

ACCOMMODATING USABILITY DRIVEN CHANGES IN EXISTING SOFTWARE ARCHITECTURE

Tamer Rafla, Rafiou Oketokoun, Artur Wiklik, Michel Desmarais and Pierre-N Robillard
Software Engineering Research Lab (RGL), Department of Computer Engineering,
Ecole Polytechnique de Montreal, P.O. BOX 6079, Station Centre-Ville,
Montreal, H3C 3A7, Canada
{tamer.rafla, rafiou.oketokoun, artur.wiklik, michel.desmarais, pierre-n.robillard}@polymtl.ca

Abstract

Issues such as whether a product is easy to learn, to use, and whether the user can efficiently complete tasks using it, may greatly affect a product's acceptance in the marketplace. In software engineering, the support for usability is widely believed to be independent of software architecture design. This belief stems from the efforts to separate the user interface component from the application's internal logic, thus enabling changes to the interface without affecting the software architecture. This assumption has been recently challenged by Bass et al. [6] who argue that architectural patterns must be in place in order to support good usability design. We investigate the degree to which this revised belief is true with a case study on the redesign of an existing application for better supporting usability. The redesign is based on implementing a task-oriented interface and help system. Preliminary results show that much, though not all, of the required changes can be done without major changes to the software architecture of GIMP. The results do support the idea that the envisioned usability redesign could not easily be implemented without some powerful architectural features that do not necessarily correspond to the specific patterns identified by Bass et al.

Keywords: usability scenarios, software architecture, interface redesign

1. Introduction

As users become aware of good interface design, poor usability may become a major barrier to the success of new commercial software applications [1,2]. However, usability is not usually considered in the software architecture design, due to the widespread assumption amongst software designers that usability has only to do with the visible part of the interface. This assumption is now challenged since usability is strongly related to the internal components of the system and it must be considered when designing the rest of the system [3]. Decisions in software architecture can severely compromise the usability of the final system. For

example, if a "help mechanism" is not devised when the software architecture is established, it will be very costly to incorporate afterwards. Therefore, it is important to relate system's usability with decisions taken in the early design of the software architecture [4]. Failure to do so can result in expensive changes late in the development process. Our approach aims to use usability scenarios for the redesign of a task oriented user interface and to assess their impact on the architecture of GIMP.

GIMP, the software concerned by this study, is a freely distributed software for such tasks as photo retouching, image composition and image authoring. It works on many operating systems, and is available in many languages. Many people find it extremely useful in creating logos and other graphics for web pages. The GIMP has many of the tools and filters you would expect to find in similar commercial products and some interesting extras as well.

The following section highlights in detail our methodology of research and discusses the usability redesign scenarios we are taking into consideration. The following three sections, will discuss how these scenarios affect the architecture of GIMP. The last section summarises our results and presents the benefit of choosing specific usability scenarios to attain a greater level of usability.

2. Usability scenarios

Usability scenarios have been used both in analyzing [5] and designing software architectures [6]. The first step in investigating the relationship between usability and software architecture was to generate scenarios that expressed a general usability issue and seemed to have architectural implications. For example, a common usability scenario is that a user changes his or her mind about issuing a command and wants to cancel that command before it has been completed. This is generally applicable to many software systems and has architectural implications because the system must allow a cancel command and should be able to restore its previous state.

Bass [6] identified 26 specific usability scenarios, where a scenario describes an interaction that some stakeholder (e.g. end user, developer, system administrator) has with the system under consideration and which has software architectural implications. For each of these scenarios, Bass identified an architectural pattern that enables the achievement of that scenario.

Of these 26 scenarios, we identified three of them which apply to the redesign of GIMP:

- **Modifying interfaces:** The GIMP interface suffers from an overload of functions simultaneously available, for a wide range of tasks, from logo creation to photo retouching, only a subset of which are relevant for a given task. The heart of the redesign requires a modification of the interface to make available only the relevant functionality for a given task. This is described further. For now, let us emphasize that it is a case of the “modifying interfaces” scenario, which essentially states that system designers should ensure that their user interfaces can be easily modified.
- **Aggregating commands:** A user may want to complete a long-running, multi-step procedure consisting of several commands. For example, in GIMP, you have to manually set the width and height of the picture you wish to create in different interfaces. It becomes tedious to invoke these commands one at a time, and to provide parameters for each command it executes. If the application is unable to accept the required inputs for this procedure up front, the user will be forced to wait for each input to be requested separately. Aggregating commands is thus a second scenario that applies to our context.
- **Providing good help:** Help procedures should be context dependent and assist users in solving problems. Considering that the redesign of the GIMP interface aims to provide a task-oriented interface, it appears evident that a help system should take the task-context into consideration. The redesign of the GIMP interface thus integrates a task aware help system.

The following section explains how we have developed design solutions for the above scenarios.

3. Satisfying the scenarios

a. GIMP’s architecture

With little documentation on GIMP’s architecture, we looked at the program using an object- oriented approach:

the system was divided into modules combining related tasks and functions. The windowing structure of the program, with its tools and features, led us to a top view of GIMP’s conceptual architecture, as shown in Figure 1. The main window becomes the Toolbar, containing all the image manipulation options. It acts as a parent to open Image Windows. Two types of plug-ins are available to the system. The manipulation effects plug-in, connected directly into the Toolbar window, is used by any Image Window, while allowing developers to add new functionalities (new tools or effects to the program). The second type is an Image format implementation plug-in. Directly connected to the Image Windows, it adds new file format capabilities. The principal difference between these two plug-ins is that the latter depends on an existing image and is especially reserved for basic image format implementation (examples: filter, save, load, etc).

Many plug-ins have been developed to work with GIMP, each encompassing all aspects of its particular effect or tool. These plug-ins connect directly to the Toolbar, allowing all Image Windows access to the new functionality.

The GIMP’s interface modification scenario in our study has been realized through the first type of plug-in.

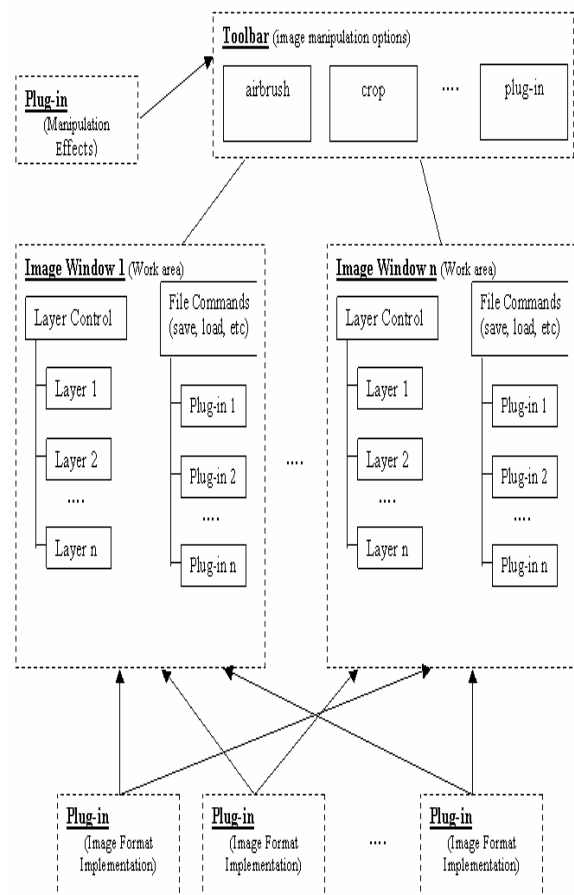


Figure 1. GIMP's conceptual architecture

b. Modifying the interface

The actual GIMP user interface suffers from an overload of functionality. The interface is extremely hard to learn and use for most people. The user is exposed to a large amount of different functionalities which are not organized by specific tasks, but instead are simultaneously available for *any* task. We have started redesigning GIMP's interface by regrouping functionalities around high-level tasks. An example of a high-level task is the removal of the red eye effect in a picture, or the creation of logos.

Can these changes be embedded without affecting the architecture? The answer is yes, the new interface components can be added quickly to the existing ones as plug-ins using the GimpTool utility.

Modifying GIMP's interface does not affect the software architecture. GIMP is a perfect example of software which has the integrated means to facilitate the interface modification without affecting its sophisticated architecture. When developers decide to update or change the existing software interface, they have to perform some minor modifications to the software architecture; existing modules have to be reworked to support the changes to the software's interface. Before adding new components to the interface, designers have to understand the software architecture and only after doing so, will they be able to change it. In order to modify the interface, we have at our disposal, tools, like GimpTool, PDB (Procedural DataBase) and libgimp. GimpTool helps install and uninstall the plug-ins for Gimp while PDB is a database that contains all functionalities, plug-ins and scripts, related to image processing. The libgimp library provides some functions to call subroutines from the PDB, or enter new functions into the PDB. Using all those tools, developers can easily create his own plug-in and integrate it to GIMP and use libgimp library to call the functions stored in the PDB. (A good deal of the GIMP's functionality is invested in external modules, called *plug-ins*.) These independent programs are called from GIMP to run as separate processes and carry out essential tasks such as loading and saving images, running effects filters, and interfacing other programs with GIMP (such as scanning software, ray tracers, other image manipulation packages, etc). Once the plug-in is created, developers can now install it using the GimpTool. Also, developers have also the choice to link the plug-in with the principal GIMP's interface. A menu item can be added for this plug-in into the main menu. Once the plug-in is installed, the user can now select the newly created menu item to open the new interface and start interacting with it.

c. Aggregating commands

Often, users interacting with the software need to perform the same task or set of tasks over and over again, which can become very frustrating for them. The image processing software's tasks are composed of a multitude of subtasks. To accomplish each subtask a user has to enter many input parameters, which can be very time consuming, especially when it has to be done repetitively. To avoid that kind of situation, some softwares use the principle of aggregating commands or macro capability. Macro capability allows the user to perform multi-step procedures many times. Users are able to group many tasks into one big task and run them all at once. They can then repeat this process as many times as needed. GIMP is a perfect example of macro capability with ScriptFu. Users can create a script file and include in it all the necessary subtasks to accomplish one specific task. The script file can be rerun several times and parameters can be changed in order to obtain the desired results. ScriptFu is a powerful tool because the amount of time required to accomplish one complex task repetitively is reduced. This allows the users to focus only on the parameters to enter and the sequence of the tasks to accomplish.

It is possible to create ScriptFu scripts without having to add new functionalities and modifying the existing architecture. The "aggregating commands" scenario can then be completed without having to rework the software's design.

d. Task oriented help system

Traditional help systems require the users to interrupt their task to find the relevant information. This interruption often is time consuming, requiring them to glance through extensive topics and information, and irritating enough that most users simply abandon their quest for help [7].

Context sensitive help system alleviate this problem by associating help with functions or interface objects that is currently the focus of the user's activity, thereby relieving the user to search across the whole documentation. Task oriented help system associates the help to the context of the task being executed. Provided the interface is itself task oriented, the task context is available to the help system and can thus be used for focusing on the specific information that is most relevant [9].

Although this work is still under way, our preliminary analysis of the changes to the GIMP architecture to support task-oriented help indicates that it can be done with a combination of the ScriptFu scripting tool and the plug-in feature. Information about the task performed can be collected through hooks in the interface components

and sent to a help system that will provide task-sensitive information. Although it requires a significant programming effort, it does not affect the GIMP architecture, but instead it can use existing features from it to allow this functionality.

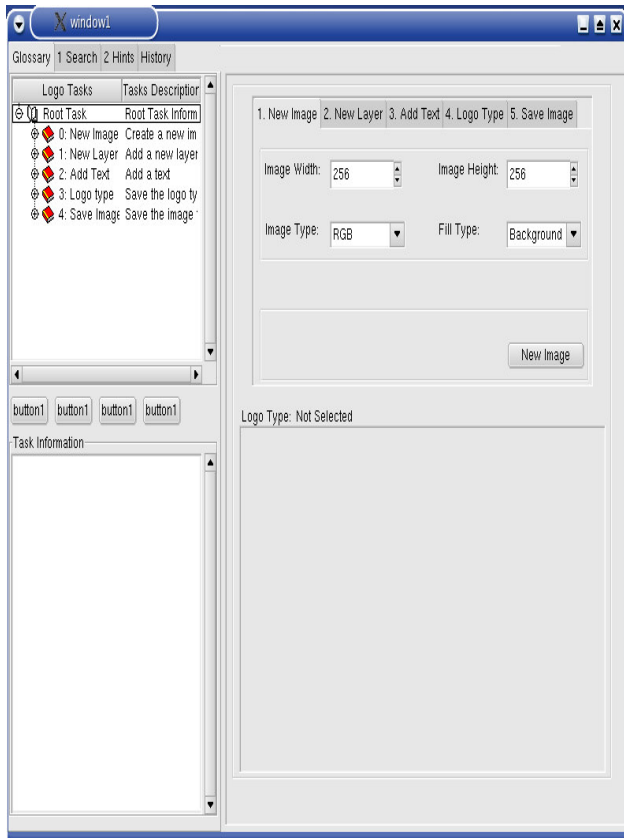


Figure 2. Task oriented GIMP's interface

4. Conclusion

In this article, we presented a case study that consisted of stating that the 3 identified scenarios that can greatly increase GIMP's usability can be implemented into its existing architecture. These modifications rest on the scripting and plug-ins facilities provided by GIMP.

Figure 2 shows the layout of the task-oriented interface for the creation of a logo in the center and its related task oriented help system on the right. The high-level task, which consists of the creation of a logo, is composed of five low-level tasks: create new image, create new layer, add text, choose logo type and save image. The interface regroups only the functionalities required to create the logo since there is no need to see the other functionalities not related to the logo. Also, the user interface respects the order of the tasks that need to be completed. When a user decides to accomplish one specific low-level task, like creating a new image, they will be able to perform the operations only related to the creation of the image. Once

they have finished with the image creation, they can move on to another low-level task like the creation of a layer.

Usability and good user assistance through task sensitive help systems, contribute to the intuitiveness of software, and may significantly affect its accessibility to standard user categories: novices, occasional users and experts. The current case study shows that such changes are feasible without incurring architectural changes, but only because the underlying architecture was powerful and flexible enough to accommodate such changes, thereby confirming Bass et al. thesis that usability driven interface design cannot be decoupled from the application's architecture. However, it also demonstrates that single and specific architecture patterns should not be considered as the sole nor the best mean to address usability scenarios, but instead that there are many ways to design and implement usability requirements in a software architecture.

As Armour argued [8] "the hard part of building systems is not building them, but knowing what to build; it is in-acquiring the necessary knowledge and identifying the areas of ignorance". Many designers think they know everything necessary about users and their needs. We have presented a way, by which user needs and the user point of view can be stated in a specific usability scenario in a task oriented development process.

5. References

- [1] J. Nielsen, R. Mack, *Usability Inspection Methods* (John Wiley & Sons, New York, 1996).
- [2] J. Nielsen, *Usability Engineering* (Academic Press Inc, 1993)
- [3] X. Ferre, N. Juristo, H. Windl, L. Constantine, Usability Basics for Software Developers, *IEEE Software*, 18(1), January/February 2001, 22-29.
- [4] A. Andrés, J. Bosch, A. Charalampos, R. Chatley, X. Ferre, E. Folmer, N. Juristo, J. Magee, S. Menegos, A. Moreno, *Usability attributes affected by software architecture-Deliverable 2. STATUS project*, June 2002. <http://www.ls.fi.upm.es/status>
- [5] P. Clements, R. Kazman, M. Klein, *Evaluating Software Architectures: Methods and Case Studies* (Addison-Wesley, MA, 2001)
- [6] L. Bass, B.E. John, J. Kates, *Achieving Usability through Software Architecture*, CMU/SEI-TR-2001-005 (Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2001)

<http://www.sei.cmu.edu/publications/documents/01.reports/01tr005.html>

- [7] T. Grayling, A Usability Test of Two Browser-based Embedded Help Systems, *Technical Communication*, 45(2), May 2002.
- [8] P.G. Armour, The five orders of ignorance
Communications of the ACM, 43(10), 2000, 17-20.
- [9] S. Deborah, J. Ray, Embedded Help: Background and Applications for Technical Communicators,
Technical Communication, 45(2), May 2002.
- [10] S. Shawren, P. Kotze, An Overview of Systems Design and Development Methodologies with Regard to the Involvement of Users and Other Stakeholders.
Proceedings of SAICSIT 2003, 37-47
- [11] ISO 13407:1997, *Human-Centered Design Processes for Interactive Systems*