The Software Architecture of the GIMP

Earl Oliver, Jaime Ruiz, Steven She, and Jun Wang

December 7, 2006

Abstract

In this paper, we examine the architecture of the GNU Image Manipulation Program (GIMP) from two separate points of view. We begin by constructing a conceptual architecture using available developer documentation. We then describe the GIMP's control flow based on our conceptual architecture and describe how the architecture allows for the extensibility of the GIMP.

The conceptual architecture is then compared to the concrete architecture, the architecture defined by the implementation of the system. We present a landscape diagram as a visual representation of the GIMP's concrete architecture. Our analysis decomposes the concrete architecture into major top-level components and provides a sub-architectural analysis of the GIMP Core.

Contents

1 Introduction **1**

1.1 About The GIMP 1

1.2 The History of GIMP 3

1.3 About this Document 3

2 Conceptual Architecture **4**

2.1 High-level components 4

2.1.1 User Interface 4

2.1.2 Extensions 5

2.1.3 GIMP Core 6

2.1.4 Plug-ins 7

2.1.5 GIMP Libraries 7

2.2 Components Interaction 8

2.3 Architecture Style 8

2.3.1 Libraries Layer 8

2.3.2 Image Manipulation Layer 9

2.3.3 User Interface Layer 9

2.3.4 Extensibility of GIMP 10

2.3.5 Plug-ins 10

2.3.6 Script-Fu 11

3 Concrete Architecture **11**

3.1 High Level Concrete Architecture 12

3.1.1 User Interface 12

3.1.2 Extensions 14

3.1.3 Plug-ins 14

3.1.4 GIMP Libraries 15

3.1.5 The GIMP Core 15

3.1.6 Actions 16
3.1.7 Libraries 17
3.1.8 Data Model 17
3.1.9 Procedural Database 17
3.1.10 Configurator 18
3.1.11 Inner Core 18
3.2 Concurrency 19
3.2.1 Event Driven Programming 19
3.2.2 Plug-in Concurrency 19

4 Conclusion **19**

1 Introduction

A raster graphics editor is a computer program that allows users to paint and edit pictures interactively on the computer screen and save them in one of many popular "bitmap" or "raster" formats such as JPEG, PNG, GIF and TIFF.

A raster graphics image, digital image, or bitmap, is a data file or structure representing a generally rectangular grid of pixels, or points of colour, on a computer monitor, paper, or other display device. The colour of each pixel is individually defined; images in the RGB colour space, for instance, often consist of coloured pixels defined by three bytes-one byte each for red, green and blue value. Less colourful images require less information per pixel; an image with only black and white pixels requires only a single bit for each pixel. Raster graphics are distinguished from vector graphics in that vector graphics represent an image through the use of geometric objects such as curves and polygons.

A bitmap or a raster image corresponds bit for bit with an image displayed on a screen, probably in the same format as it would be stored in the display's video memory or maybe as a device independent bitmap. A bitmap is characterized by the width and height of the image in pixels and the number of bits per pixel, which determines the number of colours it can represent.

A coloured raster image (or pixmap) will usually have pixels with between one and eight bits for each of the red, green, and blue components, though other colour encodings are also used, such as four- or eight-bit indexed representations that use vector quantization on the (R, G, B) vectors. The green component sometimes has more bits than the other two to allow for the human eye's greater discrimination in this component.

Two of the most popular raster graphics editors are the closed-source Adobe Photoshop and the open source GIMP. In this paper we will focus on the GIMP because of the ability to examine the source code.

1.1 About The GIMP

The GIMP is a multi-platform raster graphics editor. GIMP is an acronym for GNU Image Manipulation Program. The GIMP is suitable for a variety of image manipulation tasks, including photo retouching, image composition, and image construction.

It has many capabilities. It can be used as a simple paint program, an expert quality photo retouching program, an online batch processing system, a mass production image renderer, and an image format converter.

GIMP is expandable and extensible. It is designed to be augmented with plug-ins and extensions to do just about anything. The advanced scripting interface allows everything from the simplest task to the most complex image manipulation procedures to be easily scripted.

One of the GIMP's strengths is its free availability from many sources for many operating systems. Most GNU/Linux distributions include the GIMP as a standard application. The GIMP is also available for other operating systems such as Microsoft Windows or Apple's

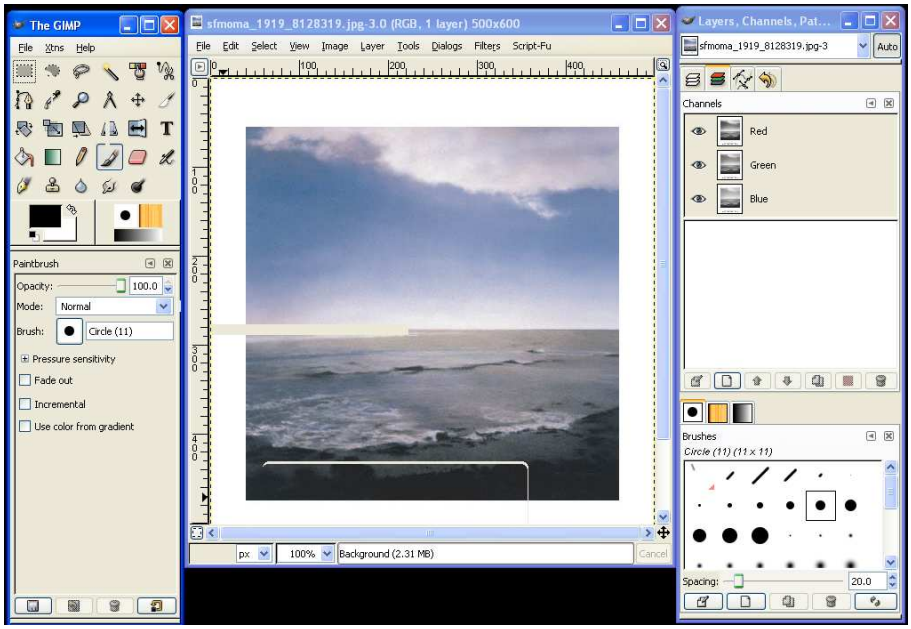

Figure 1: Gimp 2.2 on Windows

Mac OS X.

The following list is a short overview of some of the features and capabilities which the GIMP offers [17]:

- A full suite of painting tools including brushes, a pencil, an airbrush, cloning, etc.
- Tile-based memory management, so image size is limited only by available disk space.
- Sub-pixel sampling for all paint tools for high-quality anti-aliasing.
- Full Alpha channel support for working with transparency.
- Layers and channels.
- A procedural database for calling internal GIMP functions from external programs, such as Script-Fu.
- Advanced scripting capabilities.
- Multiple undo/redo (limited only by disk space).
- Transformation tools including rotate, scale, shear and flip.
- File formats supported include GIF, JPEG, PNG, XPM, TIFF, TGA, MPEG, PS, PDF, PCX, BMP and many others.
- Selection tools including rectangle, ellipse, free, fuzzy, bezier and intelligent.
- Plug-ins that allow for the easy addition of new file formats and new effect filters.

1.2 The History of GIMP

The original GIMP was created of necessity by Spencer Kimball and Peter Mattis in August, 1995. They were working on a project for a computer science class at the University of California, Berkeley. They decided, in extremis, to try something different–a pixel-based image manipulation program.

Six months later, in February of 1996, an early beta version of the GIMP was released onto the Internet. This early GIMP, version 0.54, relied on the Motif widget library for the core windowing capabilities. The use of this commercial software product went contrary to the currents of the open-source movement, which were beginning to gather momentum around the growing base of Linux users. A decision was made to generate an independent, open widget set based on an equally open core drawing library. The GIMP toolkit (GTK) was born. The first version of the toolkit was released in July of 1996.

At this point there were already many developers around the world working with Kimball and Mattis on debugging and improving the GIMP. The GIMP had several major weaknesses as a pixel-based image-manipulation program because it had an inefficient memory management system and did not have layers. After a long wait, version 0.99 was released in February of 1997. This version of the GIMP contained the main features and architecture of the GIMP as it is distributed today. It was layer-based, with a tiled memory management scheme built in, and a large number of plug-in filters had been written for it.

Finally, on May 19, 1998 the GIMP passed from beta development to its first stable release. The GIMP version 1.0 was delivered to the Internet community. Since then, development of the GIMP has continued at a rapid pace.

GIMP 1.0 evolved gradually into the very stable and widely used 1.2 release. Three years later, as the GIMP development came closer to the next stable release, they decided that the level of fundamental change to the inner workings of the program justified calling the new stable version 2.0. GIMP 2.0.0 was released on March 23, 2004. For GIMP 2.2, the developers aimed at a short cycle, adding a number of important features that did not require instability-inducing low level changes. GIMP 2.2.0 was released on December 19, 2004. The latest stable version of The Gimp is version 2.2.13 and contains over 650,000 lines of code.

1.3 About this Document

The purpose of this document is to explain the software architecture of the GIMP application. We achieve this by creating describing the conceptual architecture and concrete architecture of the system and then highlighting conflicts between them.

The paper is organized as follows. In the next section we describe the conceptual architecture of the GIMP. We then describe the concrete architecture by describing the components that make up the GIMP and how these components interact with each other. As part of our concrete analysis we also describe how the concrete architecture differs from the conceptual architecture of the GIMP.

2 Conceptual Architecture

The conceptual architecture described by this document was obtained from a top-down fashion by examining available documentation about the GIMP. Documentation included but was not limited to: end user documentation, developer documentation, documentation created from tools that extract information from source code, online articles, and books. The conceptual analysis also considered the organization of the GIMP source tree. The organization of the directories within the GIMP provide hints as to how the authors intended to structure the application. For the conceptual analysis we did not examine the code in itself.

Our conceptual analysis revealed several top-level components of the GIMP architecture. Their interactions were observed and the architectural style of the system was identified. Like most large scale open source projects, we found that the GIMP's architecture is also extensible. The following subsections describe out findings.

2.1 High-level components

Conceptually, the GIMP's architecture can be decomponsed into five architectural components. These components are:

1. User Interface
2. Extensions
3. GIMP Core
4. Plug-ins
5. GIMP Libraries

The organization and interaction of the five high level components is represented in figure 2.

The style of the GIMP architecture is discussed in section 2.3; however, it is apparent from figure 6 that GIMP has adopted a layered architectural style.

The following sections provide a detailed description of each high level component.

2.1.1 User Interface

The User Inteface (UI) component presents the application controls and data model to the user. The UI defines how the GIMP system should look and feel by through a generic Graphical User Interface (GUI) layout. The generic layout specifies, for example, where the menu bar should be, where the palette should be, how an icon should be displayed, and other common GUI attributes. In addition to these default GUI components, the UI layer exposes a public API to supports dynamic components such as additional menus, tools, icons, etc. Lower layers register components through the UI API and provide a callback

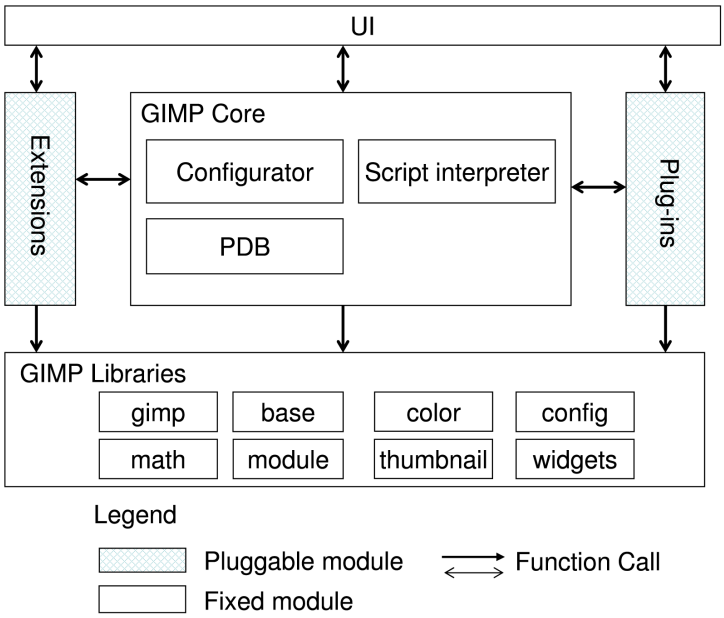

Figure 2: Conceptual Architecture of GIMP

function. For example, when the user clicks on an icon, the associated callback function will be called to handle the user's gesture. As we will see later, most of the functionality of GIMP is implemented using extensions and plug-ins. Since extensions and plug-ins are dynamic components, they must register themselves with the UI layer at startup. The main function of the UI layer is therefore to provide a place holder for dynamicly loaded GUI components.

2.1.2 Extensions

Extensions are separate processes that are invoked by the GIMP Core. These processes implement a specific interprocess communication (IPC) mechanism that allows them to receive data from the GIMP. Extensions are designed to provide functionality to the user that does not alter the image. More precisely, extensions are read-only operations on GIMP's underlying data model. For example, the ability to save or print a file is implemented as an extension within the GIMP [4]. Neither of these operations alter the image; where as a blur or colour change would.

On startup, extensions register themselves with the UI and the GIMP Core's Procedural Database through the same IPC mechanism mentioned above. Since extensions are implemented as separate processes, they may be implemented in various programming languages such as PERL, Python, Java, C and C++ [18].

2.1.3 GIMP Core

The GIMP Core handles the system level tasks such as the configuring the GIMP system, loading extensions, loading plug-ins, maintaining the PDB, and acting as a central administrator between extensions and plug-ins. The GIMP Core is thte critical component in connecting the components between this local layer as well as the whole architecture.

The conceptual GIMP Core is broken down into three components:

- Script Interpreter
- Configurator
- Procedural Data Base (PDB)

Script Interpreter The script interpreter provides a user with the ability to created automated functionality to the GIMP. Using procedures defined by the Procedure Database, the script interpreter processes scripts written in any of the supported scripting languages. Some of the supported scripting languages include Scheme, Java, Lua, Phyton and Perl. Scripts can be loaded from an external file or entered into a command console like the one shown in figure 3.

Figure 3: The Script Interpreter

Configurator The Configurator is responsible for the persistent configuration of the GIMP components between sessions. Settings such as the unit of measurement to use (centimetres, inches, or pixels), GUI layout of tools, and user-defined palettes should be stored and read by the Configurator.

Procedural Database The procedural database is a registry of the functionality of the GIMP and its plug-ins. The procedural database allows developers and users to access already existing functionality using new plug-ins or scripts. Any functionality that a developer would like to expose to other developers or users must be registered with the procedural database.

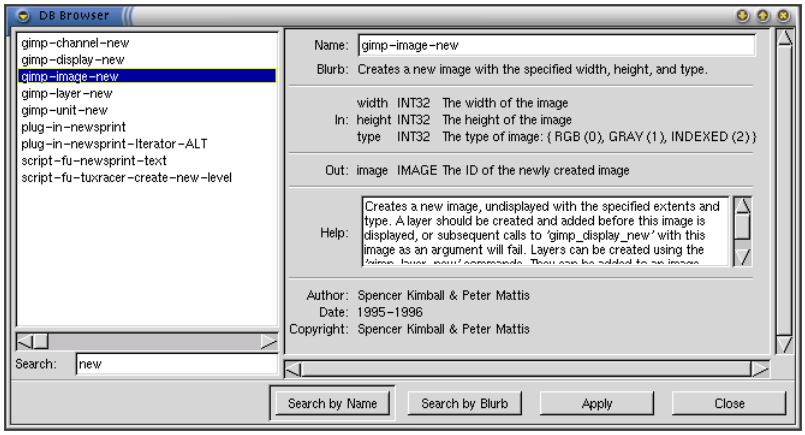

Figure 4: The Procedural Database (PDB)

2.1.4 Plug-ins

Plug-ins are separate processes that are invoked by the GIMP Core. Like extensions, plug-ins also implement a specific IPC mechanism to receive data from the Core. Plug-ins differ from extensions in that they are designed to add graphic transformation functionality to the GIMP. For instance, blurring is implemented as a plug-in. Images are passed to the blur plug-in, transformed, and passed back to the GIMP.

On startup, plug-ins also register themselves with the UI and the GIMP Core's PDB. Plug-ins can also be developed in a several common programming languages. Conservatively speaking, there are more than 200 plug-ins currently available for the GIMP [16]. The extensibility afforded by plug-ins is further discussed in section 2.3.4.

2.1.5 GIMP Libraries

The GIMP Libraries contain many common functions within the GIMP. Examples of these functions include: basic image manipulation functions, colour manipulating functions, configuration functions, mathematical functions, image thumbnail functions, and UI widgets functions. The GIMP libraries form the foundation of the GIMP system by providing services to the GIMP Core, extensions and plug-ins.

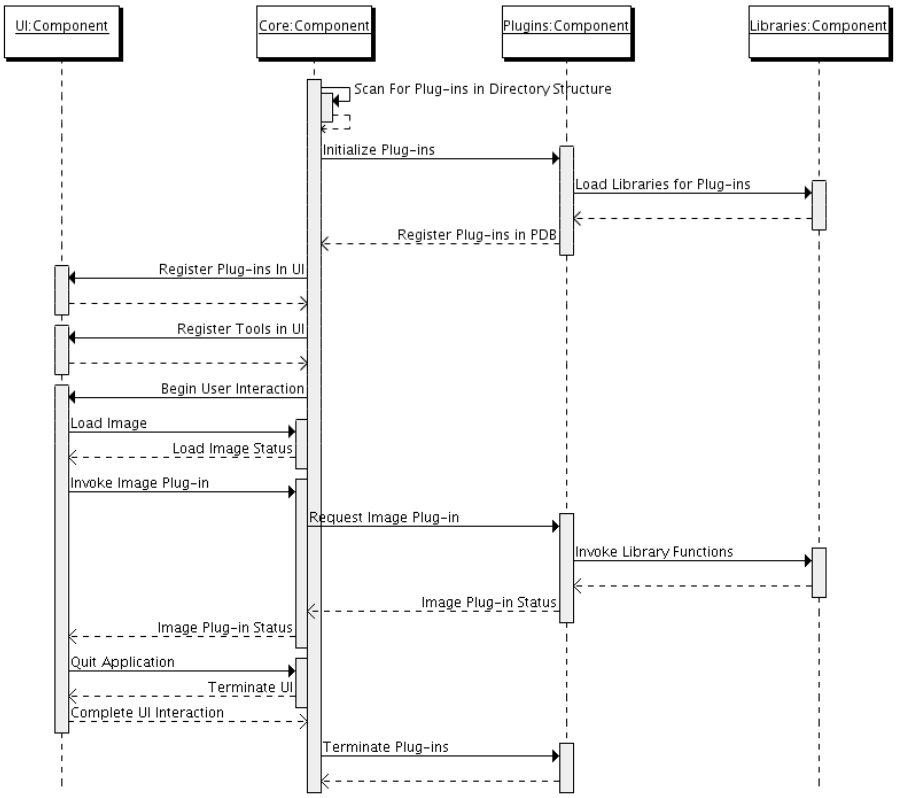

Figure 5: Interactions among high level GIMP's components

2.2 Components Interaction

The sequence diagram shown in figure 5 illustrates typical interactions among four of the major components. For the sake of simplicity, the extensions component is omitted from this diagram due to its similarity to the plug-in modules.

2.3 Architecture Style

By abstracting the details of the GIMP Core, we can condense our full conceptual architecture (Figure 2) into the following simplified conceptual architecture (Figure 6). From our simplified architectural view, we conclude that the GIMP has a layered architectural style with three distinct layers: *Libraries*, *Image Manipulation*, and *User Interface*.

2.3.1 Libraries Layer

The lowest layer in the GIMP architecture contains the GIMP Libraries component. The GIMP Libraries contains a large amount of mathematical, colour and graphics manipulation functions. These functions form the foundation of the GIMP application by providing a

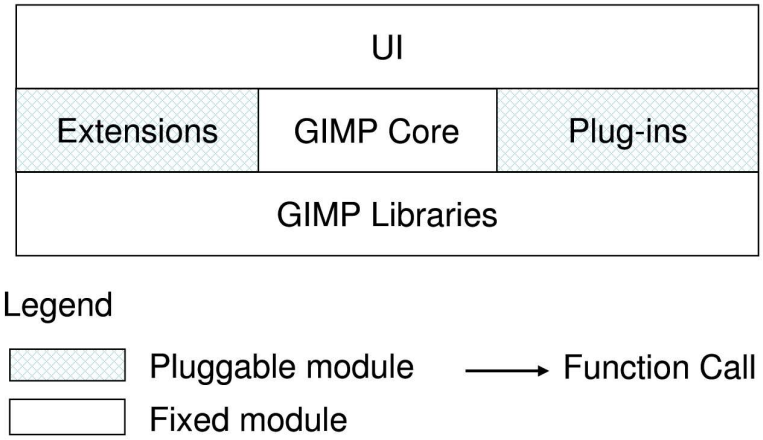

Figure 6: Layered Architecture Style of GIMP

facade layer to the mathematical and graphical operations used by the layer above: the Image Manipulation layer.

2.3.2 Image Manipulation Layer

The GIMP application grows from a series of mathematical and graphical libraries, to a full-fledged image manipulation program through the Image Manipulation Layer. The Image Manipulation Layer consists of three components: the extensions, the GIMP Core and the plug-ins. The GIMP Core acts as a central administrator to the extensions and plug-ins components. The GIMP Core handles the configuration of the application, the loading of extensions and plug-ins, the update and maintenance of the PDB, and the communication of GIMP tools with extensions and plug-ins.

The extensions component extends the functionality of the GIMP on the application level. Extensions may add features relating to hardware, such as a customized print dialog, a scanner wizard or scripting capability that will allow the GIMP Core to interpret plug-in modules written in a particular scripting language. The plug-ins component augment the image manipulation features of the GIMP. For example, a new blurring technique can be added to the GIMP as a plug-in.

2.3.3 User Interface Layer

The User Interface layer contains the code that displays the windows, menus and available tools to the user. The User Interface layer is implemented such that an action performed by the user through the UI would invoke a function call (or multiple function calls) to the layer below. The abstraction of the UI allows developers to create different user interfaces that use the same image manipulation features. An example of this can be seen in the derivative GIMP projects GIMPShop [7] and MacGIMP [8] which have different User Interface layers while keeping the same Image Manipulation and Libraries layers. GIMPShop uses a user

interface similar to the UI present in Adobe Photoshop. MacGIMP uses a user interface that is better integrated with the Mac OS X style guidelines.

2.3.4 Extensibility of GIMP

A modular and extensible architecture is an important element to the design of a large scale software system such as GIMP. A modular design allows the functionality of the system to grow linearly as the number of developers contributing to the project increases. Developers contribute to the project by developing their own modules in parallel, while a small core group focuses on the core of the system [15]. These modules are typically loaded by the software system dynamically, on demand, to provide additional functionality. Their integration is often seemless and transparently to the end user. This architectural style can be observed in many large scale open source projects such as:

- Apache (Apache Modules)
- Mozilla (Extensions and Themes)
- Sendmail (Milters)
- Asterisk (Applications, Channels, Codecs, File Handlers)
- and many others.

GIMP also supports modular development through extensions, plug-ins and scripting.

2.3.5 Plug-ins

As mentioned in section 2.1.4, GIMP plug-ins are designed as external applications that are executed and controlled by the main GIMP application. This separation protects the GIMP Core from plug-in implementation faults. For example, if a module that is dynamically linked crashes it could bring down the entire application. In the GIMP, a faulty plug-in will crash and not effect the GIMP Core or the data (images) managed by it [17]. Data is passed to the plug-ins through a rich set of application APIs. These APIs are well defined and available for download from GIMP Developer site [6].

While designing plug-ins as disjoint applications has advantages, it introduces a race condition on modifying images within the GIMP Core. The GIMP plug-ins are invoked by the GIMP Core as asynchronous operations. GIMP plug-ins do not have a mechanism for being informed about changes to the data model within the GIMP Core. Therefore, if an image is altered by another application after a plug-in has been started, the plug-in "will often crash, and when it doesn't [crash] will usually given a bogus result" [17]. Due to the GIMP's lack of synchronization on the underlying data model, users of plug-ins must be careful to invoke them sequentially.

GIMP plug-ins are managed and distributed through a web service called the GIMP Plug-in Registry [16]. Although this service is not part of the GIMP architecture, it is an important mechanism for organizing extensions in the GIMP.

2.3.6 Script-Fu

Script-Fu is an interpreted language based on Scheme that allows users or plug-ins to automate complex and/or frequent tasks within GIMP. Script-Fu is capable of querying functions within the GIMP Procedural Database [14]. This capability allows Script-Fu scripts to "glue" together plug-ins and perform sequential multi-plug-in operations on images.

In addition to automating functions, Script-Fu is capable of defining entire plug-ins and their associated user interfaces. Following is a sample plug-in defined using Script-Fu [14]:

```
(define (my-demo-box value adj1 adj2 image drawable toggle
                 pattern string font color option gradient)
  (print "Do nothing"))

(script-fu-register "my-demo-box"
            "<Toolbox>/Xtns/Script-Fu/My Stuff/Demo Box..."
            "Do nothing"
            "Joe User"
            "Joe User"
            "August 2000"
            ""
            SF-ADJUSTMENT "SF-ADJUSTMENT (slider)" '( 30 1 2000 1 10 1 0)
            SF-ADJUSTMENT "SF-ADJUSTMENT"         '(400 1 2000 1 10 1 1)
            SF-COLOR      "SF-COLOR" '(255 0 255)
            SF-DRAWABLE   "SF-DRAWABLE" 0
            SF-FONT       "SF-FONT" ""
            SF-GRADIENT   "SF-GRADIENT"  "Golden"
            SF-IMAGE      "SF-IMAGE" 0
            SF-OPTION     "SF-OPTION" '("Option 1" "Option 2" "Option 3")
            SF-PATTERN    "SF-PATTERN" "Wood"
            SF-STRING     "SF-STRING" "Test String"
            SF-TOGGLE     "SF-TOGGLE" TRUE
            SF-VALUE      "SF-VALUE" "0"
            SF-FILENAME   "SF-FILENAME" "/")
```

When invoked by the GIMP, this script will produce the UI panel shown in figure 7.

As mentioned above, Script-Fu scripts are processed by the Script Interpreter sub-component of the GIMP Core.

3 Concrete Architecture

The concrete architecture described by this document was obtained using a bottom-up approach by examining the source-code of GIMP. The source code was analyized using ASX and LSEdit. ASX is a fact extraction tool that extracts source information from C, $C++$, assembler, object, libraries, dynamic libraries and executables, in a format that may then

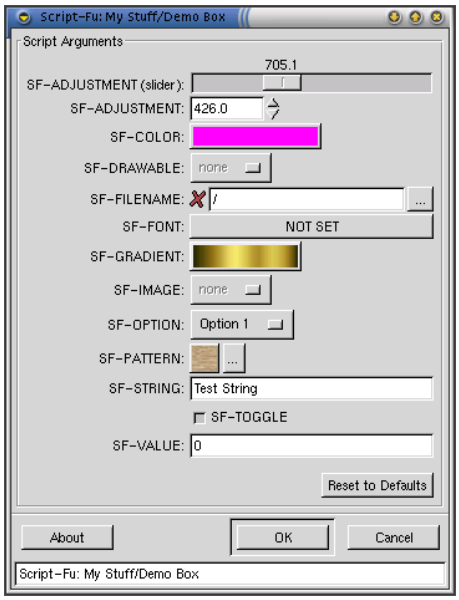

Figure 7: Gimp 2.2 UI Panel [14]

immediately be visualized using LSEdit. LSEdit is a graph visualization tool that is particularly suited to exploration and editing of software "landscapes". Once ASX has extracted information from the source code, we used LSEdit to cluster the source code into subsystems using our the conceptual architecture as a guide.

3.1 High Level Concrete Architecture

Our concrete analysis of the GIMP revealed little differences between our conceptual architecture and the actual architecture. From a high level perspective, our concrete architecture is nearly identical to our conceptual analysis. There was some variation in inter-component communication and a distinct separation between the Core and plug-ins. The following sections present the findings of our concrete analysis for each high level component of the GIMP.

3.1.1 User Interface

In our conceptual analysis, the GIMP UI was a viewed as a single component. Our concrete analysis revealed the the GIMP UI actually consists of several sub-components. The sub-architecture of the GIMP UI consists of the following components:

- Dialogs - implementation of the GIMP template dialog boxes. Templates allow plug-ins and the Core to invoke dialog boxes without having to understand how a dialog box is implemented or displayed.

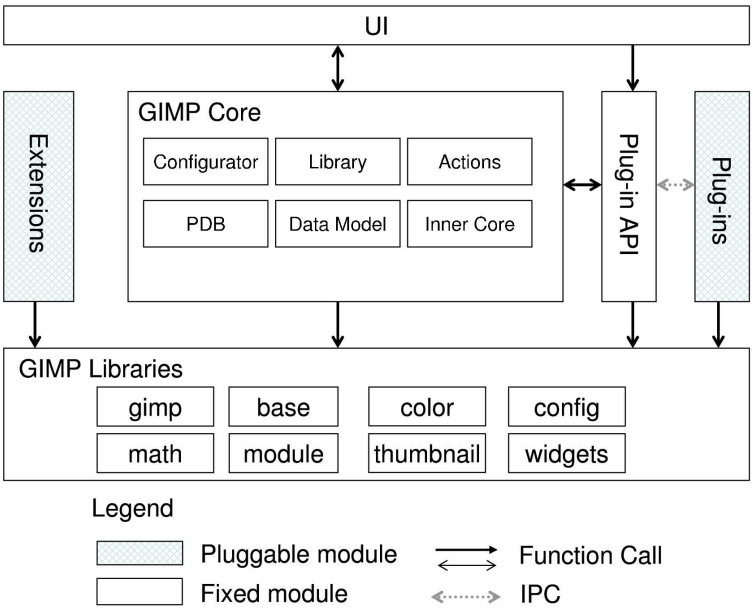

Figure 8: Overview of the GIMP concrete architecture

- Menus - the functionality aforded by the menu sub-component is equivalent to that of the dialogs. Plug-ins and the Core have the ability to specify different types of menus and menu items through generic menu templates.
- Widgets - the widget sub-system implements complex compound-widgets such as tool paletes and document windows. Examples of these widgets include: *gimpeditor.c*, *gimpdock.c* and *gimphelp.c*.
- Display - the display serves as a place holder for UI components. The display component is responsible for organizating and managing UI components.
- Graphical User Interface (GUI) - the GUI component serves as the API between the UI and the Core and plug-ins. It also specifies common settings within the UI component. The GUI sub-component is the lowest layer of the GIMP's UI system.

These sub-components are implemented using a series of primative widgets such as buttons, sliders, windows, etc. The GIMP has been designed to be platform-agnostic. As such, the GIMP had stubbed out OS-specific widgets into a library called *libgimpwidgets* library. The use of this library was not anticipated in our conceptual architecture and introduces interaction between the UI and the library components. This interaction also violates the GIMP's conceptual layered architectural style.

The GIMP UI communicates with the GIMP Core and plug-ins through a well defined set of APIs. This clean separation between the UI "layer" and the rest of GIMP has lead to the development of third party projects such as GIMPShop [7], which completely replace the GIMP UI.

3.1.2 Extensions

The extensions component of the GIMP was smaller than anticipated. The extensions component consists of 12 files that performed functions that did not alter the document model. None of the extensions interacted with the GUI. As such, there was no interaction between the extension and UI sub-systems. Extensions are not invoked as applications by the GIMP Core at runtime; however, they are dynamically loaded. Also contrary to conceptual expectations, there was no code dependencies between the extensions and GIMP Core. Extensions only made function calls to *libgimpmodules* within the GIMP Library component.

3.1.3 Plug-ins

Our concrete analysis of plug-ins differed greatly from the conceptual architecture. Conceptually, plug-ins have two-way communication with functions within the GIMP UI. In practice, this is not the case. The only connection between plug-ins code and the UI component is a single dependency from: *mapobject_image.c* and *mapobject_preview.c* to the UI's *about-dialog.c* file. In both cases, the files were referencing a global variable called *background* which contains the background colour of the dialog box. We attribute this relationship to a developer's 'hack', and exclude it from the concrete architecture.

In our conceptual analysis we were convinced that the plug-ins were able to alter the GIMP UI (through some IPC mechanism within the Core); however, IPC communication from GIMP plug-ins is actually handled through a new component called the GIMP plug-in API. The plug-in API relays communication from the plug-ins to the GIMP Core. It is the Core which invokes dialog boxes, menus, etc. on behalf of the plug-ins. This separation between plug-ins and the GIMP UI is made possible through an abstract UI definition. Defining the UI in a textual form is a common technique used in many open source and highly modular software systems. For example, Mozilla has defined the language called XML User Interface Language, which allows plug-ins to define their associated UI components using XML [9].

Conversely, the GIMP UI makes several calls to the plug-ins API. In particular, the GIMP UI calls functions within *plug-in-run.c*, *plug-ins.c* and *plug-in-proc-def.c*. The file *plug-in-run.c* contains several functions for invoking, querying state, restarting and stopping GIMP plug-ins. This file is accessed by the UI when the user chooses to apply a particular plug-in to their image. Functions within *plug-in-run.c* are also called by the GIMP Core's Actions sub-system.

The plug-in API also interacts with the GIMP Core through the Core's configuration and PDB sub-systems. The plug-in API contains a parser, which parses the configurations files for each plug-in. This data is passed to the configuration sub-system's *gimpscanner.c* and *gimpconfigwriter.h* files at startup. Functions within *gimpconfig-error.c* are also called by the plug-in API to report plug-in configuration errors. The most significant relationship between the plug-in API and the GIMP Core is through the PDB. Figures 9(a) and 9(b) illustrate this relationship. On startup, plug-in API calls several functions within the PDB's *procedural_db.c* file. This database contains the information needed to invoke plug-ins. The PDB also calls several functions within the plug-in API. The GIMP Core PDB is further discussed in section 3.1.9.

As anticipated in our conceptual architecture, both the the plug-ins and plug-in APIs make

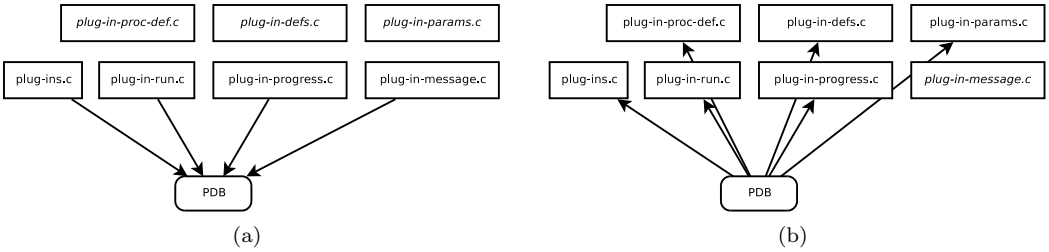

Figure 9: Interaction between the (a) Calls from the plug-in API to the PDB, (b) Calls from the PDB to plug-in API.

calls into the GIMP Library component. As mentioned above, the plug-ins and the plug-in API interact through a common IPC mechanism. This IPC mechanism is implemented within *libgimp*. The *libgimpmath* and *libgipcolor* libraries are also used extensively by individual GIMP plug-ins.

3.1.4 GIMP Libraries

The concrete interaction between GIMP and its libraries has minimal deviation from our conceptual interpretation. The GIMP libraries consisted of:

- *libgimp*
- *libgimpbase*
- *libgimpcolor*
- *libgimpmath*
- *libgimpmodule*
- *libgimpthumb*
- *libgimpwidgets*

These libraries are mutually disjoint and organized in their own directories within the source tree. These libraries are frequently utilized for graphics processing in other applications. We did not find evidence of a configuration library.

The relationship between the UI and the libraries was unexpected. Conceptually, the layered architecture did not permit the UI from making library calls. It was believed that the library functions were intended for image processing by the Core and plug-ins/extensions; however, the UI calls functions within all of the above libraries.

3.1.5 The GIMP Core

As mentioned above, our concrete analysis of the GIMP revealed many differences between the conceptual and concrete architecture of the GIMP Core.

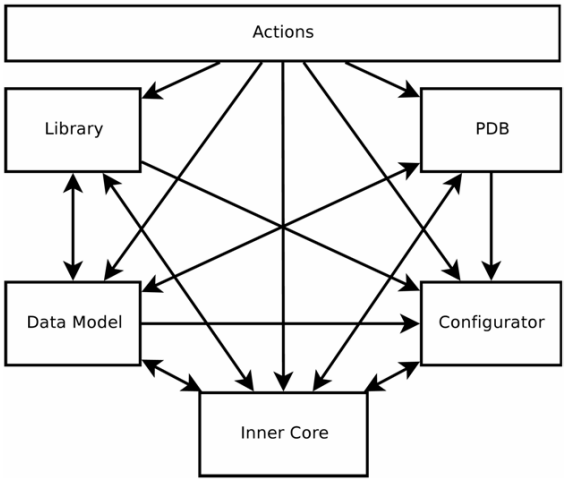

Figure 10: Concrete architecture of the GIMP Core

The concrete architecture of the GIMP Core did not contain a separate Script Interpreter sub-component. The scripting capabilities were implemented as a plug-in which accessed functions within the GIMP through the PDB and the Actions sub-component.

Our concrete analysis revealed the following sub-components of the GIMP Core:

- Actions
- Libraries
- Data Model
- Procedural Database
- Configurator
- Inner Core

The following sections provide a detailed outline of each sub-system within the GIMP Core.

3.1.6 Actions

The Actions sub-component exposes functions from the GIMP Core to the User Interface component. This sub-component acts as the facade design pattern giving developers a simplified interface to the GIMP Core functionality. The source files for the Actions sub-component reside within the app/actions directory.

The Actions sub-component was not present in our conceptual architecture. In our conceptual architecture, the User Interface directly communicated with the PDB within the Core and plug-ins. However, since the plug-ins were implemented as separate executables, the existence of awit facade layer provide the UI component with an unified interface for accessing

all of the inner core sub-components. The Actions sub-component provide easier evolution of the GIMP Core without having to propagate all changes to the UI component.

3.1.7 Libraries

The Libraries sub-component contains functions for the image manipulation tools provided to the UI component. The air brush tool, colour picker, and the bucket fill tool are several examples of the tools contained within this sub-component. The tools contained in the Core Libraries sub-component are GUI-centric and implement several static functions for UI event interaction. These static functions include button press, button release, modifier key, operation update and cursor update events. The actual event functions implemented by each tool differ in their respective use by the user. For example, the gradient tool implements drag and drop actions, while the paint bucket tool implements only the button click and modifier keys. The Core Libraries source files are located in the app/tools directory of the GIMP.

In our conceptual architecture, the Libraries sub-component was integrated within the GIMP Core component. However, our concrete analysis had revealed that the image manipulation tools were a separate sub-component within the GIMP Core. This sub-component provides a more structured architecture of the GIMP Core, separating the image manipulation tools from the other functions of the GIMP Core.

3.1.8 Data Model

The Data Model sub-component contains the functions related to the creation and modification of the image internally within the GIMP. Reading and writing the vector and raster information in the image to the file system is included within this sub-component. An image is stored in a *GimpImage* data structure, and pointers of the *GimpImage* are passed around the application. The *GimpImage* data structure is serialized into the XCF file format. A function called *xcf_load_image(Gimp*,XcfInfo*)* returns a *GimpImage* data strcture when supplied an XCF file through the *XcfInfo* data structure. A file that is not of GIMP's XCF format is loaded by a plug-in and converted to conform to the *GimpImage* data structure.

The Data Model sub-component did not exist in our conceptual architecture. While constructing the conceptual architecture, we neglected the importance and size of the necessary structures and functions to support the image. During our concrete analysis, we found a considerable amount of code dedicated to the storage of the image. There were also functions for serializing vector information. As a result, we grouped both the vector serialization and general image serialization functions to form the Data Model sub-component.

3.1.9 Procedural Database

The PDB is implemented as a hash table data structure in the GIMP. Procedures are registered using *procedural_db_register(Gimp*, ProcRecord)* and executed using the *procedural_db_execute(Gimp*,GimpContext*, GimpProgress*, gname, Argument*)* function. The PDB calls the Libraries sub-component in order to register a new user tool, such as a new

paint brush. The PDB also calls the Inner Core sub-component in order to register new menu and dialog items.

A procedure is registered in the PDB using the *ProcRecord* data structure with the following group of fields: general information, GIMP procedure type, input arguments, output arguments, and execution method. General Information includes the name of the procedure, the author, a short description and other descriptive fields. The GIMP procedure type is a variable of *GIMPPDBProcType* and can set to internal, plug-in, or extension. The input and output arguments are arrays of type *ProcArg*. *ProcArg*s store the argument type along with a name and description. The execution method is an exclusive variable which can be set to a data structure containing information required for an internal, plug-in, or extension.

The PDB is integral to the use of plug-ins and extensions within GIMP in the concrete architecture. Plug-ins and extensions must register their procedures with the PDB in order to be used. This behaviour matches the behaviour described in our conceptual architecture.

3.1.10 Configurator

The Configurator sub-component is responsible for maintaining and initializing the configuration of the GIMP's components upon start up and saving of the configuration when required. The Configuration sub-component also contains functions to serialize and deserialize a configuration file written in a LISP-like syntax. There are also functions that handle XML parsing. Our concrete analysis revealed that the menu and dialog structures were written in XML. As a result, it is logical that the Configurator contains XML parsing routines. The function of the concrete Configurator match the function of our conceptual Configurator.

3.1.11 Inner Core

The Inner Core sub-component contains all of the code that pieces the various part of the GIMP together. The Inner Core contains a *GimpClass* data structure which is used to contain the various components of the GIMP. The *GimpClass* data structure also contain lists of plug-ins, a GUI virtual table (which is connected to the GUI at run-time), a list of images, and PDB references. The Inner Core also contains supporting functions relating to undo and redo, memory management, and error handling routines. These functions are not part of the GIMP Libraries component because they appear to be specifically written for use in the GIMP. The Inner Core also manages an internal swap file. The swap file is used to store buffered blocks of image data, called tiles, when they are not in use. A tile is moved to RAM when it accessed by a function. The Inner Core source code is stored in the app/core directory.

In our conceptual architecture, the functions of the Inner Core were included within the Core component, however no explicit sub-component existed. During our concrete analysis, we felt that the architecture of the Core should be completely partitioned into separate sub-component since the functions of the Inner Core were separate from the other sub-components.

3.2 Concurrency

Like most GUI based applications, our concrete analysis prove that the GIMP is concurrent application. The GIMP exhibits concurrency in two forms: through even-driven processing in the GTK library and through the use of asynchronous plug-ins.

3.2.1 Event Driven Programming

As previously mentioned, the GIMP toolkit (GTK) is an integral component of the GIMP. The GIMP interacts with the GTK by registering event handling callbacks with the GTK API [10, Chapter 2]. The GTK is then responsible for interpreting GUI gestures such as: clicking a menu item, clicking an icon in a toolbar, dragging an image, etc. Each gesture is translated into an event by the GTK and the associated event handler callback is invoked within the GIMP. This style of application development (and the associated concurrency model) is known as Event Driving Programming [1]. Since many events maybe triggered and handled concurrently, the GIMP is a concurrent application [10, Chapter 16, 22].

3.2.2 Plug-in Concurrency

As mentioned above, plug-ins are separate processes that run concurrent to the GIMP. When a user selects a plug-in, to for example blur an image, the plug-in is invoked asynchronously. Our concrete analysis revealed that there is no mutual exclusion in the data model. As a result, running two or more plug-ins simultaneously on an image will cause unexpected and generally unwanted side effects. For example, performing blurring and rotating the same image object can have different results. Different pieces of the image may be altered in a different order each time the plug-ins are concurrently applied.

4 Conclusion

We have presented the software architecture of the GNU Image Manipulation Program (GIMP). To describe the architecture we constructed a conceptual architecture using a top-down approach. We then compare our conceptual architecture to the concrete architecture constructed using fact extraction software. Our analysis of the architectures showed that both architectures contain the same high-level components. However, our concrete architecture contains unexpected interactions between components as well as unexpected components with-in the GIMP Core.

References

[1] Event-driven programming. Available from: http://en.wikipedia.org/wiki/Event_driven_programming_language.

[2] Gimp reference manuals for the gimp 2.0 api. Available from: http://developer.gimp.org/api/2.0/index.html.

[3] The gimp toolkit. Available from: http://www.gtk.org.

[4] Gimp user faq. Available from: http://www.netads.com/~meo/gimp/faq-user.html.

[5] Gimp user group. Available from: http://gug.sunsite.dk/.

[6] Gimp.org ftp sever. Available from: ftp://ftp.gimp.org/pub/gimp/plugin-template/.

[7] Gimpshop dot net - info on gimpshop. Available from: http://www.gimpshop.net/.

[8] Macgimp - professional digital image software for mac os x. Available from: http://www.macgimp.org/.

[9] Xul - mozilla developer center. Available from: http://developer.mozilla.org/en/docs/XUL.

[10] Tony Gale, Ian Main, and The GTK Team. Gtk+ 2.0 tutorial. Available from: http://www.gtk.org/tutorial/index.html.

[11] David Neary. Gimp developer resources - how to write a gimp plug-in. Available from: http://developer.gimp.org/writing-a-plug-in/1/index.html.

[12] Sven Neumann and Michael Natterer. Rfc: The future of the gimp., 2000. Available from: http://www.mail-archive.com/gimp-developer@scam.xcf.berkeley.edu/msg03656.html.

[13] Tamer Rafla, Rafiou Oketokoun, Artur Wiklik, Michel Desmarais, and Pierre-N Robillard. Accommodating usability driven changes in existing software architecture. In *8th IASTED International Conference on Software Engineering and Applications*, MIT, Cambridge, USA, November 2004.

[14] Ingo Ruhnke. Script-fu tutorial. Available from: http://pingus.seul.org/~grumbel/gimp/script-fu/script-fu-tut.html.

[15] Anthony Senyard and Martin Michlmayr. How to have a successful free software project. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, pages 84–91, Busan, Korea, 2004. IEEE Computer Society.

[16] The GIMP Development Team. Gimp plug-in registry. Available from: http://registry.gimp.org.

[17] The GIMP Documentation Team. Gnu image manipulation program user manual. Available from: http://docs.gimp.org/en/.

[18] Micheal Terry. Jgimp. Available from: http://jgimp.sourceforge.net.

[19] Wikipedia. Gimp — wikipedia, the free encyclopedia, 2006. [Online; accessed 4-December-2006]. Available from: http://en.wikipedia.org/w/index.php?title=GIMP&oldid=91220880.