

THINKING MODULAR TOWARDS A PLUGGABLE ATLAS USER INTERFACE

Stefan Huber, Patrik Jeller, Marianne Ruegsegger

Institute of Cartography
Swiss Federal Institute of Technology (ETH)
ETH Hoenggerberg
CH-8093 Zurich, Switzerland
e-mail: {huber, jeller, ruegsegger}@karto.baug.ethz.ch

ABSTRACT

The number of available functions and tools in interactive high standard atlases is steadily growing. The needs of the users vary; some functions are of interest only to a small group of users and might be confusing for others. As a consequence the complexity of the graphical user interface increases and many functions are hard to find and therefore remain unused. Thus the demands are not only increasing on users, but also on developers. To handle this large number of atlas functions and to keep the software maintainable and extensible, a new architecture for atlases has to be found.

Hence the concept of modularization of atlases is introduced and will be illustrated by the example of the newly released “Atlas of Switzerland 2”. The four basic characteristics of modularity, encapsulation, replaceability, composeability and extensibility are discussed. The paper shows how modularization can be achieved in an interactive atlas environment exemplified by the graphical user interface.

It was found that a user interface is best split into interface elements, interface modules and shell containers. These components communicate through the interface manager within the core of the application by loose coupling. It is shown how this aim can be achieved and that it is possible to create a bipolar, modular architecture for interactive atlases. The interface part is hierarchical (constrained flexibility, but easy-to-use); the required flexibility has been effectually moved to the application core (interface managers). Flexible modularization is a step towards the goal of the “pluggable atlas” that allows the user to customize it to his/her specific needs and helps developers to expand and upgrade the functionality easily.

1 INTRODUCTION

The demands on high standard interactive atlases, like the “Atlas of Switzerland”, are steadily growing. Many users are familiar with Geographical Information Systems (GIS), interactive maps on the World Wide Web and 3D real time navigation. These applications offer a wide variety of functions and tools to analyze and visualize the data at hand, for example tools to measure distances or virtual flights. Acquainted with these possibilities, users expect to have the same functions and tools available in an interactive atlas. Thus, to satisfy the needs of this advanced user group, a complex atlas with as many functions as possible should be developed. The resulting longer initial learning time to get accustomed to all functionalities would be acceptable to this group. Another important user group of interactive atlases, especially national atlases, are students and interested laymen. Typically this group is not used to interpret and analyze geographical data and often even their computer literacy is limited. For these users the atlas design has to be kept as simple as possible. The developers are thus confronted with the problem to design an atlas that meets the demands of both user groups at the same time. As a consequence, functionalities have not just to be well structured internally, moreover they need to be presented in a clear and intuitive way to the user. Hence, an optimal software-architecture and an appropriate user interface has to be found.

It is obvious that a software architecture with mainly interdependent components is hard to understand, difficult to maintain and troublesome to update. Consequently, the software elements should be largely independent from, but communicating with each other. To meet this requirement, the concept of a modular atlas is presented in this paper, considering the “Atlas of Switzerland” as example. The architecture of the present version, “Atlas of Switzerland 2” [2004], is partly modular. The 3D-part of the atlas consists of largely independent sections (query, navigation, and visualization) that are linked to the rendering engine. These again are subdivided into independent modules.

2 MODULARITY AND USER INTERFACES

The concept of modularity is not just known in computer science but also in other sciences, e.g., engineering and management. It is therefore interesting to find that the basic definitions are the same in all disciplines. Modularity can be defined as a way of organizing complex products and processes efficiently [Baldwin & Clark 1997] by splitting complex tasks into simpler ones that can be handled independently [Mikkola & Gassmann 2003]. The resulting parts can be produced and handled separately and it is possible to use them in different environments, either in the same product or even in another project (*reusability*). These parts are called modules or components. Each module contains a set of related data structures and functions. Modern programming languages, like C++, support this modular approach [Litvin & Litvin 1998].

The benefit of modular software design includes a clearer structure, more efficient implementation and upgrades, easier software maintenance and reusability. Each module in a project carries out a certain task. Software modules can be compiled separately and linked into one program. Once the modules are linked with each other through so called *interfaces*, they can be implemented and modified independently [Litvin & Litvin 1998].

Standard components have well defined, standardized interfaces and are therefore simple to implement and replace. New modules, on the other hand, need new interfaces as well and therefore are more time-consuming and cost-intensive; in addition they are more error-prone. Hence, new modules should only be introduced after careful consideration and extensive testing. In addition, Hatton [1996] claimed in his controversial paper that smaller components tend to have more bugs than bigger ones. In the following he [Hatton 1997] and many others tried to find the optimal size of modules. According to El Emam et al. [2002] there is a simple continuous relationship between class size and faults and that small class sizes seem to be better than bigger ones.

Modularity in an atlas environment

It is the purpose of this paper to show how modularity can be understood and implemented in an interactive atlas. Here the use of modularity is proposed in a wide sense. The whole architecture of an atlas can be understood as modular. The "Atlas of Switzerland 2" comprises a 2D- and a 3D-part, both of them can be regarded as a module. They again can be subdivided into smaller components, e.g., the user interface, the visualization module and the analysis module. These sub-modules are split again into smaller modules, and so on. Some of these sub-modules are exclusively used for a specific task and others are shared by several modules (e.g., the graphical user interface). This paper will exemplify the concept of modularity by means of the graphical user interface GUI.

User interfaces

A user interface in the context of atlases is the accumulation of means by which users interact with a software program. The most common types of user interfaces are command-line interfaces (input is provided by a string typed on a keyboard), file based interfaces (input comes from text within a file) and graphical user interfaces (user manipulates widgets, text, etc.).

The task of a user interface is to provide a way for the user to communicate with a computer program. Users should be able to complete relevant tasks with ease. The user interface should not only be usable, it should be useful. At the same time it is important that the user is not restricted in his action and the user interface supports as many features as necessary.

The design of a GUI can be a very complex task for several reasons. The more features supported by a GUI, the more its complexity increases. At the same time, a software program is used by a variety of users (laymen, advanced users, etc.). The needs on a GUI differ from user to user (especially in an atlas environment). More advanced users need more features than others, and less experienced users might be scared of by the amount of possibilities provided by the GUI. For developers on the other hand it is important that the software is well structured, extensible and easily maintainable. The use of modularity within the design of GUI can be an approach to solve such conflicts.

Encapsulation (information hiding) minimizes the interdependency of modules. The three main criteria of modularity are replaceability, composeability and extensibility.

Encapsulation

Encapsulation can be reached by grouping the equal and separating the different elements of the GUI into individual types of modules and by creating well-defined interfaces for each type of module. Every module hides its information and communicates through a well-designed interface shared by equal types of modules.

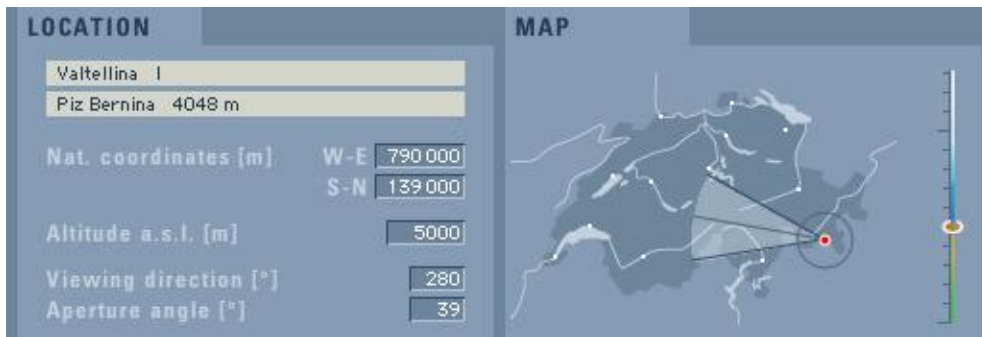


Fig. 1 “Atlas of Switzerland 2”, panorama navigation module a) text-based, b) map-based

Replaceability

Every module can be replaced by a module of the same type. In an atlas environment one could think of two different navigation modules, where the input could be text based (fig. 1a), or graphical (fig. 1b), but each could replace the other without any side effects whatsoever. Even the GUI itself could be seen as a module and thus can be replaced by a completely different implementation.

Composeability

The variety of users is broad and their intuitive approach on a GUI implementation is different. Therefore it often meets the demands of some users, but fails to meet the demands of many others. All too often a GUI is designed on the least common denominator of all users. The concept of modularity brings the advantage of customizable user interfaces. Because the GUI is composed of single modules, independent from each other, it can be assembled in any way. For a less experienced user a GUI with only the essential tools could be provided, while for experts more features could be made available (“GUI-Lite” vs. “GUI-Expert”). Users can even customize the user interface on their own by removing, replacing or adding modules.

Extensibility

A key feature that comes along with modular software architecture is the possibility of plug-in modules. Many times after the release of an interactive atlas the question of extensibility is raised by users. Can they include their own data, or can they visualize the data in a manner not yet supported by the software? Equally for software developers it is a goal to add functionality to a software program without changing the existing program code.

3 GRAPHICAL USER INTERFACE

The “Atlas of Switzerland 2” (AoS 2) has been released in November 2004. Currently, first drafts of “Atlas of Switzerland 3” (AoS 3) are discussed and prototypes of new features will be developed. At this stage the “Atlas of Switzerland 2” [Huber & Schmid 2003] will be reviewed in order to deal with (potential) new features and concepts. In the AoS environment three types of interfaces will be distinguished: graphical (visual) user interface (GUI), key command interface, and control file interface. As the GUI is the most important part of the AoS user interface, the component types will be introduced in terms of the GUI. The following discussion of the user interface will mainly be illustrated by the AoS 2 3D user interface (fig. 2).

The basic component of the AoS user interface will be called the *interface element* (interaction control). It is composed of both interactive and non-interactive elements. Interface elements are compounds consisting of primitives [Cooper 2003]; i.e. smallest indivisible action (click, drag, keystroke) and feedback mechanisms (cursor, text) will be assembled to create generic interface elements. Interface elements have no knowledge of the task under consideration: they are basically not application specific idioms. Interface elements are able to handle input and/or output. Generic input actions are discrete and typed (key stroke, mouse click, etc.). Typical interface elements are buttons, check boxes, text fields, radio buttons, etc. However in the AoS 2 there are several more complex interface elements like the panorama map (fig. 1b, a camera interaction control and state tool). Even this tool is a compound. It handles three bounded input and output variables (position, aperture angle, and direction), but these variables cannot be handled simultaneously. The panorama 3D map itself is a simple pane with a very complex user interaction mechanism. It is a query and a navigation tool based on different navigation modes. Queries will be handled by mouse point events, navigation by mode dependent mouse click and mouse click-and-drag events. Therefore, the 3D map can also be understood as a compound. Moreover the 2D-part contains dynamically created interface elements like the legend and the interactive histogram of the analysis tool. These elements are compounds, too.

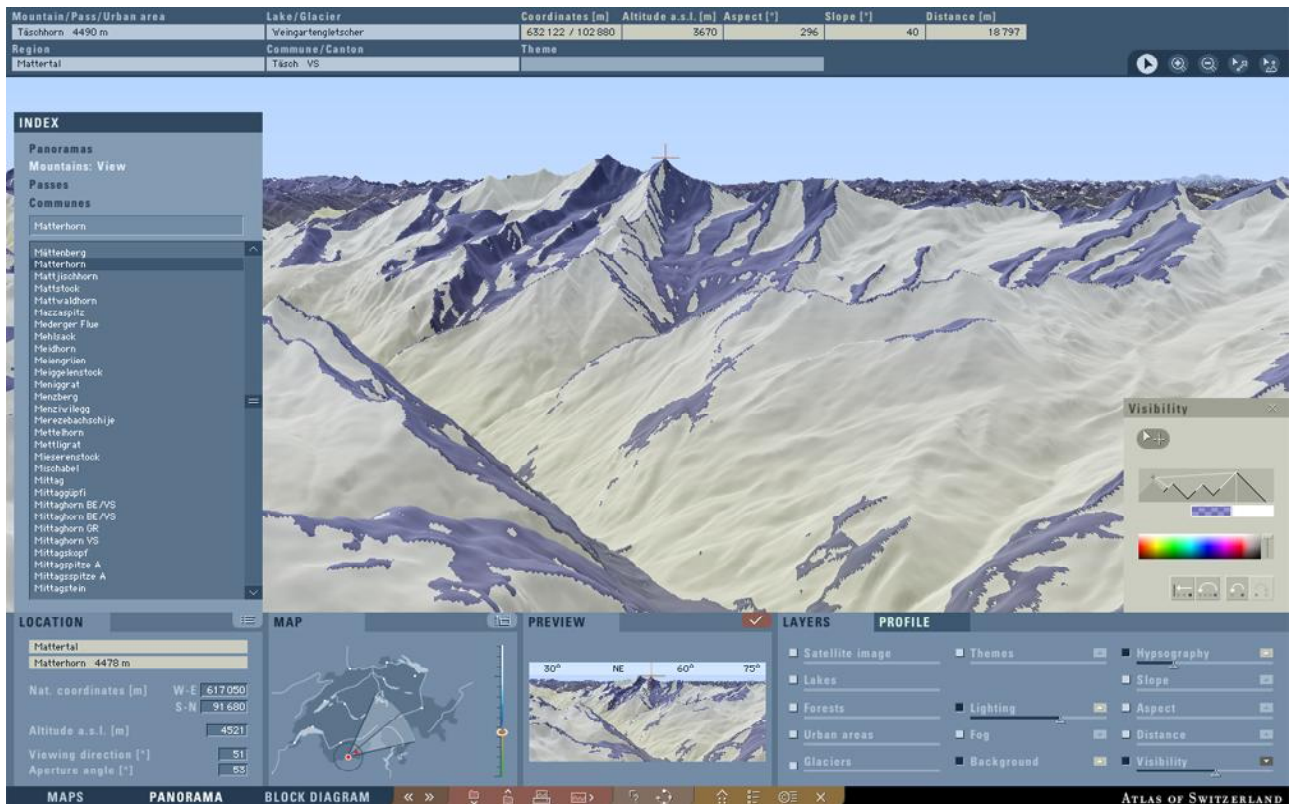


Fig. 2 “Atlas of Switzerland 2”, 3D-part (panorama).

The next step, organizing the user interface modularly, requires applicable criteria. Looking at the AoS 2 3D interface (fig. 2), a user-centered approach could distinguish the “location”, “map”, “preview”, “layers”, and “profile” panels; these panels are in fact clearly arranged. The visualization (“layers”) section for example, is made up of various sub-modules, e.g., slope, aspect, distance, visibility, hypsography and thematic overlays. But on one hand, the “location” and “map” panels are more or less replaceable (“location” is the text-based and “map” is the map-based interface panel of the camera), on the other hand the “layers” panel consists of 14 similar unities that will be treated on an equal basis; the “layers” panel is basically extensible. Moreover the “index” panel consists of four similar lists, and also affects the camera. The criteria to define a unity that aggregates interface elements are replaceability, composability, and extensibility. Therefore it will be useful to treat the “location”, “map”, and “preview” panels as an *interface module*. Additionally, each of the 14 unities of the “layers” panels is an interface module. Some of these modules include an extra panel like the “visibility” panel (fig. 2). Finally, the “index” panel consists of four mutual excluding interface modules. A good example of a composed tool is the “smart legend” draft for AoS 3 [Sieber et al. 2005]. Dynamically created interface elements like the legend and the highly interactive histogram of the analysis tool (fig. 3) are inevitably interface modules, too. They satisfy the criteria of replaceability.

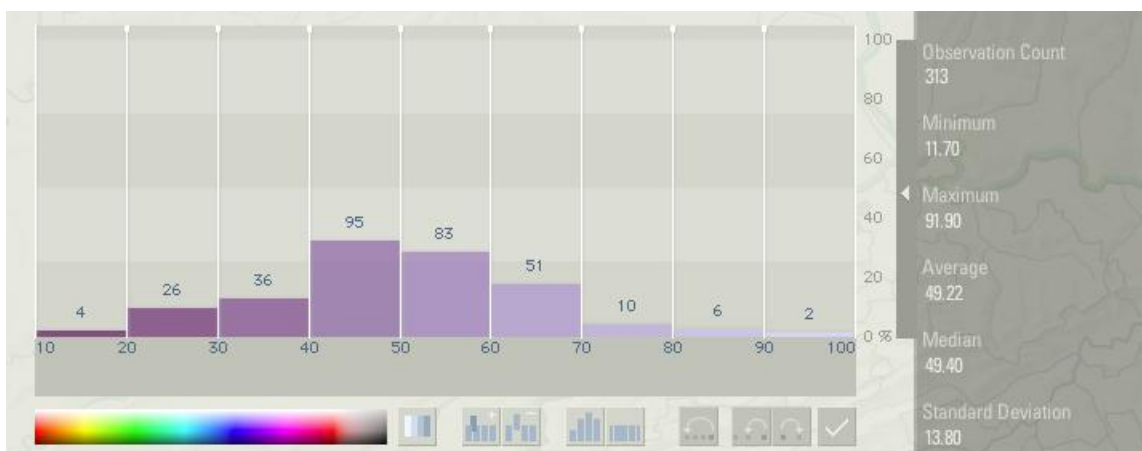


Fig. 3 “Atlas of Switzerland 2”, 2D-part, analysis tool

Actually the interface module does not end in itself but fulfills the three criteria replaceability, composability, and extensibility. In addition the AoS 2 interface module fulfills two supplementary criteria; first it is not generic, but application specific, second its contents are thematically related. These two criteria are important to implement the communication model as discussed below.

A GUI can be conceptually split into the *interface shell* and the *interface contents*. The interface module term explained above represents the contents part. The shell part deals with panel arrangement and navigation. Basically the shell part works independently. Panels can be opened or toggled without knowledge of the contents. In reality shell and contents in a complex GUI cannot be completely separated. The aim is to lay down the duties and responsibilities of each part in order to separate both parts as much as possible.

Given the contents the interface shell has to arrange the parts according to screen size. Panels can be always visible or temporary visible, they may be moveable and they may be scalable. The shell has to manage the different panel states. Scaling panels can be accomplished by user interaction or automatically (e.g., screen size). Moving panels is usually an interactive feature as well as switching to temporarily visible panels. User interaction requires interface elements like buttons, tab controls, lists, pop-up menus, etc. *Shell interface elements* are mere navigation elements. Because screen space is limited the shell has to handle mutual exclusion of panels or intelligent arrangement of temporary visible panels. Among these tasks only scaling really affects the contents part; e.g., the rendering engine requires the 3D map panel size. As an optimization it makes sense that the application knows if a panel is visible or not. In both cases only “one-way traffic” is required. The shell part informs the contents part. Given the contents the other tasks can be handled by the interface shell itself. The shell part will be assembled with hierarchically nested *shell containers*. A container comprises other containers and/or interface modules and/or shell interface elements. To handle mutual exclusion or arrangement of panels, the concept of hierarchically nested containers is not sufficient. The panels concerned register themselves in a singleton list that is responsible for these tasks.

So far shell and contents are only weakly linked. But in reality the contents will be replaced or altered. Panorama and block diagram have different navigation tools, 2D-part tools like “analysis” and “legend” depend on levels of measurement (nominal, ordinal, numerical) and graphical data type (raster, vector). In addition to the two cases that require communication between shell and contents (panel scaling and visibility), the responsible shell containers will be initialized at start-up time (screen size, user preferences, available extensions, etc.), and report the situation at the end to write to the preferences file. Despite these four cases, both parts are still relatively weak linked. As the interface modules on the contents side are rather static (a set of interface elements for a given task), they have to be replaced or modified if the conditions change. Panorama and block diagram have its own text and map based navigation interface module, “analysis” and “legend” interface modules may be newly created when a new theme has been selected. So the shell containers frequently receive messages to modify, load, remove, and arrange interface modules from the responsible manager in the application core. Affected shell containers are able to receive messages from the responsible manager and send messages to interface modules that are part of the container. Nevertheless the message types remain limited: modify, load, remove, arrange, scale interface modules (from the responsible manager), and scaling and visibility state (to the interface module).

4 FLEXIBLE MODULARIZATION

Up to now, three hierarchically dependent interface component types have been defined: interface elements (the basic component), interface module (the contents component), and shell container (the shell component). All component types encapsulate their contents and provide a well-defined interface. But who is in communication with whom? And how? As mentioned above, a fourth but indirectly GUI related component has to be introduced: the *interface manager*.

The AoS 2 application core contains a lot of manager classes (interface manager) that serve as the contact points between the interface and the application core. Typical manager classes are camera, and currently more than a dozen visualization classes like lighting, hypsography, aspect, orientation, distance, etc. [Huber et al. 2003]. The *interface module* mainly communicates with the corresponding manager class. One manager class can be connected to one or more interface modules (replaceability of heterogeneous interface modules). As mentioned above, shell containers receive messages from interface managers, too. In the following discussion the manager classes will be treated as a black box. Interface elements only communicate with interface modules and as shell interface elements with shell containers.

Interface elements are able to autonomously handle generic visual feedback of user interaction, e.g., highlighting on mouse click, slider mechanics, etc; this function is encapsulated. The resulting value of a user interaction is discrete. Some interface elements like the map navigation tool provide intermediate results, too. While this kind of user

interaction is continuous, the results are always discrete. Interface elements have to catch the results and to send a message to the responsible interface module. The message is simply structured: interface element id, value type, and value (e.g., integer value, current slider position); more complex interface elements additionally add to the message the interaction type (mouse point, mouse click, mouse click-and-drag, with or without modifier key(s), etc.). On the other side, interface elements receive messages from the responsible interface module in order to display the new state (interface element id, value type, and value). In certain cases the value type may be not trivial (e.g., panorama map navigation module: a structure of a point and two integer values). Shell interface elements are treated similarly; they send messages to and receive messages from the shell container.

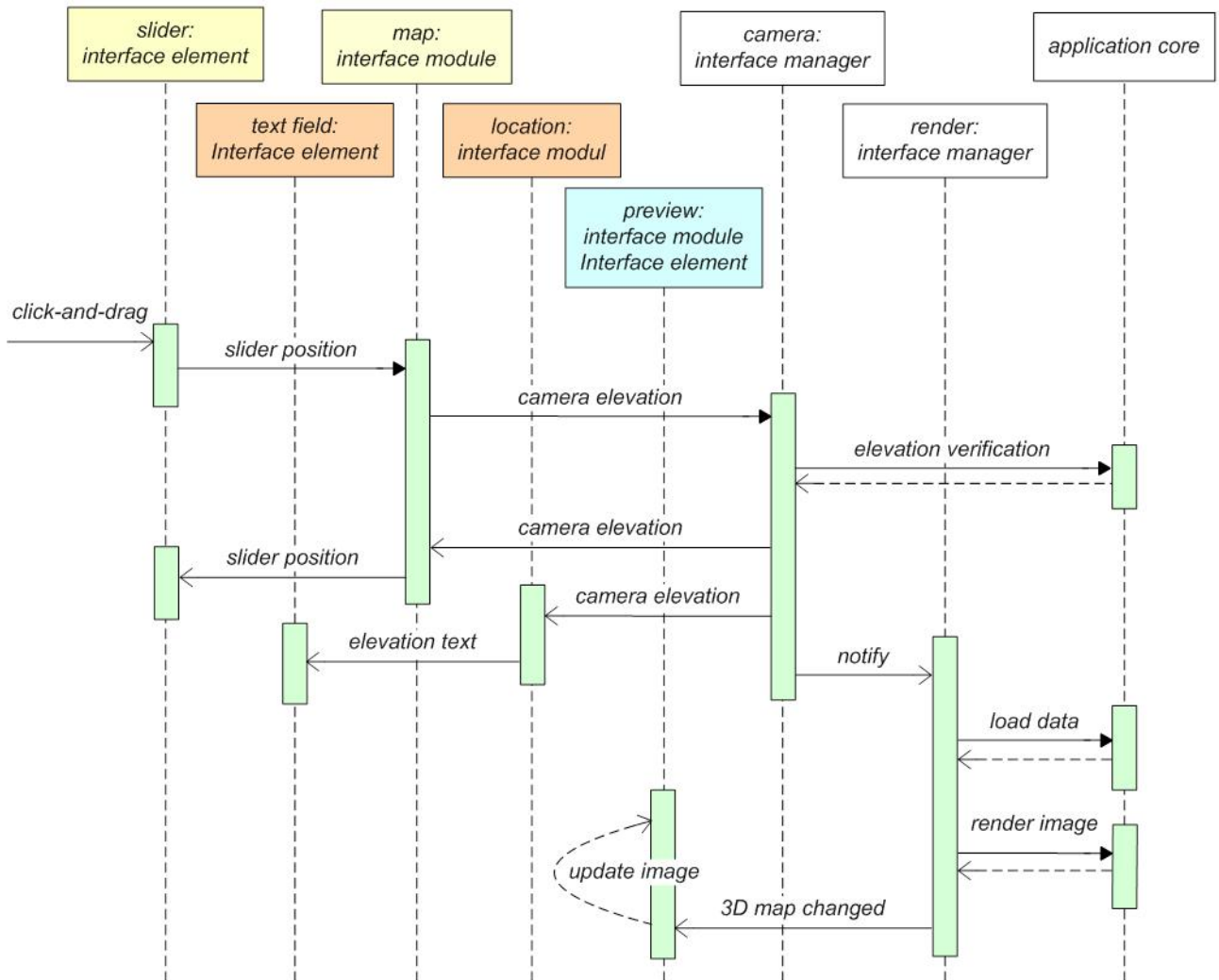
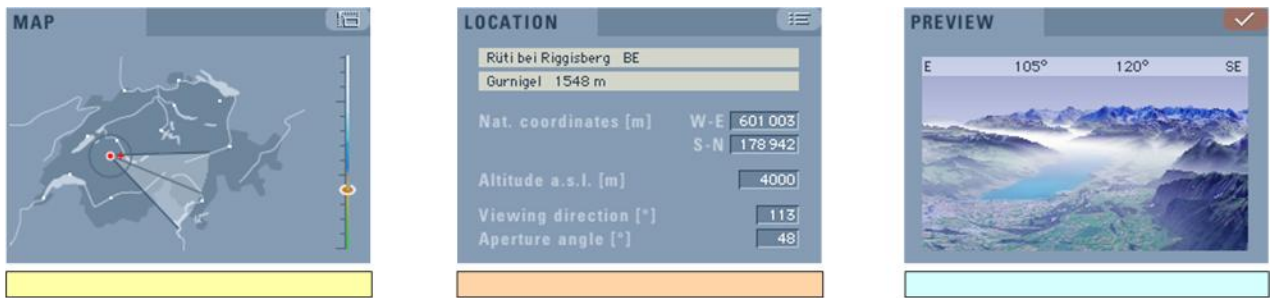


Fig. 4 Sequence diagram of an user interaction

The interface module has to fulfill several tasks. First, it dynamically generates interface elements and/or makes static interface elements available. Secondly, it receives generic messages from the interface elements. The interface module knows the semantic type of each interface element. It translates the generic value into a semantic value; e.g., the check box (interface element) of a visualization module like slope sends an “I have been clicked” message to the interface module. The module itself translates the message into a “turn the slope module on” message and sends the translated message to the corresponding interface manager. In other words, the interface module is the relay between interface elements and the application core – one level of indirection to encapsulate various homogeneous and heterogeneous interface elements. Third, the other way round, the interface module receives messages from the interface manager and sends the translated messages to the interface elements. Fourth, it receives scale and visibility messages from the wrapping shell container and send them to both the interface elements affected as well as to the interface manager. As mentioned before, the shell container receives a few messages types (modify, load, remove, arrange, scale interface modules) from the corresponding interface manager.

An example scenario (fig. 4) illustrates a simple user interaction: The user want to change the elevation of the panorama camera using the elevation slider (interface element) of the map interface module. The slider position will be translated into camera elevation by the interface module. The camera interface manager has to verify the value, because the camera has to be over ground. The possibly corrected value will sent back to the slider and the text field of the location interface module. At last the preview 3D map will be rendered and updated.

The GUI should be separated as much as possible from the application core; loose coupling between interface manager and interface module/shell container is essential. A message from an interface element will be sent via interface module to one interface manager only. The other way round a message from an interface manager will be sent to one or more interface modules (e.g., camera manager to all modules that reflect the camera state). Messages may have different priority. Behavioral patterns [Gamma et al. 1995] like command pattern, state pattern, and observer pattern help decoupling interface and application core. The GUI of the AoS is based on nearly exclusively hierarchical components. Hierarchical modularization is easy to use but unfortunately it lacks flexibility. The required flexibility has been effectually moved to the interface managers. The domain of the interface managers can be characterised by using a rhizomic (the term rhizome has been established by Deleuze & Guattari [1976]) modular concept where a module (e.g., a class) from one branch can be connected directly to a module from another branch.

5 CONCLUSIONS AND OUTLOOK

Modularization is a popular term, especially in the area of software engineering. It is obvious that design and production of a software system includes many processes (and often many people, too), and must be divided into separate parts. For this reason the question is not if modularization should be implemented, but to which degree and how it should be done effectively.

The GUI modularization concept is based upon three criteria of modularity: replaceability, composability, and extensibility. The resulting interface modules are application specific and thematically related. In addition the GUI will be conceptually split into interface shell and interface contents. There is a gap between the user-centered and the AoS approach. Interface parts on the same user-centered level (fig. 2: “location”, “map”, “layers”, etc.) may be assembled very differently: as interface module or as hierarchically nested shell containers. Using the criteria of modularity to define interface modules, a trend has been found: the more complex the interface, the smaller the interface modules and the larger the gap between the user-centered and the AoS approach.

Cartographic research in the “Atlas of Switzerland” project is an ongoing process that results in new features and maps. The final GUI of AoS 3 will be realized in a rather late phase of the production cycle. The AoS 2 GUI has been partly developed with an authoring system, other parts (complex interface elements) have been written in C++. Merging 2D- and 3D-part to a large extent is one of the primary goals of AoS 3 [Huber & Sieber 2001]. Thus an easy-to-use flexible GUI modularization concept is crucial. A bipolar modular architecture has been developed. The interface part is hierarchical structured (constrained flexibility, but easy-to-use); the required flexibility has been effectually moved to the application.

The techniques to decoupling interface and application core as well as the rhizomic modular concept behind the interface has been mentioned. A thorough discussion of the subject has to be done in the future.

REFERENCES

- Atlas of Switzerland 2* (2004) [DVD / 2 CD-ROM]. Swiss Federal Office of Topography, Wabern, Switzerland.
<http://www.swisstopo.ch>
- C. Y. Baldwin and K. B. Clark (1997): *Managing in an age of modularity*. Harvard Business Review, Vol. 75, No. 5. 1997.
- A. Cooper (2003): *About face 2.0: the essentials of interaction design*. Wiley, Indianapolis.
- G. Deleuze and F. Guattari (1976): *Rhizome*. Minuit, Paris.
- K. El Emam, S. Benlarbi, N. Goel, W. Melo, H. Lounis, S.N. Rai (2002): The Optimal Class Size for Object-Oriented Software. *IEEE Transactions on Software Engineering*, Vol. 28, No. 5, pp. 494-501, May 2002.
- E. Gamma, R. Helm, R. Johnson and J. Vlissides (1995): *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, Reading MA.
- L. Hatton (1996): Is modularization always a good idea? *Information and Software Technology*, Vol. 38, pp. 719-721, 1996.
- L. Hatton (1997): *Reexamining the Fault Density – Component Size Connection*. *IEEE Software*, Vol. 14, No. 2, pp. 89-96, 1997.
- S. Huber and C. Schmid (2003): 2nd Atlas Of Switzerland – interactive: Concepts, Functionality, Techniques. *Proc. of the 21th Int. Conference of the ICA*, Durban, South Africa.
http://www.atlasofswitzerland.ch/pdf_publi/ICC03_HuberSchmid.pdf
- S. Huber, R. Sieber and A. Wipf (2003): Multimedia in der Gebirgskartographie - 3D-Anwendungen aus dem «Atlas der Schweiz - interaktiv 2» *Kartographische Nachrichten (KN)* No. 5, 2003.
http://www.atlasofswitzerland.ch/pdf_publi/KN03_HuberSieberWipf.pdf
- S. Huber and R. Sieber (2001): From Flatland to Spaceland – Concepts for Interactive 3D Navigation in High Standard Atlases. *Proc. of the 20th Int. Conference of the ICA*, Beijing, China.
http://www.atlasofswitzerland.ch/pdf_publi/ICC01_HuberSieber.pdf
- M. Litvin and G. Litvin (1998): *C++ for You++*. Skylight Publishing, Andover MA.
- J. H. Mikkola and O. Gassmann (2003): Managing Modularity of Product Architectures: Toward an Integrated Theory. *IEEE Transactions on Engineering Management*, Vol. 50, No. 2, May 2003.
- R. Sieber, C. Schmid and S. Wiesmann (2005): Smart Legend – Smart Atlas! *Proc. of the 22th Int. Conference of the ICA*, A Coruña, Spain.