function Involved

The function f is the composition of g and h

$f = g \cdot h$

and likewise

$c = e \cdot d$.

3.2.1 Mapping Model UoD

The user selects some subset from the model she is interested in and this subset is mapped to the UoD; this mapping can be for example a SQL statement. The UoD maintains the identification links between entities in the UoD and in the model.

3.2.2 Mapping UoD to Screen

g is the function that translates the stored UoD to a visible screen. It may have additional parameters for style, user preferences, etc., which must be constant for a conversation. g is a function, as the state of UoD determine what is shown on the screen.

g: UoD $\rightarrow$ screen

The screen is seen as a unit with many different states; parts of the screen are not considered individually.

$p = e \cdot f = e \cdot g \cdot h$

$a = d \cdot g = d \cdot k \cdot l$

3.2.3 Mapping Screen to Human Mind

The human flexibility and potential to learn allow this mapping to became a function from screen state to interpreted model.

3.2.4 Mapping Human to Action

The user selects an action o to change the interpreted model. The immediate feedback as this change can be observed will, over time, help her to learn what the actions produce and adapt to the interpretation of objects and operations shown on the screen by the program.

3.2.5 Mapping Action to UoD

This is the most difficult problem: interpreting the actions the user intends and communicates. Action consists of operations and objects operated on. Two forms

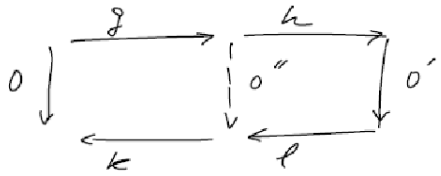

Figure 6: The mappings in a GUI using the UoD approach

for commands are possible first verb (operation) then object (e.g., "open myfile.txt") or first selection of the object and then the operation (e.g., "myfile.txt open"). The selection of operations determines what objects (and how many) must be selected. Alternatively, the selection of an object determines the operations possible. In either caFse the possible selections are limited to the UoD. If the user intends other objects as targets for objects, it is first necessary to bring them into the UoD.

3.2.6 Mapping to UoD Model

The UoD must only offer operations possible to execute in the model.

3.3 Conclusion

The programmer is asked to provide the 4 mappings g, h, k, l , which are listed above; They must be functions to achieve that the GUI appears transparent to the user as she has the illusion of directly manipulating the model visible on the screen.

F

Transparent GUI can be read as a diagram of mappings (Figure 5) and

o = g · h · o · k ·l.

f = g · h

c = e · d

The complex relation c, which is the mapping from screen action to computation is broken into g · h, which can be restricted to functions. The restrictions are communicated to the user by visible feedback. Programming is divide into two parts, namely one of managing the screen (g and k) and one of changing the state of UoD (h and l).

4 Case Study: managing a shared repository of documents

Applications where the user manipulates a coherent and compact model are easy to program, even without an explicit concept of UoD (e.g., a file manager for local files). The application that brings together data from three different

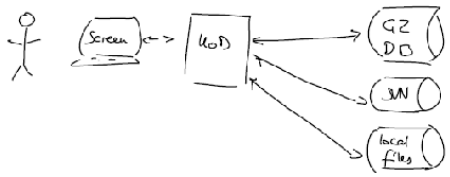

Figure 7: The UoD unifies the data from different sources

sources: A team with each person having a computer, which is not always connected needs to organize a shared repository for documents. There was already a database with a description of dockets and their numbers (GZ) first only used as an entry in the archive of paper files accumulated over years. To extend this for computer files, a version management system (SVN) was installed on a mirrored disk attached to a server and regularly copied to external storage.

In this system information on dockets are in three places: the database with GZ numbers and descriptors, the folders on individual computers and the shared repository. User operations on these three locations must be coordinated and a single GUI was desirable. This means, the user sees a unified model with which she interacts; the UoD shows this model (Figure 7).

The database for docket numbers (GZ) serves to find documents for a variety of keywords and allows also to enter the state of a docket (e.g., paper draft, paper submitted, etc.).

The convention for folder names is that they start with the GZ number followed by whatever name is meaningful for this user (e.g., one user may add the name of the original creator whereas I do not want to have my name in all folder names). The user keeps all GZ folders in a work directory on their personal computer. Folder names (except for the leading GZ number) do not necessarily agree on different computers.

5 Design and CodingF

5.1 UoD

5.1.1 Data

The user focuses at once one docket, for which the information is:

- GZ (docket number)
- Descriptor (descriptive text for the docket content)
- Name of SVN folder
- Name of local folder

The wider context of

- GZ database connection
- Local work directory
- SVN connection

Is shown as "preferences".

5.1.2 Operations

The UoD includes the operations to get information or to change something and get feedback:

get :: gz → info
find :: keywords → info
new gz :: descriptor → info with new gz
change : changed descriptor → info
change-preferences : preferences → preferences

The following operations do not give the user feedback in the UoD, but some confirmation about details of executing the operation.

checkout :: ..
update ::
insert ::
comitt ::

In the module UoD.hs Figure 8definition for operation tokens, button (or menu) text and tooltips are entered together with the data type definitions. The generation of the alphanumeric GUI is then automatic using wxGen. The operations for change-preferences, preferences in a separate window are

Edit preferences
Save/commit
Missing:

- Description of semantics of operations
- List of possible error conditions.

Open questions:

1. How to have more influence on the appearance of the data presented through wxGen?
2. How to deal with graphical presentations?
3. How to have the descriptions of the UoD closer to or merged with a description or semantics?
4. How to enter constraints on data? WxGen offers some facilities that need to be explored.
5. How to build a fully generic GUI to interact with the preference? Currently the data description for preferences must be imported.

Figure 9

5.2 The Static Appearance of the GUI

The GUI has a window with a title and some instructions at the top. Then the fields of the UoD, the buttons for the operations and a field for posting details of operations (Figure 8).

[figure missing]

The definition of variables for the preferences and the UoD are also necessary.

Question:

How much of this code can be generalized and reused? Different applications would only require different text entries, describing the application only.

5.3 The Dynamic GUI

The actions are started by pressing an action button after having filled the corresponding field in the UoD. This is a "data entry then action" paradigm.

Question.

1. Can a "return" in an entry field start an action? How to add this to the generated UoD entry?

2. If a button is pressed and no data entered then show a dialog data entry panel. This would give an "operation then data entry" paradigm.

Processing an action starts with the entry from read and the UoD variable updated. Then a case statement separates the actions to call the pertinent actions on the database, the SVN repository or the work directory. The operations called are specific for the applications and do not contain GUI aspects; they are not of interest here. The operations update the UoD variable if appropriate. After the specific operations have completed, the screen is updated.

In this application, all values functionally depend on the GZ. It is therefore appropriate to retrieve from each of the three sources the particulars and show them in the UoD. The only special case is the search for GZ given a list of keywords. The result from the SQL query is entered into a list and the user selects with double click. Then the UoD is updated with the selected GZ. The list remains visible and another item could be selected to correct the first selection.

Questions:

1. Could a description of the operations as functions with parameters be used to produce the code for actions?

2. Add an error treatment using the moved transformer ErrorT.

5.4 Part_of Relation

Many UoD contain objects at multiple hierarchical level; for example the operations discussed in the case are focused on folders, but the add command takes a

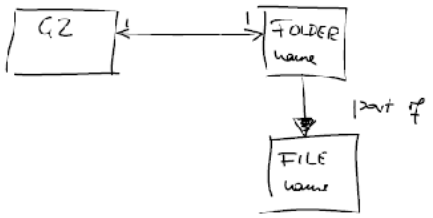

Figure 10: The folder consists of files—a part_of relation

file and adds it to the versioned part of the folder. Part_of Relations are visible in the ontology:

[figure missing]

In order to refer to part_of an object (here called subobjects), which is necessary for operations where an argument is a sub-object; the GUI must provide a tool to select subobjects. This is typically a selection from a list. It is therefore possible to translate part_of relation systematically to the GUI.

5.5 Error Handling

Interactive programs may fail for a variety of reasons and it is important to inform the user about the reason for failing and possible actions for correction. Remember, however, that interfaces where no failures are possible are even better and code that prevents user errors is desirable. Errors are detected at one point in the code and information to the user are produced where the screen is handled.

The simplest error handling approach sufficient for the present purpose is to have the application routines signal failure with a descriptive text and adding a handler for such exceptions at the GUI. At the place the error is detected a call to fail "error string" is included (in monadic "do" code). The handler is simply taking the error string and shows it in the display window.

Open questions:

1. How to deal with DB errors? Is another error data type used?
2. Should exceptions be used to deal with the sub-operations (i.e., selection of sub-objects, in the example files as parts of the folder)?

5.5.1 Reactive Computing

The use of a GUI brings the GUI for an administrative process closer to the examples given for reactive GUI design. Despite the appearance of the GUI similar to a form to enter data, the logic is not one of data entry and the corresponding tools are not appropriate (e.g., wxGeneric). Data entry in forms is a special case of a GUI, but the application considered here is not subsumed under this. The following test shows this. In a "forms" application there is typically a pair of buttons, ok and cancel, which start processing after the data are entered.

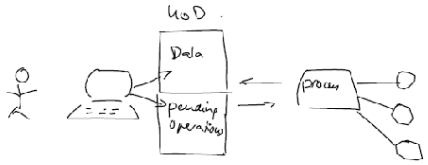

Figure 11:

Entering the user preferences id such a "forms" part in the case study, but the major interface start processing when the user enters a GZ number or a keyword and units return, presses a button of a list.

Conceptually the GUI can be reduced to deposit the changed data and the desired operations in the UoD and the application process is executing the operations and applies updates the UoD. The GUI then propagates the updated UoD data to the screen (Figure 11). Questions:

1. Is this feasible even for the selections from list for aggregates
2. Does this give a structure that leads to generalization?

5.5.2 Conclusion

The introduction of a stored UoD to which all interactions refer has streamlined the code. Most of the code is generic and reusable for a new application. The next test will be with an application using a graphic interface to see whether the same overall structures and code can be used.

References

Evers, S., P. Achten and J. Kuper (2004). A Functional Programming Technique for Forms in Graphical User Interfaces. Implementation and Application of Functional Languages 16th International Workshop, IFL 2004, Lübeck, Germany, Springer.

Grice, P. (1989). Studies in the Way of Words. Cambridge, Mass., Harvard University Press.

Hallgren, T. and M. Carlsson (1995). Programming with Fudgets. Advanced Functional Programming. J. Jeuring and E. Meijer. Berlin, Springer-Verlag. Lecture Notes in Computer Science 925: 137-182.

Shneiderman, B. (1997). Designing the User Interface - Strategies for Effective Human-Computer Interaction. Reading, MA, Addison-Wesley.

Sperber, D. and D. Wilson (2004). Relevance Theory. Handbook of Pragmatics. G. Ward and L. Horn. Oxford, Blackwell: 607-632.

Tomasello, M. (2003). Constructing a Language: A Usage-Based Theory of Language Acquisition. Cambridge, Massachusetts, and London, England, Harvard University Press.