Fachhochschule für Technik
und Wirtschaft Berlin
University of Applied Sciences

Bachelor Thesis

Pipeline Based Image Editing with JAlbum

David Fichtmüller

July 2008

University of Applied Sciences (FHTW), Berlin
International Media and Computing

Supervisor:
Prof. Dr.-Ing. Kai Uwe Barthel
Prof. Thomas Bremer

Acknowledgment

I would like to thank David, Anna and Daniel for their thorough corrections and improvement suggestions.

Furthermore I would like to thank Prof. Barthel for accepting this thesis and for his advice and feedback.

Last but not least I want to thank Laura for her continuous support and patience.

Table of Content

1. Introduction

Abstract

Almost every kind of creative software allows the user to undo operations which did not lead to the desired result. Once an operation is undone it can be redone but only of no other operations are applied in between. At no point of the editing process can the parameters of an operation be adjusted after that operation was first applied and the user can only undo operations in the order in which they were applied. Wrong decisions which were made several steps ago are hard to correct because undoing them means loosing all the steps which were done after the wrong one.

The idea for this Bachelor thesis is to develop a concept for software which allows the user to go back to any stage of the editing process and change the parameters of any operation as well as the order in which those operations are applied. This gives the user much more flexibility in working creatively and it enables the user to easily correct mistakes later on. The concept will be explained in general so it can be used for all kinds of creative software. It will also be concretized with the focus on image editing software and certain characteristics or potential problems which could arise in that particular area are shown.

Also part of this thesis is a practical prototypic implementation of the developed concept based on the image editing functionalities of the photo publishing software JAlbum. The implementation also takes care of the special characteristics of the JAlbum software.

Motivation for this topic

During the internship which was part of my studies I worked for the Swedish company JAlbum. Their main product is a photo publishing software by the same name. My job was to implement image editing functionalities. My original concept of the image editing operations (in the JAlbum context called *Image Filters*) already included the idea that the user should be able to change the parameters of any applied operation at any time. This concept was rejected by the team because it was considered to complex for unexperienced users. But I was assured that this concept would be of great use for power users. So we decided to focus on a simple version of the filter handling for the first release and leave the option for the more complex concept for future versions. Since my internship was about to end I decided to work on this concept as my Bachelor Thesis.

2. Basics

Digital Image Editing

Digital Image Editing means to modify existing digital images by using computer software to achieve desired effects. Digital images can either come from digital image sources like digital photo cameras, from digitalization of analog images, like scanning classic photographs, or from other software for creating new digital images, like rendering 3D models. Many image editing software can also create new images “from scratch”.

Digital images can be classified into two basic kinds: raster graphics and vector graphics.

Raster graphics have predefined dimensions in number of pixels. Each pixel holds a particular color and all pixels in total reassemble a recognizable image. The quality of the image depends on the number of pixels it consists of and the range of possible colors. Most major image formats are raster graphics such as JPEG, GIF, PNG or BMP. Well known raster graphics editors include *Adobe Photoshop*, *The GIMP*, *Coral Photo Paint* and *Microsoft Paint*.

Vector graphics use vectors with relative coordinates to describe an image. Therefore the image is scalable without loss of quality. Examples of vector graphics editors are: *Adobe Illustrator*, *Inkscape* or *Coral Draw*.

In the context of this Bachelor Thesis each mentioning of “Image Editing” refers to Digital Image Editing. In this paper the term “image” or “images” will only relate to raster graphics, usually but not exclusively digital or digitalized photographs. Digital image creation will not be covered by this thesis.

Undo and Redo

Undo is a command used for reverting changes done in almost any kind of creative software. It allows users to remove the last changes done in the program step by step in the reverse order in which they were done. Redo is the opposite of undo and it allows the user to restore the changes which were previously reverted.

Usually those two commands are offered to the user as buttons where the undo command is displayed as an arrow facing to the left whereas the redo command is displayed as an arrow facing to the right.

It is a widely accepted paradigm to assign the keyboard shortcut Ctrl + Z (or Cmd +Z for Mac) to the Undo command and Ctrl + Y (or Cmd +Y) to the Redo command.

Java

Java is an open source programming language developed by Sun Microsystems. It is object oriented and can run on any major operating system or system architecture provided that there is a Java Virtual Machine installed.

Since JAlbum is written in Java the prototypic implementation of the concept developed as part of this thesis will also be programmed in Java. The code will be written in and compiled with the version Java SE JDK 1.5.0 also referred to as JDK 5.0. JAlbum uses the Java 2D technology to work on images. For the internal storage and handling of the images *BufferedImage* objects are used. The GUI of the application developed for this thesis will be done using Swing Components.

In the field of image editing Java is not wide spread. There are however a few implementations of image editing software in Java. One of the most notable applications would be the open source software ImageJ[1] which was designed for analyzing medical and biological images and which has an open API to allow developers to create their own plug ins.

What JAlbum does[2]

JAlbum Software

The main product of JAlbum is a software with the name “JAlbum”, also referred to as the “client”. It allows users to publish digital online photo albums. All the user has to do is to put the digital images in the software, press a button and the HTML code for the album is created. JAlbum takes care of the resizing of the images, for the preview images as well as for the final images. After the album is generated, the user can use the build-in FTP-client to upload the album either to their JAlbum Hosting account or to their own private webspace.

1 http://rsbweb.nih.gov/ij/
2 This chapter is based on the Internship Report from the internship at JAlbum by David Fichtmüller

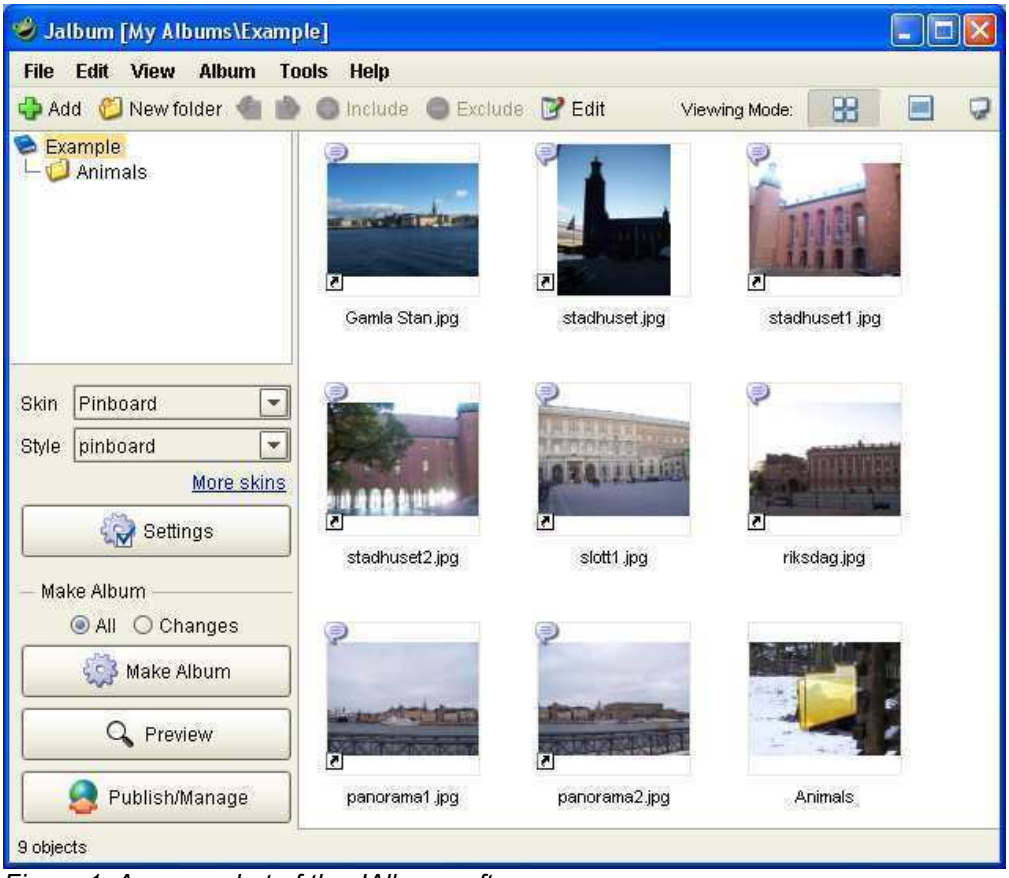

Figure 1: A screenshot of the JAlbum software

One of the major principles of JAlbum is the apparent contradiction “Easy to use – Endless customization”, meaning the software should be intuitively usable for users who have little experience with computers and no knowledge whatsoever about programming and web publishing but on the other hand the software should not limit the professional users in making their albums look exactly the way they want them to look. This principle is very important as the user base of JAlbum reaches from one of those extremes to the other one. So the main user interface is kept very simple and work flow oriented. The user can add, rearrange and add comments to images in an explorer like user interface which also supports sub folders and other media types, such as PDFs, movies and documents.

The biggest influence on the appearance of an album has the “Skin”, which is selected for creating the album with. A skin is basically a template which JAlbum uses to generate the final album. JAlbum comes with 7 preselected skins but over 130 more skins can be downloaded via the homepage. Skins use an open API and therefore everybody who knows a little bit about programming and web design can make their own skins or alter existing ones to fit their needs. Some of the skins can then be used with different style sheets or have their own settings tab to

further customize the look of the final album. A few skins also produce their albums as flash albums using Adobe Flash.

Figure 2: The album from Figure 1 generated with two different skins

JAlbum currently comes with a native language support for 31 languages. But most important JAlbum is freeware, without banners, ads, spyware or expiration date.

Hosting

Besides the software itself JAlbum also offers a hosting solution for the users. Every registered user gets 30 MB of free webspace as well as their own subdomain in the form of <username>.jalbum.net. If the user needs more space it is possible to upgrade to a paid account with up to 10 GB of storage.

Community

In February 2008 JAlbum released a new homepage with a community site, which gives its users a lot more options to share their albums with other users, comment on each others albums and photos and interact with each other.

How the JAlbum software works

The JAlbum Filter Concept

Album Filters

Before the Image Filters were introduced the only option to change an individual image was to rotate it around 90°. The only other way to edit images was by applying *Album Filters*. Album Filters change every image in the entire album in the same way. Applying Album Filters is only possible by adding a user variable to the album settings like "`Class=WatermarkFilter text="Copyright by David Fichtmueller"`". This was not only impractical but also vulnerable to input faults. On the other hand one of the advantages of the album filters was that they were all compiled classes in the subfolder */plugins* of the JAlbum installation directory and all implemented the interface *JAFilter*. So it was easy for other developers to make their own filters and add them to their albums.

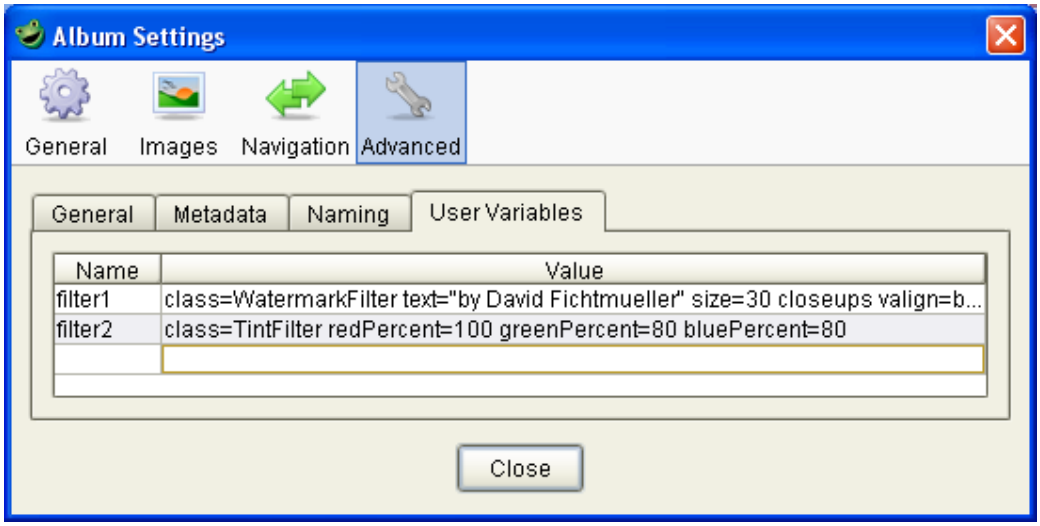

Figure 3: Example of 2 Album Filters being applied to all images of the album

Image Filters

While defining the requirements for the new Image Filters it became one of the objectives to have the filters independent from the main application so that other developers could write their own filters by just implementing a single interface and putting the compiled files in a specific folder. In order to make the Image Filters attractive for non professional users it was unavoidable to offer a simple way to apply the different filters via the GUI of JAlbum. The ability to undo/redo applied

filters was also important.

When the first version of JAlbum with Image Filter support was released[3] users were offered 12 different Image Filters which were: Cropping, Red Eye Removal, Levels (Color Adjustment), Gamma, Rotation, Gray Scale, Sepia, Sharpen, Blur, Flip, Invert and Pixelate. Seven other filters were in production and to be released later on. The filters themselves are structures in a complex inheritance hierarchy specifically designed for later reuse. The structure of this hierarchy itself is beyond the scope of this thesis.

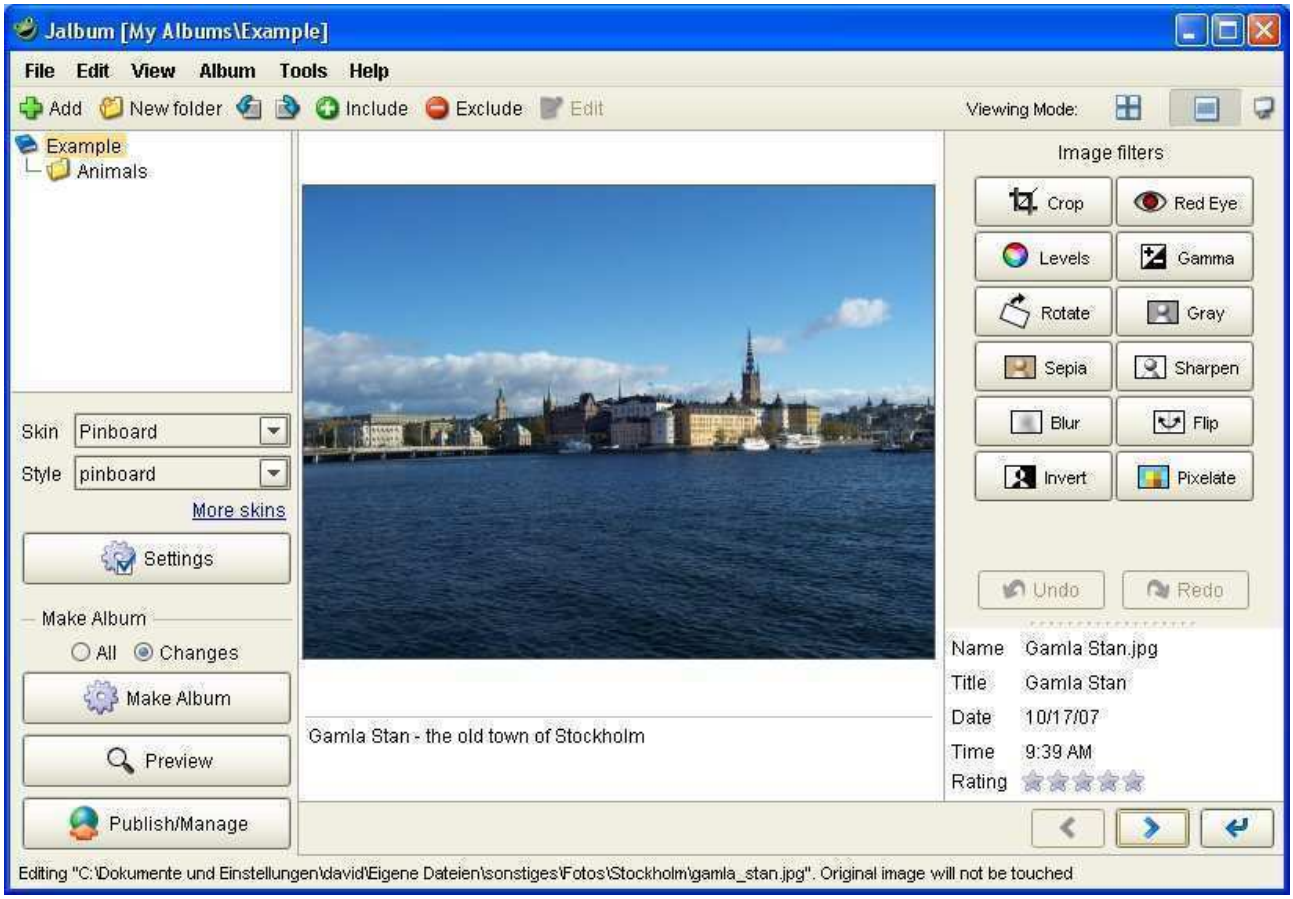

Figure 4: JAlbum 8.0 with the possible Image Filters

Upon start up the application looks in the folder */ext* in the JAlbum installation directory and searches for files with the name pattern **Plugin.java* which are then checked if they implement the interface *JAlbumImageFilter.* This also works for files in subfolders or in **.jar* files.

When an Image Filter is selected by the user to be applied to an image, that image gets handed to the filter. If there are other filters applied beforehand, the input image already has those filters applied to it. The user then sees the GUI of that filter which will use that image as a base for a

3 JAlbum 8.0 was released on 21. May 2008

preview of what the resulting image might look like with that filter applied to it. This filter preview can be different from the resulting image rendered by the filter. Other elements can be displayed on there as well to allow the users to set filter parameters by clicking and/or dragging and dropping the mouse over the image. Those elements are called *Control Elements*. A good example for those Control Elements would be resize control handles on resizable objects which are added by the filter. The preview image could show even more than just Control Elements. The Cropping Filter for instance shows the entire image as it is given as input and then draws a rectangle on it for the area to be cut out where as everything outside this rectangle has a semi transparent white layer over it so the selected area will stand out from the rest of the image. The rectangle has resize control handles, which will also not be visible on the final image. Figure 5 shows how the preview of a filter can differ from the actual output image of this filter. In this example the Control Elements of the Cropping Filter are visible as well as the parts of the image which will be discarded later on. In the GUI of every filter there is a button “Preview” displayed which will open a window with a preview of what the final image will look like when all filters are applied regularly. Here Control Elements and other content which is visible in the filter preview will not be displayed.

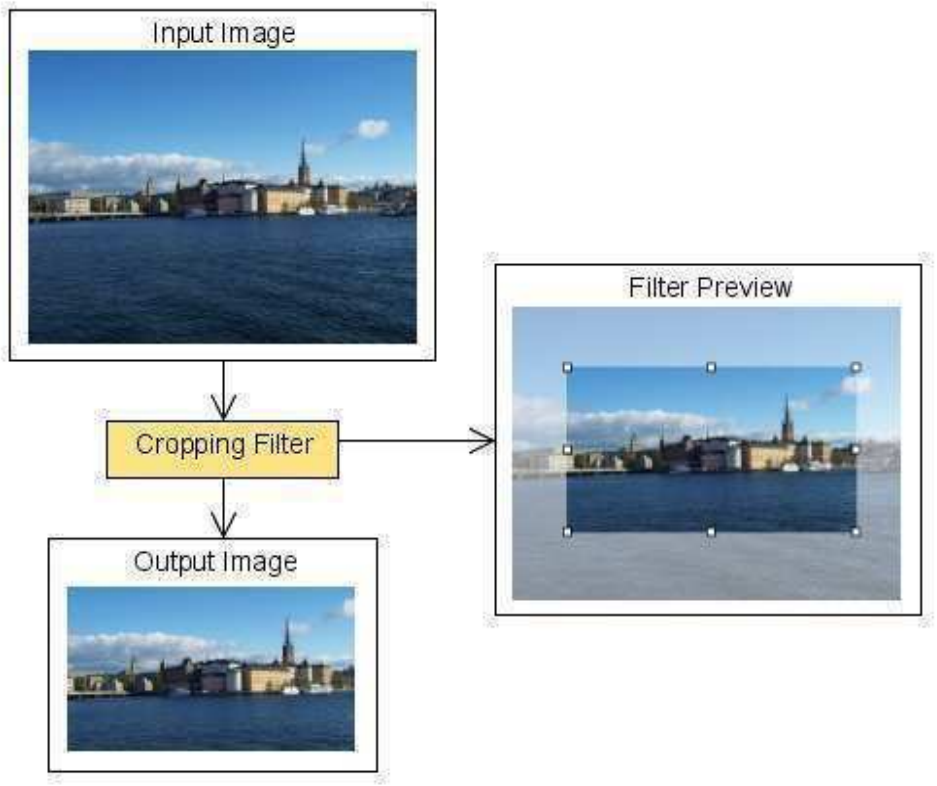

Figure 5: Cropping Filter with Input Image, Filter Preview Image and Output Image

Filters can optionally display a control panel where the user can change the settings for that filter. But such a control panel is not always relevant as some filters do not need any settings. They can either be applied or not. A good example for such a filter would be an Invert Filter. It makes no sense to set a value to what extend an image should be inverted or not. Either this filter is applied or not so no control panel is necessary.

The user has two different options of leaving the filter GUI. Either by clicking the button “Apply” if the resulting image is as expected or by clicking on the button “Cancel” if the filter should not be applied to the image. By clicking “Apply” the filter will be put in the list of applied filters, the final image is rendered to update the preview image and the GUI will return to the Standard View. There the user can select other filters to be applied. It is also possible to add a new instance of an already applied filter.

To control all the Image Filters applied to a particular image an object called FilterManager is used. The FilterManager is the only connection between the JAlbum core and the individual filters. It takes care of rendering all filters for the preview as well as for the final image generation. The filters are saved in a list in the order in which they were applied. In order to store the filters permanently this entire list is exported and later imported again. This way JAlbum only saves the filters, i.e. only the changes applied to each image, and not the changed image itself. This is important as it is a main principle of JAlbum never to change the original files unless the user specifically wants that. The filters are exported as XML files. Since the GUI elements of the filters are not used anymore after the filters are applied, they are discarded before exporting. Only the settings of the filters are saved which is enough to apply the same filters later again with the same result.

The JAlbum Album Generation Process

When the user presses the “Make Album” button, JAlbum first generates all the HTML pages based on the templates provided by the skin. Then the images are rendered. The album uses two basic image sizes, *Thumbnail* for the small images on the so called *Index Pages* and *Closeup* for the images on the *Slide Pages*. The maximum sizes of both image types can be set via the album settings page. This means that most images are scaled down twice for the two types of images if they are larger than the bigger image type, usually *Closeup*. When applying Album Filters the user can define via additional variables at what point the filters should be applied, either before the image is scaled (*prescale*) or after the scaling (*postscale*). For Album Filters it is also possible to

specify if the filter should be applied to either only *Thumbnails*, only *Closeups* or both. Depending on those settings it might require forking the process for rendering the thumbnails and closeup images individually.

With Image Filters the user currently has no choice at what stage the filters are applied and if they should be applied to only one of the two image types. Each filter specifies if it can be applied prescale or postscale or both. Typically prescale filters are those which change the size of the image such as Cropping, whereas postscale filters are those which change the image depending on single pixels and where the effect would be lost if the image was to be scaled afterwards, such as neighborhood operations like Blur. When the image is rendered, the FilterManager goes through the list of applied filters in the order in which they were added by the user and selects all prescale filters and applies them to the image. It then scales the image and applies all postscale filters by going through the list again. Filters which can be applied both prescale and postscale are treated as prescale filters. This means that the filters are not necessarily applied in the same order in which the user selects them. When the user selects a new filter, the FilterManager already looks where this filter will be applied and gives the filter an image with all filters applied to it, which will also be applied before that specific filter in the final rendering process. The filter then uses this image as base for a preview of its own calculation within its GUI. This could confuse the user if some of the filters selected earlier might not show up in this preview. For example: A user first applies a blur filter on its image and then a cropping filter. Because in the final rendering process the cropping will be applied before scaling whereas the blur will be applied afterwards, the blur filter is not applied to the image used as an input for the preview of the cropping filter. This example is demonstrated in Figure 6. When the user selects an area to be cropped and clicks on “Apply” then the preview for the entire image will be updated and the user will see the cropped and blurred image. While being in the preview of the cropping filter the FilterManager indicates that there will be filters applied afterwards by specially marking the preview button. When clicking on that button the user can see a preview of the final image rendered as if the filter was applied at that stage.

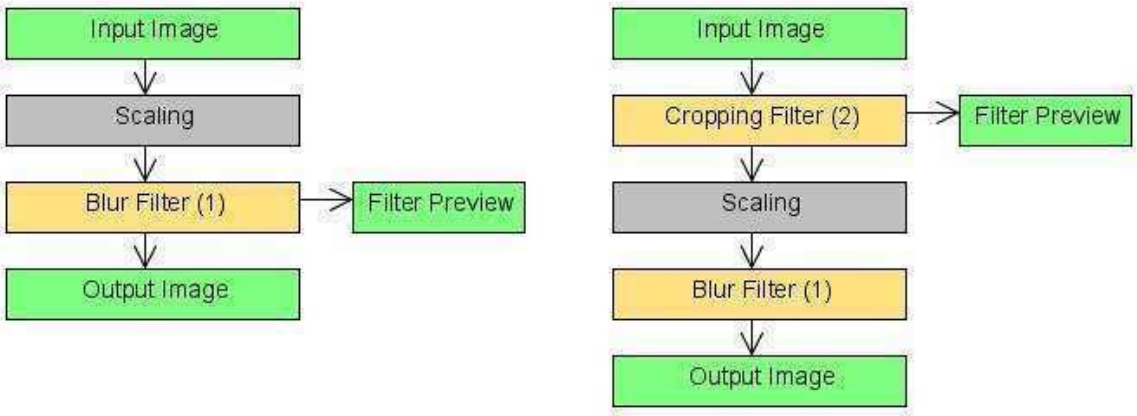

Figure 6: Problem with the rendering order in the Filter Preview

When there are Image Filters as well as Album Filters selected for an image first the prescale Album Filters and then the prescale Image Filters are applied, then comes the scaling followed by the postscale Image Filters. Finally the postscale Album Filters are rendered to the image.

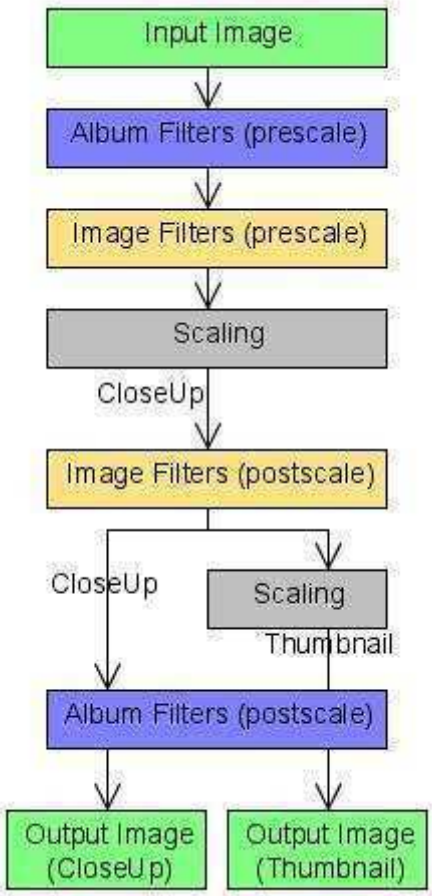

Figure 7: Rendering Order of Album Filters and Image Filters

Caching

In order to avoid unnecessary calculations the image is cached at several stages during the generation process by the *FilterManager*. There are four cached versions of the image in total. At first the input image as handed by the application is stored as *InputCache*. Then all prescale filters are applied. The resulting image is stored as *PrescaleCache*. The image is then scaled down to the final size and cached as *PostScaleCache.* After that all postscale filters are applied. The final image is than cached as *OutputCache* before it is handed back to the application to be displayed on the GUI. Those four caches clearly speed up the process of adding new filters or undoing/redoing existing filters. When a new filter is added as postscale filter then the image from the *OutputCache* is taken as the input for that filter. So only this new filter has to be rendered and the resulting image will be the new *OutputCache*.

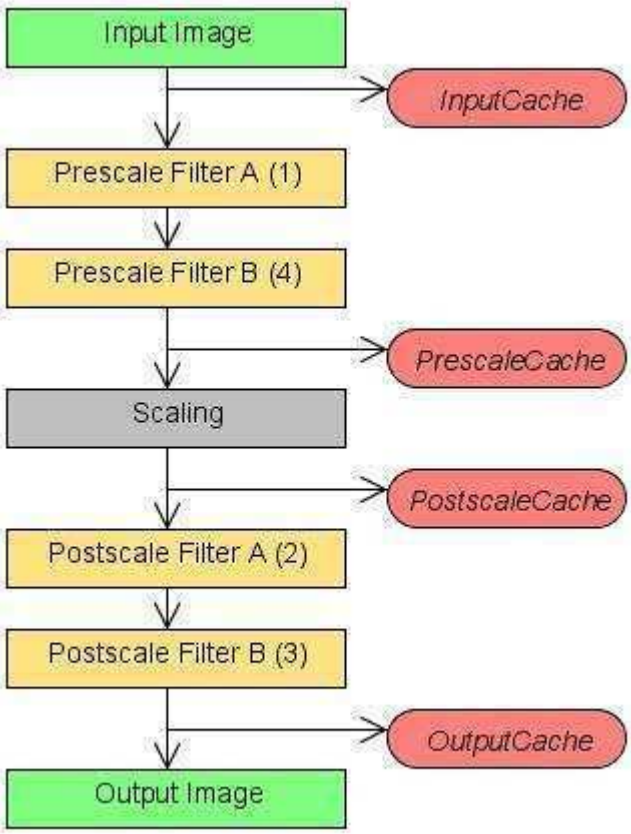

Figure 8: The four caches and the points at which they are stored

When a new filter is applied as prescale filter then the new filters takes the image from the *PrescaleCache* as input. The resulting image will be stored as the new *PrescaleCache* before it is

scaled and all the postscale filters are applied. This also updates the *PostscaleCache* as well as the *OutputCache*.

When the last postscale filter is undone then the *PostscaleCache* is taken and all the postscale filters are rendered again. The same is done likewise if the last prescale filter is undone with the *InputCache* only that afterwards the image is scaled and the prescale filters are applied like it is done after adding a new prescale filter.

Redoing an undone filter is very similar to adding a new filter and therefore the same caching mechanisms apply.

3. Concept

Undo/Redo Concepts

Most modern software where the user can create or alter some kind of content, support the option of undoing previously done operations. Any operation or "step" the user does is saved in some kind of list, hereafter referred to as "history". Undone steps are then stored again in some other kind of list so that the user can redo undone steps. How many steps can be undone/redone and when those lists are reset depends very much on the undo/redo concept used and its specific implementation in the software.

Linear Undo

Almost all implementations of the Undo/Redo Concept use the linear undo. The user can only undo the steps in the reversed order in which they were done. This makes the history to a clear FIFO stack. Steps that are undone can be redone if the user did not change the undone version in which case all the possible redo steps would be lost. Depending on the particular software the number of possible undo-steps is sometimes limited to save memory and achieve better performance. The history and the redo list are usually not saved, so when closing and opening the file which was edited, it is rarely possible to undo/redo steps done before closing. Some programs even discard those lists when just saving the file without closing the application.

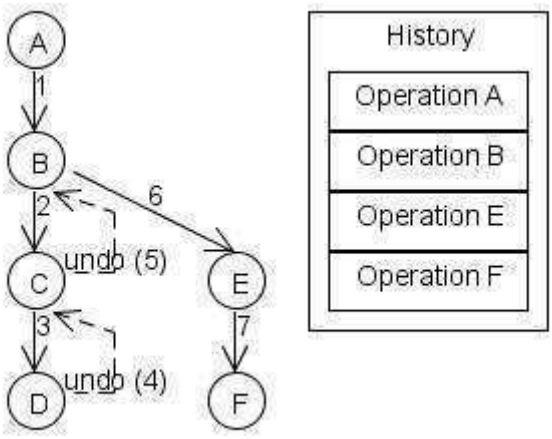

Figure 9: Linear Undo with visible History

In Figure 9 the user undoes the operations C and D and then applies Operation E and F. Before step 6 it was possible to redo steps C and D but after Operation E was applied the redo list was reset and therefore the state of the file after Operation C or D were applied can not be restored anymore.

Non Linear Undo

A more advanced approach to the undo/redo concept is the *non-linear undo*. Here the user can undo any single operation in the history without also having to undo the operations done afterwards. This concept however is rarely implemented.

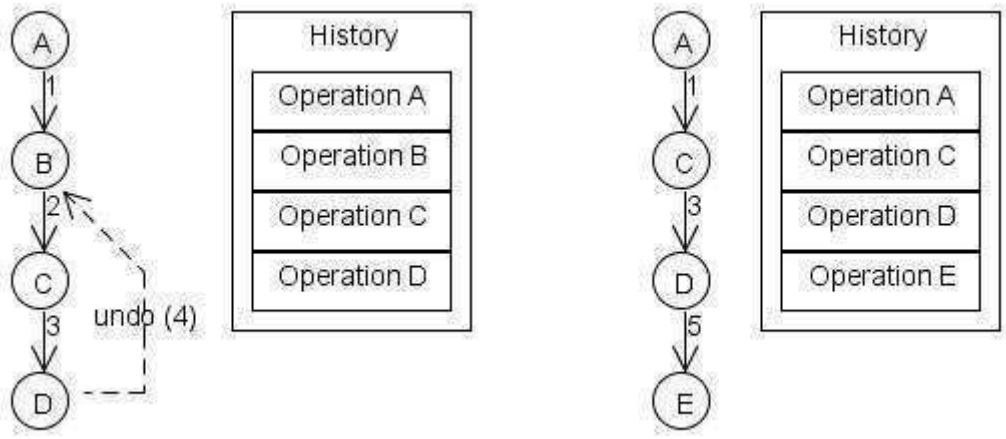

Figure 10: Non Linear Undo

In Figure 10 the user undoes Operation B as the forth editing step and afterwards applies Operation E. So Operation B is undone but the Operations C and D are still applied before Operation E is added.

One problem of this concept is the interdependence of operations. So if in Figure 10 Operation B were the creation of a new object, Operation C the positioning of this object and Operation D the coloring of this object then undoing Operation B would make the Operations C and D redundant.

Branching Undo

A branching undo allows the user to go back to any stage the document was previously in even if the operations which led to this stage were undone and other operations were applied afterwards. So instead of discarding the steps in the redo list after applying a new operation, a new branch is created. The user gets some visual representation of the resulting “undo tree” so it is possible to go back to any of the previous stages. This concept is sometimes also referred to as “Forking Undo”

Adobe Photoshop has such option build in but there it is called “Non-Linear History” even though it is not a non linear history as described above. This option is disabled by default. If the option is enabled[4], the user sees all the previous steps in the history window to which the image can be reverted. If the user clicks on one of those previous steps and modifies the image at that stage, the undone steps are not deleted as in the regular mode instead they are kept in the history and the new steps are added at the end of the list. So the user can go back to any branched version which would have been delete in the normal mode.

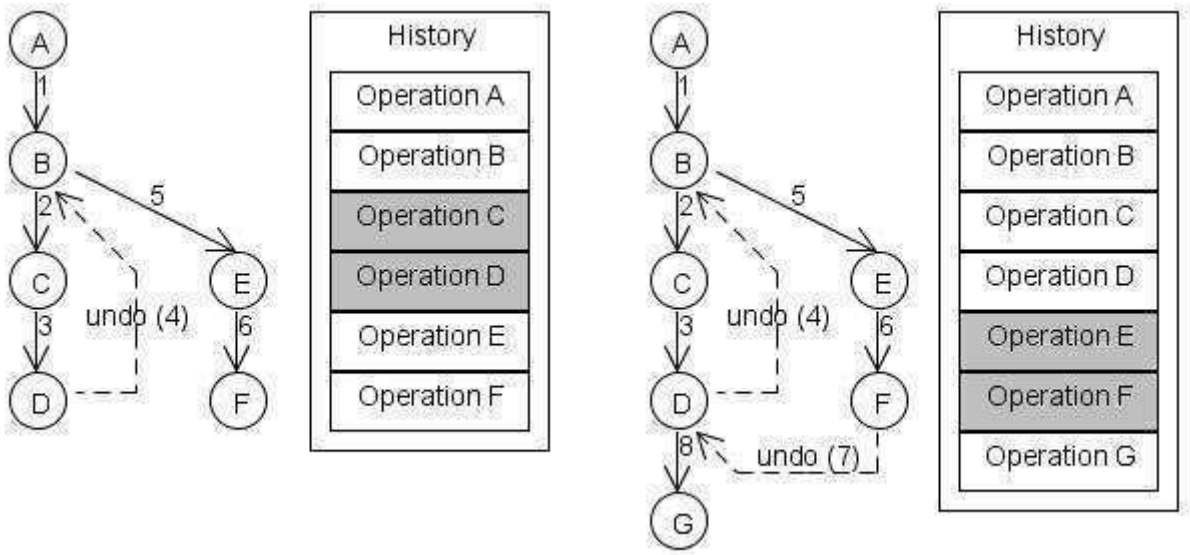

Figure 11: Branching Undo in Adobe Photoshop

Figure 11 shows how such a branching will be displayed in Photoshop's history palette. In this example the user first applied the operations A, B, C and D before clicking on operation B in the history causing the operations C and D to be undone and B to become the last active one. After that operations E and F were applied. Unlike in the regular history mode the operations C and D are still kept but are not active, so the only operations which are done are A, B, E and F. That is the stage as it is displayed in the first part of the figure. The user then realized that the operations C and D were actually better than E and F . So by clicking on operation D in the history the user went back to the previous branch causing C and D to be reactivated and E and F to be disabled. Operation G which was done afterwards now follows operation D in the logical order.

In Photoshop the currently not active operations however are not displayed differently from the active operations unlike it is indicated in this figure. This can cause some confusion as the user can

4 To enabled this option one must open the “History Palette”, click in the upper right corner on the small double arrow pointing down and select the menu entry “History Options”. There the option “Allow Non-Linear History” must be enabled.

not see which operations are currently applied and which are not.

Multi Level Undo

Another approach to a more complex undo/redo model is to differentiate between a local and global undo. In a software where the user can edit multiple objects each of the objects has its own local list of steps which can be undone. But there is also a global list of all the steps the user did regardless of which object was edited. So the user can also undo steps from this list. This concept is also called "Object Based Undo". A theoretical approach for combining the Object Based Undo Model with a non-linear undo is described in [3] but it is very rarely used in practical software.

Multi User Undo

Very similar to the Multi Level Undo is the concept used for multi user undo. When several users can work with the same software at the same time each user has its own undo list as well as a global undo list for all the changes all users did.

The Pipeline Concept

The Pipeline Concept goes beyond the non linear undo. As in the non linear undo concept the user can undo any step previously taken, regardless of the order in which the steps have been done. Additionally it is also possible to change the order in which the steps are done as well as adjust the parameters for each of those operations. So the user can go through the operations stored in what was previously the history list and modify each operation until it fully fits their needs.

The purpose of this concept is to give the user more control over the previously done operations. This makes the creative process more fault tolerant and encourages the user to try more variations of operations and use those which are best suitable for the desired result.

Potential Use in Software

This concept is not suitable for all kinds of software which use regular undo/redo. One important factor is the size of each operation or *operation graining*. If the operations are to small and atomic then there are only few or even no settings for those attributes. So instead of adjusting the settings or attributes for each operations the user can in most cases only decide if that operation should be applied or not which causes the concepts to loose one of the main advantages over the regular non linear undo concept. But on the other hand if the graining gets to big, the amount of settings the user

can change on a single operation will become to complex for the user to easily adjust the operation to get the desired result. It will be harder to get the same combination of settings again if the user wishes to go back to a previously selected setting of this particular operation. This will enforce the need for a meta undo (see chapter “Meta Undo/Redo” on page 23).

An example for too small operation graining would be a simple text editor. Here it makes no sense to have each key stroke as a single operation. Those operations would have no sensible options. To have the option to change a typed letter into another letter would be useless as it would require more time instead of just hitting backspace and typing the correct letter. In that case it also is overshot to implement it as a non linear undo because undoing the typing of a new character would require more effort that just deleting the character again. But of course a non linear undo might come in handy if the user deleted some content previously, wrote some more and then realizes that the deleted content was actually important. So the user could then undo the deleting without loosing the text typed afterwards.

Another key factor which will show if a software is suitable for a pipeline concept is the *order dependency* meaning that the results will vary depending on the order in which the operations are applied. This might vary even within one domain of application and depend on the specific operations which are compared to each other. In an image editing software it makes not difference if an image will be cropped first and then reduced to gray scale colors only or if it will be reduced to gray scale before cropping. But if instead of cropping the user would draw a colored line on the image, it would make a difference as this line would either be displayed in color or in gray scale depending on the order of those two operations.

So software with more order depending operations will benefit more from implementing the pipeline concept than those software with less order dependency.

Target User Group

Because of its wide spread in software most users are aware of the linear undo/redo concept. This however causes people to assume it is the only way to handle undo. The fact that even a software like Photoshop which is focused on advanced users, has their branching undo[5] option disabled by default shows that alternative undo concepts are not even used widely among professional users.

So for the Pipeline Concept the target user group will be a small group of advanced users because

5 As mentioned in the chapter “Branching Undo” on page 17, this option is called “Non-Linear History” within Adobe Photoshop

the concept as such will be too complex for regular users[6]. Ironically regular users would particularly benefit from this concept as it allows for mistakes to be easier corrected even if they were done several steps ago. With a clear visual representation of the pipeline and its containing operations and a user friendly and easy to use GUI even regular users might be able to make use of the advantages of such a concept without necessarily understanding that the list of operations they are working on is basically the list of operations they would only be able to undo in other software.

Programming Requirements

There are several ways of implementing the regular undo concept. An object oriented approach is described with the *Command* design pattern [2]. Each operation inherits from the same object, here the *Command* object, and has the methods `execute()` and `unexecute()`. The concrete commands are stored in a list in the order in which they are applied. When the last command is undone its unexecute-method is called and it would be removed from the list. This design pattern would also work without the unexecute method. If the last command was undone, it would be removed from the list and the program would go back to a certain saved previous point and execute all the commands again which follow. This requires more execution time but it is useful if it can not be guaranteed that the unexecute method will not produce the exact same state as it was before calling the execute command, like when rounding errors can occur.

The implementation of the pipeline concept is very similar to the implementation of the regular undo. Each operation the user can apply needs to be represented as an object and all those operation objects should inherit from the same parent object. Whether implementing the undo by un-executing the undone operation or by re-executing all remaining operations from some saved point depends on the domain of application. The only difference to the *Command* design pattern is that the operations now also need a method for setting the parameters of this operation. As it is unknown to the super class what and how many parameters the inheriting class might need, this method should be as general as possible. But on the other hand the user still needs to know what parameters each of the operations has and there should be a way to set those parameters individually.

Global States

Another issue that needs to be looked at a little closer is the assignment of global states. Global states in this context refer to any setting which is done in the software that has an effect of how the following operations are executed. It is one way of communication between different operations

6 “regular users” in this case refers to non professional users of that particular software which use it occasionally and mostly for private purposes.

without them being aware of each other. Examples for such global states would be the highlighting of text in a text editor which then can be formated, the selection of a foreground color which will then be used as the color for all afterwards created content or the masking of parts of an image in an image editing software. The selection itself could already be an operation, especially when special tools are used to do this selection, like the lasso tool for masking parts of the image. But the state itself could also be an additional parameter for the following operations. The operation of a box drawn in a drawing tool could have the foreground color as parameter. The option to implement the global state as a parameter is only useful if the parameter is of low complexity. So letting a filling tool have the selected part of an image as an option will make adjusting that parameter too complex.

Even a combination of both options, having the selection as an operation as well as a parameter of the following operations, is possible. For instance: when the user selects red as new foreground color and then draws a rectangle and a circle, both objects will be displayed in red (Figure 12a). The user can now go back to the color selection and change the color to green, causing both objects to turn green (12b). When selecting the Circle-Creation operation the color green is now the color parameter since it reflects the current foreground color. Now the user can change that color to blue thereby defining the color of the circle locally and making it independent from the global foreground color (12c). So when the user then goes back to the selection of the global foreground color and changes this color to yellow only the rectangle will turn yellow whereas the circle will remain in blue (12d).

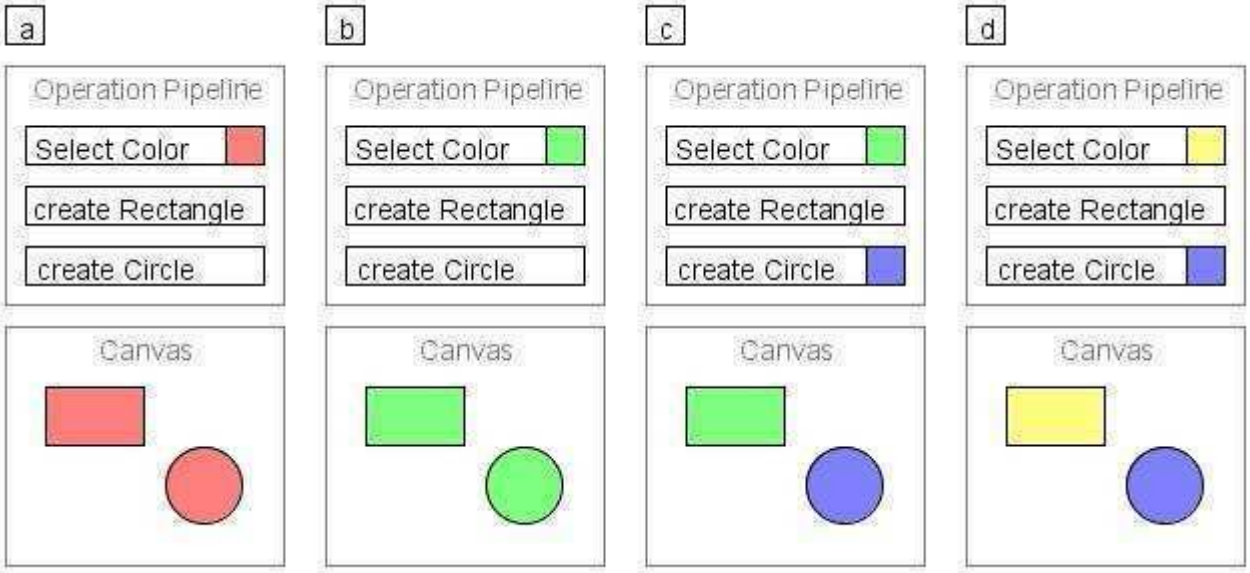

Figure 12: Example of Global States

There is no “always right” answer on how to implement the selection of global states. It depends on the complexity of the states as well as their usage by the following operations. In any way, it should always be obvious to the user why certain operations will or will not change when the global states are changed. So this might require some weighting between simplicity and functionality.

Meta Undo/Redo

With the introduction of the Pipeline Concept the regular Undo/Redo becomes redundant. Now the user can go through the list of applied operations which would formerly have been the history list and modify any operation at any stage. So the user does not navigate linearly through the list of operations as it used to be with the basic Undo/Redo model. This leads to the concept of a Meta Undo/Redo which represent the steps and modifications the user did to the pipeline of operations in a chronological order. This allows the user to go back to any previous stage without having to set the parameters as they were set previously. This Meta Undo/Redo could be implemented as a regular Undo/Redo or with one of the more advanced concepts mentioned previously. Despite the fact that it is possible to implement such a Meta Undo/Redo it still is questionable if such an approach would be adequate or if it just overshoots the target and confuses the user. This decision of cause depends on the specific implementation.

Implementing Meta Undo/Redo requires some kind of object representation of the different steps the user can do while working on the different operations in the pipeline. This could again be similar to the *Command* design pattern (see chapter “Programming Requirements” on page 21). It is enough to implement the Meta Undo itself as a regular linear and therefore the regular *Command* pattern can be used. Maybe an easier approach would be a simple versioning system for each operation. Whenever an operation is changed the old version is stored in a list of previous versions. So when the user decides to meta undo an adjustment of an operation this adjusted operation is replaced with the latest version of the list of previous versions.

Figure 13 demonstrates how such a Meta History would look like in an image editing software which uses the pipeline. In this example a user takes a digital image and first crops the image and adjusts the color by giving the red channel a little more strength. After that the user realizes that the image was cropped a bit too much and changes the settings of the cropping to show a bigger part of the original image than in the first cropping. As the next step the user sharpens the image and finally tweaks the color adjustment again to lower the strength of the blue channel. In the figure, the solid lines represent the rendering process of the final image whereas the dashed lines represent the

steps the user took. The gray boxes are older versions of operations and the current versions of those operations are more to the right and in orange. In the Meta History we see the steps of the user in chronological order. So if the user is not satisfied with the result of last modification of the color adjustment this step can be reversed by “meta-undoing” it. The image would then be rendered with the first version of the Color Adjustment operation.

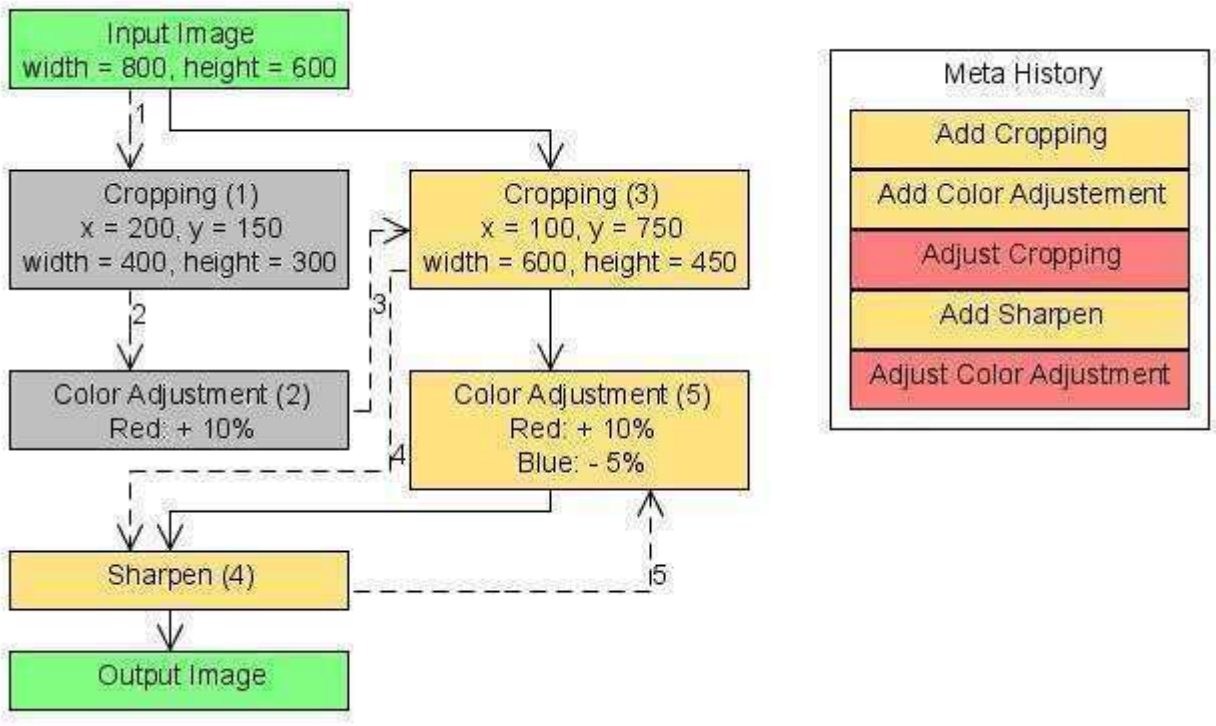

Figure 13: Pipeline Editing steps with Meta Undo History

Limitations of the Concept

The Pipeline Concept is a relatively abstract concept which could be applied to almost every kind of software which also uses undo and redo functionalities. That is why problems can arise when trying to implement this concept to a particular branch of software with its own uniquenesses. Since this thesis is focused on image editing software only problems from this domain are discussed in the following.

To many of the problems there is no single correct solution. Instead they depend on the context in which they should be applied as well as soft factors like user experience, general work flow or required programming effort.

Pixel Specific Editing

Some tools in image editing software work in a way so that the user can select parts of the image via mouse input. A simple example would be a pencil tool with which the user can draw a line on the image. Most software handle such operations as a single step from the pressing of the mouse until it is released. It is obvious how such an operation could be disabled or removed as well as moved up or down within the list of applied operations. But it is not clear how the settings for that operation should be changeable. The easiest approach would be to just let the user disable/remove this step and not offer any way of changing the line. Or the line as a such could be moved on top of the image it is drawn on. A more complex solution would be to store the line as vector based line which is rendered on the raster image. In that case the user could move the nodes of the line as well as their deflection. It is also possible to introduce another undo concept at this point and make the line gradually undoable. This means that the user has some kind of slider where it is possible to undo that line in the same small steps in which it was drawn, more or less pixel by pixel. An example implementation of a gradual undo can be found via [1].

Similar questions arise from other pixel specific editing functions such as the “Clone Stamp”, brushes or masking tools.

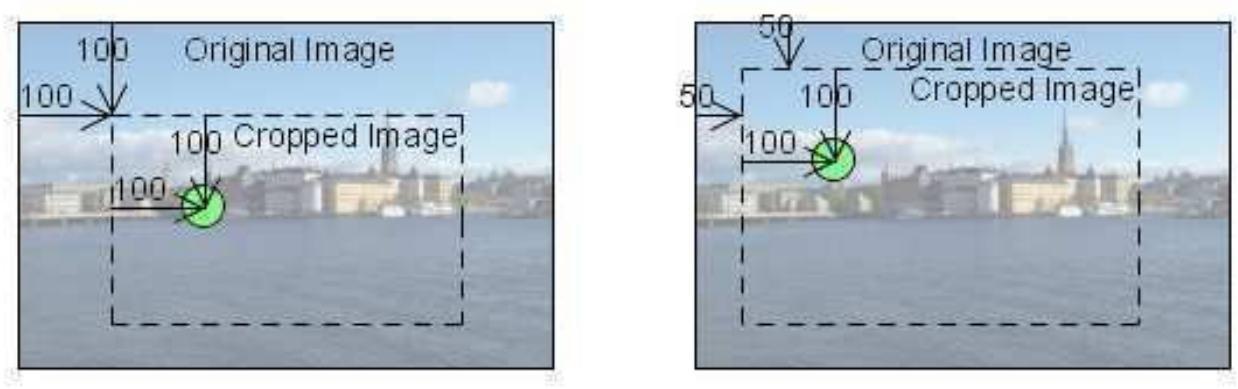

Figure 14: Example for problems with pixel specific editing

Another problem about the pixel specific editing is the fact that many operations are not pixel safe, meaning that is not guaranteed that a certain point of the image will always be at certain coordinates when a specific operation is applied. For example: The user has an image and crops it so that the image starts with its upper left corner where the coordinates x=100,y=100 of the previous image were (Figure 14). Then the user does some retouching close to the coordinates (100,100) of the cropped image. After this the user realizes that the cropping was too much, goes back to the

cropping operation and only crops off 50 pixels from the original width and height instead of 100. This leads to a different input image for the retouching operation and therefore the coordinates of the retouched area are now wrong as can be seen in the figure. The green circle (representing the retouched area) is now at a different part of the image.

There is no general solution to this problem. One approach would be to save coordinates always relative to the original image no matter how they are translated in between. But this means that each operation has to be aware of all changes other operations can apply to the image size or pixel positioning. Implementing new operations requires changes to all other operations as well. So this approach is only practical for small fixed sets of possible operations.

Layer Based Software

Many image editing software such as Photoshop or GIMP use layers to compose the final image. Integrating the pipeline based concept into a software which uses layers raised the question how to handle the different operations in the pipeline for the individual layers. One way of handling this is having a multi level undo (see chapter “Multi Level Undo” on page 19) where each layer would have a pipeline of operations applied to it. There would also be a global pipeline with all the operations for all layers in it. So far this is just a regular multi level undo regardless of whether it is connected combined with a linear undo, a non linear undo or even a pipeline concept. But with the pipeline concept it is additionally possible to have the the layer to which an operation belongs as a parameter of this operation in the global pipeline and thus this operations could easily be assigned to another layer. In an even more flexible approach it would be possible to assign an operation not only to one but several layers at the same time. This kind of pipeline option however is beyond the range of this paper.

Naming of the Pipeline Concept

The name Pipeline Concept is an allusion to the pipelines in the various Unix operating systems. When working in Unix via the command shell or with scripting, the pipe operator | (also referred to as “vertical bar”) can be used to channel the output of one program to be the input of the next program. Each of the programs can be set via specific parameters individually and they are not aware of each other just like the operations in the “Pipeline Concept”. Since this concept could also be explained as an extension of the “Non Linear Undo Concept” it may also be referred to as “Pipeline Undo Concept”.

4. Analysis

JAlbum Image Filters

The prototypic implementation of the Pipeline Concept is not done from scratch but based on the image editing functionalities build for JAlbum 8.0. So it is important to comprehend the existing code and its underlaying concept first before understanding the implementation done as part of this thesis. It is also important to demonstrate what part of the programming was done as part of this thesis and what part already existed prior to this work.

Methods of JAlbumImageFilter Interface

As already mentioned all Image Filters implement the interface *JAlbumImageFilter*. While it would be beyond the reach of this paper to explain the various filters in detail, it is important though to understand how the filters work in general. The interface specifies certain methods each filter has to implement. Most of those methods are just for getting basic information about this filter which will later be displayed in the GUI of JAlbum. That basic information is: the name of the filter, a short name, its author, the version, an icon, a description of what the filter does, optional help information as well as an optional text "other" where the filter author can add any other information about the filter which is not covered by the rest of the options, like thank you notes etc. Also part of the basic information is if a filter is a prescale filter and/or a postscale filter. A filter can either be only one of the two options, both, or even none. In the last case the filter is not applicable to an image. This might be useful if the filter itself is only there to be inherited by other filters which have common properties.

Besides the basic information the interface also includes methods used for instantiation. A filter is instantiated by a class called *FilterManager* which will be explained in more detail on page 35. This *FilterManager* is the only connection between the filter and the rest of the application. It also bundles all the communication from and to the filter. When a filter is instantiated, the *FilterManager* registers itself with the filter using the method `setFilterManager(FilterManager)`. Then it hands the filter the image to which the filter is applied. This image might already be processed by other filters but that does not matter to the filter. After the image is set, the `init()` method of the filter is called to initialize objects which can not be initialized in the constructor, like parts of the filter which rely on the input image already being

given at the point of the initialization. Initializing those objects in the constructor can cause *NullPointerExceptions*. Also when this interface was designed it was done with possible future extensions of the concept in mind. A possible future implementation of the pipeline based image editing has already been discussed at that point. The implementation of the pipeline concept will show that an instantiation of a filter and its initialization will be even more independent from each other as it can be seen later on.

After a filter is initialized the *FilterManager* calls the method `getControls()` which will return the control panel of that filter and is an instance of a *JPanel*. This control panel is then handed to the GUI to be displayed so that the user can use it to set the parameters of the filer. If the filter does not have a control panel `null` is returned and no control panel is displayed.

When the image is to be rendered, either in the final image rendering process or to update the preview of the image, the *FilterManager* calls the filter using the method `renderImage(BufferedImage)`. This method will also return a *BufferedImage* which is the image with the filter applied to it.

When the filters applied to an image are saved, the entire list of filters is exported using the Java *XMLEncoder*. Before exporting the filters, the *FilterManager* calls the `dispose()` method of the filters to discard any objects which should not or can not be exported, like *BufferedImages*.

The *JAlbumImageFilter* interface also extends the interfaces *Serializable, KeyListener, MouseListener, MouseMotionListener and MouseWheelListener*.

Restraints on the Image Filters

Because the *XMLEncoder* is used to export the filters for permanent storage, some restrictions apply to the development of Image Filters. The *XMLEncoder* exports a class by creating a new instance of this class, going through all public standard getters and comparing the values retrieved by both classes. If those values are different, the value of the current class is exported. An example of an exported filter list can be found in the Appendix under “FilterList class parsed as XML”. When importing a class later again using the *XMLDecoder* it creates a new instance of the class and calls the setters for each of the stored values. The consequence for the classes to be exported is that each value which is supposed to be exported needs a public getter and setter even if this value should be hidden from outside classes. Both getter and setter also need be be default getters and setter, so no additional code like checking for the correct values is allowed.

```
<type> value;

public <type> getValue(){
      return value;
}

public void setValue(<type> value){
      this.value = value;
}
```

Figure 15: Code Example of default Getter and Setter

The *XMLEncoder* also has problems exporting inner classes, so no inner classes can be used in the filters and also no classes having inner classes can be used by the filter if they should be exported.

Test Environment for Developing Image Filters

During the development of the *FilterManager* it turned out to be a good approach to keep the software as modular as possible. Therefore the interface (*FilterManagerFriend*) was created which was to be implemented by some part of the JAlbum software. When the *FilterManager* is instantiated it is handed a reference to the class which has that interface implemented. This is the only line of communication between the *FilterManager* as well as all the filters and the rest of the JAlbum software.

In order to have the development of the *FilterManager* independent from the development of the JAlbum core a test environment with the name *FilterManagerTester* was created which implemented the *JAlbumFilterManagerFriend* interface and provided a simple GUI. This environment did all the tasks the JAlbum core was supposed to do for the *FilterManager* but it had no album creation abilities. It also provided a graphical framework for the filters, the *FilterManager* and its related components. Using this approach the *FilterManager* could be developed and tested without being a part of the actual JAlbum software. The test environment could be compiled and started as a stand alone program which contained all the image filters. At the end, when the *FilterManager* was integrated in the JAlbum core, some class just had to implement the interface *JAlbumFilterManagerFriend.* Apart from some smaller adjustments to both, the interface and the *FilterManager*, this concept worked quite well and proved to be practical.

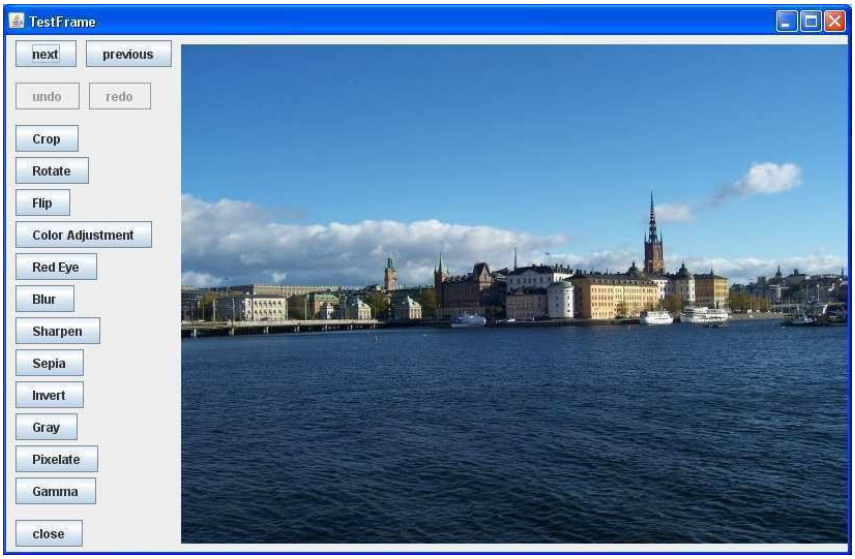

Figure 16: Screenshot of FilterManagerTester in the Standard View

Figure 16 shows a screenshot of the *FilterManagerTester* with a demo image. On the left side are the buttons for adding various filters to the image. This state of the program is called the "Standard View". Figure 17 shows the *FilterManagerTester* with an active Cropping Filter. This view is referred to as the "Filter View". In the Filter View the buttons for selecting individual filters are gone. Instead the control panel of the filter is now visible to set the filter parameters.

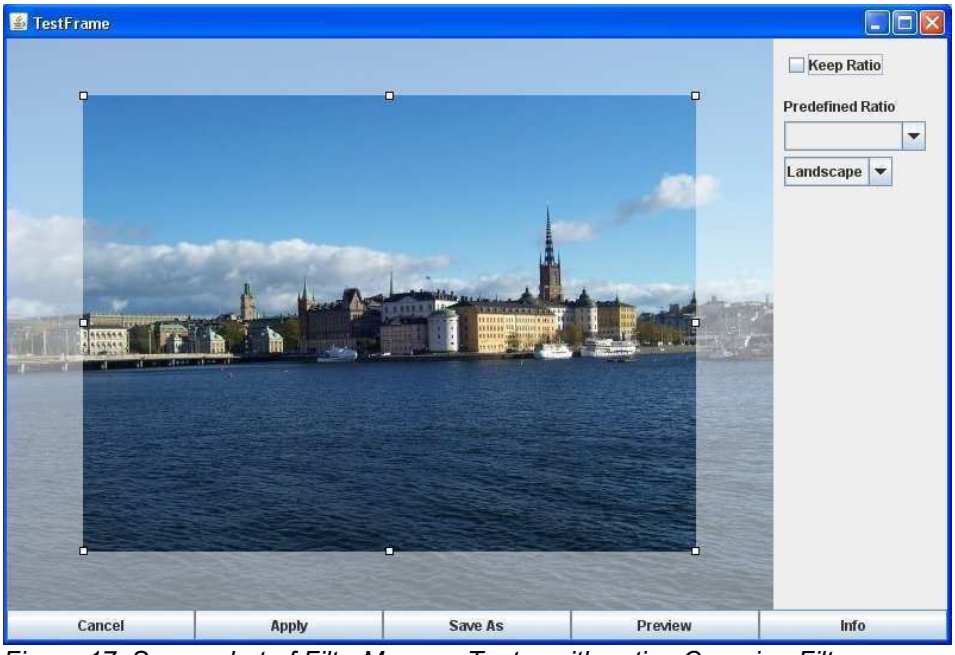

Figure 17: Screenshot of FilterManagerTester with active Cropping Filter

Running the Test Environment

The program FilterManagerTester can be found on the CD (see “Content of CD” on page 64). It can be started by running the file FilterManagerTester.jar. The images used are located in the folder /images within the folder of the program. The XML files for saving the filters would usually be saved in this folder as well. But since it is on a CD the program will not be able to save the files and just exit without saving. In order to be able to save the files one has to copy the program as well as the folder to a writable drive and start the software from there.

The program will save the filters applied to the current image, when the user clicks on either the “Next” or “Previous” button to get to another image or when closing the program via the “Close” button. When exiting the program by closing the window, the changes will not be stored.

Classes of the FilterManagerTester

The *FilterManagerTester* application contained all the classes needed for the filter managing capabilities including all the filters. Figure 18 shows an UML diagram of the classes of the test environment.

All filters which were released with JAlbum 8.0 inherit from a class called *BasicFilter*. *BasicFilter* implements the *JAlbumImageFilter* interface and takes care of common parts used by all other filters. It is not applicable as a regular filter though as it returns false on both `isPrescale()` and `isPostscale()`.

The other classes of this diagram are in the following explained in brief what they do and how they communicate. For further details see JavaDoc[7] pages for the code provided on the CD (see “Content of CD” on page 64).

7 Since those classes are not documented very well, more information on how the classes work can be found in the JavaDoc for the later introduced PipelineManagerTester which is build on top of this architecture.

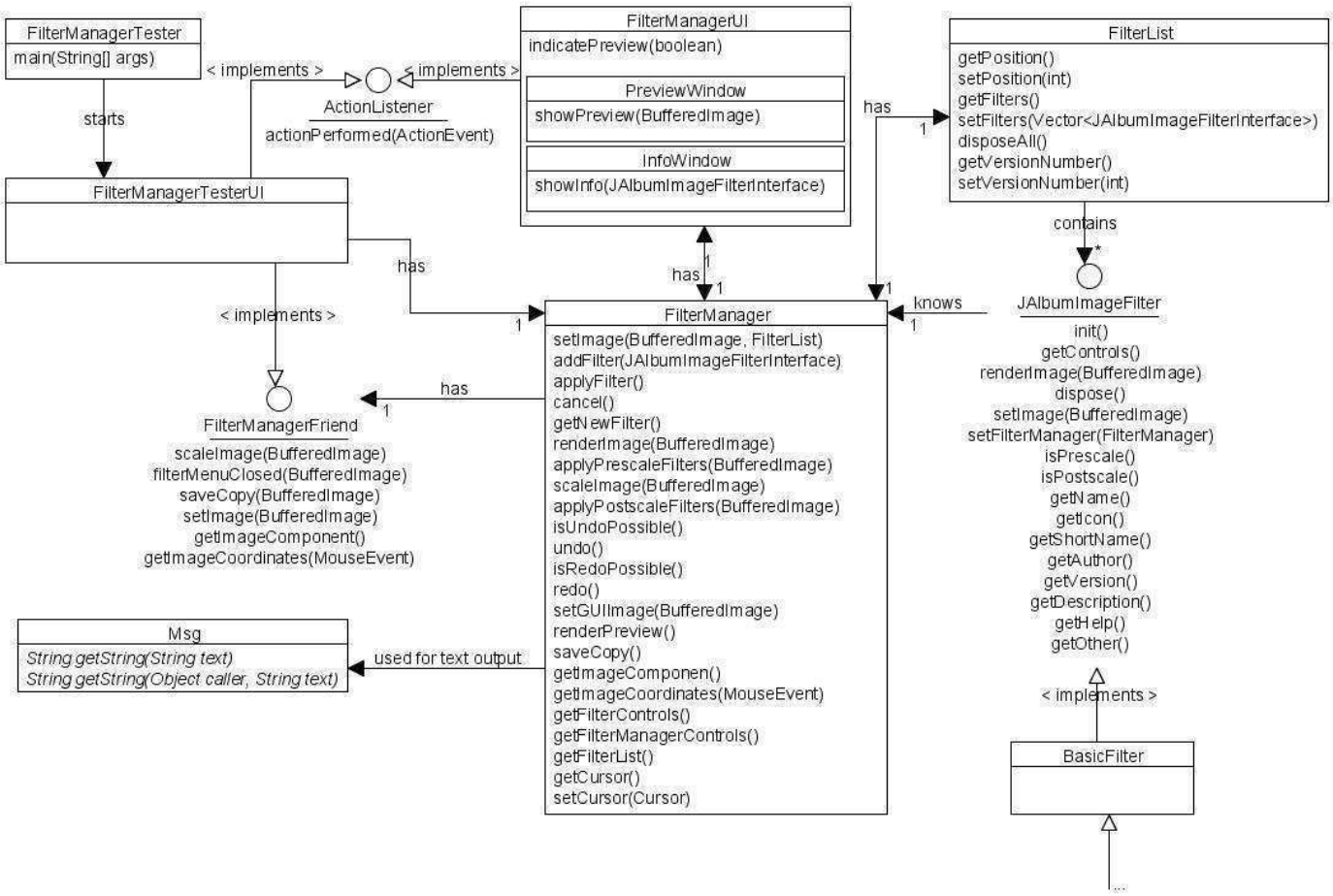

Figure 18: Class Diagram of the FilterManager and its corresponding classes

FilterManagerTester

The class *FilterManagerTester* only includes the main-method the program is started with. It instantiates the *FilterManagerTesterUI*.

FilterManagerTesterUI

The class *FilterManagerTester* is an instance of JFrame and provides the GUI for all other components. It provides all the buttons which can be seen in Figure 16. It has a list of predefined images through which the user can navigate with the buttons “Previous” and “Next”. All the buttons for the different filters are hard coded. Since it implements the interface *FilterManagerFriend* it offers the methods required by the interface, to the *FilterManager* which it instantiates upon its own construction. It loads the currently selected image and the stored filters to this image if there are any. Both are handed to the *FilterManager*. The *FilterManager* then applies all filters to the image

and returns the resulting image to the *FilterManagerTesterUI* to be displayed as current preview.

This class does not provide any methods to the *FilterManager* other then the ones specified in *FilterManagerFriend* interface, since the *FilterManager* only knows the interface and not the class which it implements.

FilterManagerFriend

The *FilterManagerFriend* is the interface used by the *FilterManagerTesterUI* to communicate with the *FilterManager*. The following methods need to be implemented:

`scaleImage(BufferedImage)`
This method is used by the *FilterManager* when rendering the image. The scaling of the image can not be done by the *FilterManager* as it does not know to what sizes the image should be scaled. So it forwards the task back to the JAlbum core. This method returns the scaled image as *BufferedImage*. In the *FilterManagerTestUI* this method however just hands back the original image as rescaling is not needed for the testing purpose.

`filterMenuClosed(BufferedImage)`
When the user either accepts or cancels the application of a filter this method is called with the image to be displayed on the GUI as a parameter. This image either has the filter applied to it or not depending on the user's decision.

`SaveCopy(BufferedImage)`
The user has the possibility to save a copy of the current image during the editing process. If the user selects that option from the GUI provided by the *FilterManager* this method is called and the implementing class is asked to take care of the saving.

`setImage(BufferedImage)`
This method causes the implementing class to show the handed image on the GUI. It is used to show the previews of the filters.

`getImageComponent()`
This method returns the component the image is displayed in. It is useful for adding additional listeners to it.

`getImageCoordinates(MouseEvent)`
Any filter is automatically registered as *MouseListener*, *MouseMotionListener* and *MouseWheelListener* of the component which contains the image. But if the user clicks on

an image, the coordinates of this click on the image component may not be identical to the coordinates of the clicked point on the image, since the image may be scaled or translated. This method translates those coordinates into a point which is then returned. In the *FilterManagerTestUI* this method just returns the coordinates of the click as the image will neither be translated nor scaled.

FilterList

The *FilterList* is the place where all the filters are stored. It is also the object which is exported when exporting the filters in order to permanently store the changes made to an image. It consists of 3 objects: an integer value representing the version number of the *FilterList* as the concept was designed for later extension and so this number is used to have an easy way of handling backward compatibility; a vector of the filters applied to the corresponding image in the order in which they were added by the user and an integer value representing the index of the vector to which the filters will be applied to the image. A schematic representation of the *FilterList* can be seen in Figure 19. The index is used for undo/redo. When the user undoes the last operation the index is reduced by one and the last applied filter will not be rendered onto the image anymore. In the figure the user first applied 5 filters before undoing the last two. When those filters are redone, the index goes back to 5. But if instead the user applies a new filter (Filter G) the filters E and F would be deleted, filter G would be appended to the vector and the index would be counted one up.

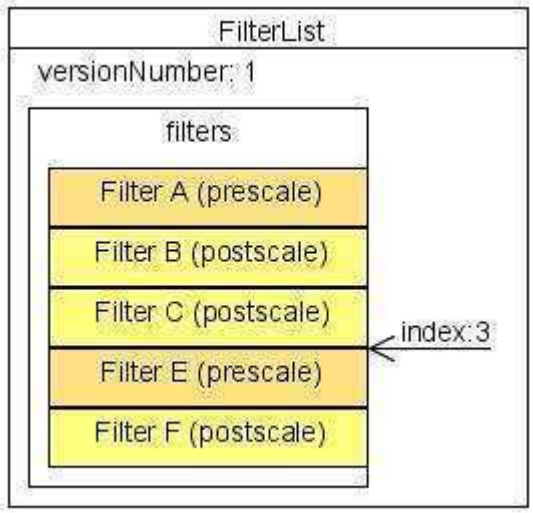

Figure 19: A schematic representation of the FilterList

FilterManagerUI

The FilterManagerUI takes care of the visual components of the *FilterManager*. Mainly it handles the input from the user via the 5 buttons seen in the bottom of the window in Figure 17. “Apply” and “Cancel” make the application return to the standard view as seen in Figure 16 either with or without the current filter applied to it. The “Preview” button opens a window which shows a preview of how the image will look like if the user applied the filter with the current settings. The “Info” button opens a window which displays the basic filter information as described in “Methods of JAlbumImageFilter Interface” on page 27. Both the preview window and the info window are inner classes of the *FilterManagerUI*. The “Save As” button allows the user to save a copy of the current image with the currently selected filter applied to it. The image saved is the same as the one being displayed in the preview window.

Msg

The class *Msg* is a simplified replica of the class *Msg* of the JAlbum core. It provides two static methods for accessing resource bundles for the localization of text. By calling the method `get(String text)` the text is looked up in the default resource bundle of JAlbum whereas the method `get(Object caller, String text)` looks for a resource bundle in the package of the calling class. If in the last method no matching localization of the text is found, the first one is tried before returning the text untranslated. When the *FilterManager* is embedded in the real application this class just needs to be deleted so that the “real” class *se.datadosen.jalbum.Msg* can be found.

FilterManager

The *FilterManager* is the core of the test application. It takes care of managing the filters and acts as the only connection between the class[8] implementing the *FilterManagerFriend* interface on the one side and the various filters and the *FilterManagerUI* on the other side. Because of this many of the methods of the *FilterManager* are just to pass calls from the *FilterManagerFriend* to the filter, like `getFilterControls()`, or from either the *FilterManagerUI* or the individual filters to the *FilterManagerFriend*, like `saveCopy()`, `setGUIImage(BufferedImage)`, `getImageComponen()`, `getImageCoordinates(MouseEvent)`, `setCursor(Cursor)` and `getCursor()`.

8 Currently the interfaced is implemented by the FilterManagerTesterUI but in the actual implementation this would be done by some class of the JAlbum core.

While some of the remaining methods are self explaining there are some which are worth taking a closer look at since they are vital to the application.

`setImage(BufferedImage, FilterList)`
When an image is loaded this method is called to make that image the current image which the *FilterManager* is managing. If there are any filters stored for this image, then the *FilterManagerFriend* hands those as well. Otherwise the second parameter will be **`null`** and a new *FilterList* is created for that image.

`addFilter(JAlbumImageFilter)`
This method is called when the users selects a new filter to be applied to the image. The filter preview of that image will be shown in the UI. The user then can work on the parameters of that filter. If the user wishes to apply this filter onto the image and hits the "Apply" button shown by the *FilterManagerUI*, the method **`applyFilter()`** is called causing the filter to be added to the *FilterList*. Likewise if the user does not wish to apply this filter and hits the "Cancel" button, the method **`cancel()`** is called causing the *FilterManager* to discard that filter and show the image as it was before that filter was selected.

`renderImage(BufferedImage)`
In the final rendering process this method is called. It consists of the three methods **`applyPrescaleFilters(BufferedImage)`**, **`scaleImage(BufferedImage)`** and **`applyPostscaleFilters(BufferedImage)`** which can also be called individually by the *FilterManagerFriend* if more flexibility is needed in the rendering process like for speeding up the process, for example by skipping the scaling for the preview or for inserting other operations like additional rendering of album filters if needed. Both of the methods for applying the filters go through the *FilterList* until the index is reached and apply all the filters which are supposed to be rendered in their section. Filters which can be applied both as prescale and postscale are applied as prescale filters.

Similar Concepts in other Software

Concepts similar to the Pipeline Concept are most commonly used for applying effects on video or sound clips in video editing software like Final Cut or Adobe Premiere or in sound editing software like Adobe Audition. Other software uses the concept to batch process several elements with the same pipeline like for renaming files. In principle adding an effect to a video clip is nothing else

then batch processing each single frame of that particular clip. Surprisingly the pipeline concept is not wide spread in the field of image editing software. The only two programs worth mentioning are *RoboRealm* for Windows and *Acorn* by *flying meat* for Mac.

Video Editing

Many applications for video editing use the pipeline concept for applying effects on video clips. The user sees a list of already applied effects and can change their settings. Generally these settings remain the same for the entire clip. But additionally it is also possible to set so called Key Frames at which the settings are defined for a specific frame of the clip. If there is a second key frame for the same clip, the values which differ between those frames will be interpolated for the frames in between. It is also possible to disable individual effects or change their order.

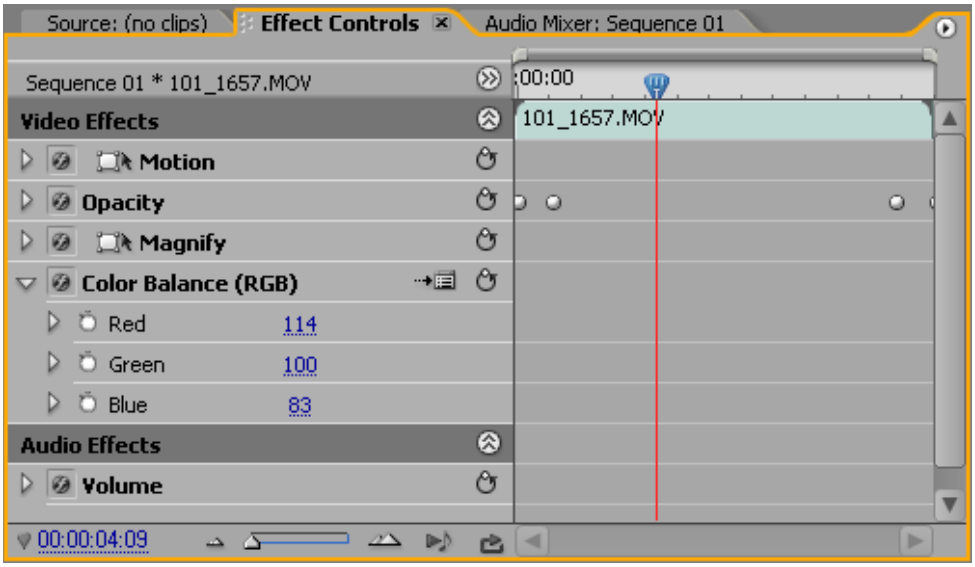

Figure 20: Effects Control Window in Adobe Premiere Pro

RoboRealm

RoboRealm[9] is actually a software developed for enhancing visual perception of robots but it can also be used for image editing. It works with regular still images from a hard drive as well as images from live cameras. It therefore can be understood as live video editing tool but because the user can work on and save still images as well as make snap shots of video streams and save those, it is mentioned here explicitly.

9 http://www.roborealm.com/

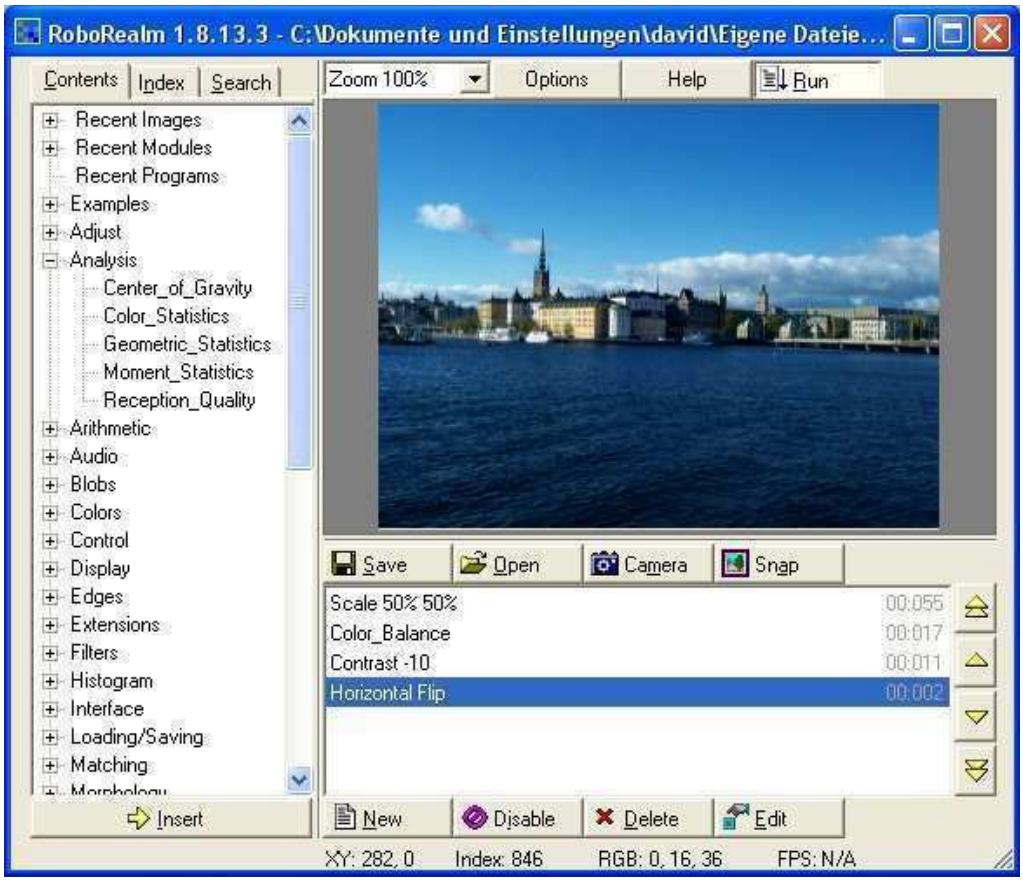

Figure 21: Screenshot of RoboRealm

RoboRealm has around 200 different “modules”. Most of them are from the area of image processing. There is also an if-module which can check any value of the image and depending on the outcome the following modules are applied or not. It also has script support so the users can program their own modules.

Even though image editing is possible with RoboRealm, that is not what this software was designed for and therefore it is unsuitable for less advanced users and lacks some of the functionalities expected from an image editing software.

Acorn

Acorn[10] is an image editing software developed for the Mac with many standard features for image editing software like layers, masking, wand selection etc. It partially supports the pipeline concept for single image editing. The user can apply different filter groups to the image. Those filter groups can consist of several filters in a particular order and with specific settings. While working on this

10 http://flyingmeat.com/acorn/

filter group the user gets a graphical representation of the filters as a pipeline and can set the settings for each filter as well as their order. An example of what this pipeline with some filters in it looks like can be seen in the screenshot in Figure 22. A preview of the resulting image can be displayed in the actual image window as well as in a specific preview window for the filters. In the filter preview window the user sees the image up to the point at which the currently selected filter is applied whereas in the actual image window the final image is always displayed. In the preview window of the filter the user can set certain filter parameters via mouse interaction like the positioning of filter objects or selecting a specific region for cutting. Once this filter group is applied to the image, the user can not change the parameters of those filters anymore and when using undo only the application of the entire group as a whole can be undone. It is possible though to save the filter group as "Preset" and then load an entire group with its specific settings instead of adding all filters manually again. So when the user did save the filter group, applied it but then does not like the resulting image it is possible to undo the application of the filter group, load the saved "Preset" again, adjust the settings and apply the filter group with the adjusted parameters again. This approach is highly unpractical though and is probably not what the saving of filter groups was intended for.

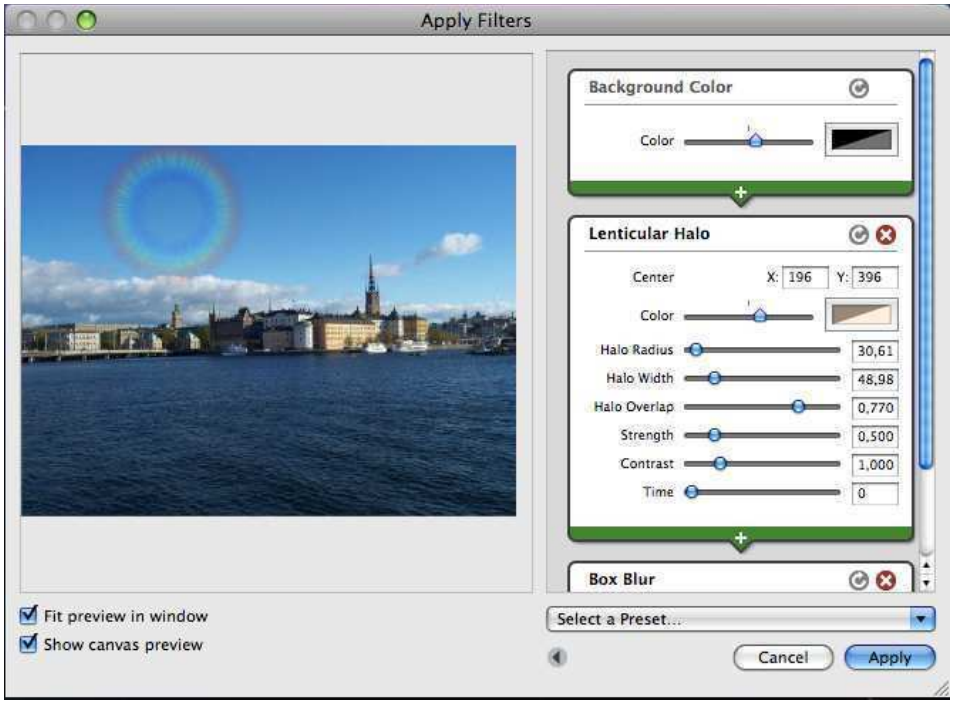

Figure 22: Screenshot of the Filter Window in Acorn

Acorn comes with around 80 different filters which are grouped into different contexts for better overview. Filter groups can only be applied to one layer. In order to apply filters to various layers

one can save the filters as preset and then apply the same filters to other layers as well but this can cause strange results when filter effects are applied to (semi-) transparent regions of a layer.

The Pipeline concept in context of JAlbum

As of version 8.0 the JAlbum software had a linear undo for the Image Filters where each undo step is the application of one of the filters. All the filters implement the interface *JAlbumImageFilter*. This gives a clear way of handling different filters in the same way. The list of filters represents the undo/redo chain. The user can undo already applied filters by pressing the button “Undo” or redo undone filters by pressing “Redo”. If a new filter is applied after one or more existing filters were undone, those undone filters are deleted and can not be applied anymore.

With the enhancement of JAlbum through the ability to edit images based on the Pipeline Concept this list of applied filters needs to become more flexible. The user should see any filter applied to a specific image and be able to change their order as well as their settings.

But since this concept is targeted on advanced users it becomes necessary to keep the old functionalities as well. So when the user goes on the “Edit Mode” of an image, the same GUI elements appear as in the previous version without the Filter Pipeline concept, only this time there is some kind of small button to switch to an “Advanced Mode”. This will change the view so that the user can work directly with the pipeline. In the following this mode is called “Pipeline Mode” or “Pipeline View” whereas the regular Edit Mode is called “Default Mode” or “Default View”.

Since there will already be the regular undo option in the Default Mode it would not be wise to implement a meta undo for the filter pipeline as well. If a meta undo was implemented then the user would have two sets of undo and redo buttons which have different behaviors. This will probably be very confusing for most of the users.

Some of the other specifications of the Pipeline Concept discussed in this paper so far have little relevance for the prototypic implementation of this concept within JAlbum. JAlbum is not an image editing software and does not aim to be one. It does however offer image editing functionalities so its users can improve their images before generating the album. Because of this focus it does not have many functionalities one would expect from a pure image editing software, like layers or masking. There is also no communication between the different filters and there are no global states.

5. *Design*

Extension of Test Environment

Because of the previous experience with the *FilterManagerTester* and in order to have the code done for this thesis to be easily distinguishable from the rest of the JAlbum software, the prototypic implementation of the Pipeline Based Image Editing concept into JAlbum is also done using the previously mentioned test environment. The version which will be created for this thesis will be named “Pipeline Based Image Editing Demo Application”. For simplicity reasons in the context of this thesis it will also be referred to as “PipelineManagerTester” or just “test environment”. To have a clear distinction between the version of the FilterManagerTester as it was when this thesis was started and the resulting version of the PipelineManagerTester both versions can be found on the CD belonging to this thesis as runnable Java programs and as source code .

Goals

The goal is to extend the filter handling capabilities of JAlbum so that the user can see the different filters in the logical order in which they will be applied to the image in the final rendering process. It then should be possible for the user to disable any filter or reenable filters as well as ultimately remove filters from that list. The order in which the filters are applied can be changed by moving individual filters up or down. By clicking on one of the filters, the user interface of that particular filter shows with the current parameters of the filter. The user should be able to change these settings and then see a newly rendered image with the new parameters.

If the concept were to be implemented into JAlbum later on, the GUI should not be visible to everybody by default as its main target user group are advanced users. So it will only be accessible via some kind of “Advanced” switch. The work flow for the regular users who do not use this pipeline should remain the same.

Requirements

The most important part of the implementation of the pipeline concept would be the visual representation of the filter pipeline so that the user is able to adjust the individual filters as well as their order. This pipeline should only be visible while being in the “Advanced Mode” where as the “Default Mode” still shows the UI similar to the implementation of the Album Filters in JAlbum

8.0. Both of those modes have a “Standard View” and a “Filter View” as described in the chapter “Test Environment for Developing Image Filters” on page 29 and as displayed in Figure 16 and Figure 17. For the Advanced Mode the filter pipeline will be visible in both views and the area on which it is displayed will now be called “Pipeline Panel”.

A schematic outline of two modes each with the two different view types can be seen in Figure 23 and a more detailed outline of the pipeline panel can be seen in Figure 24.

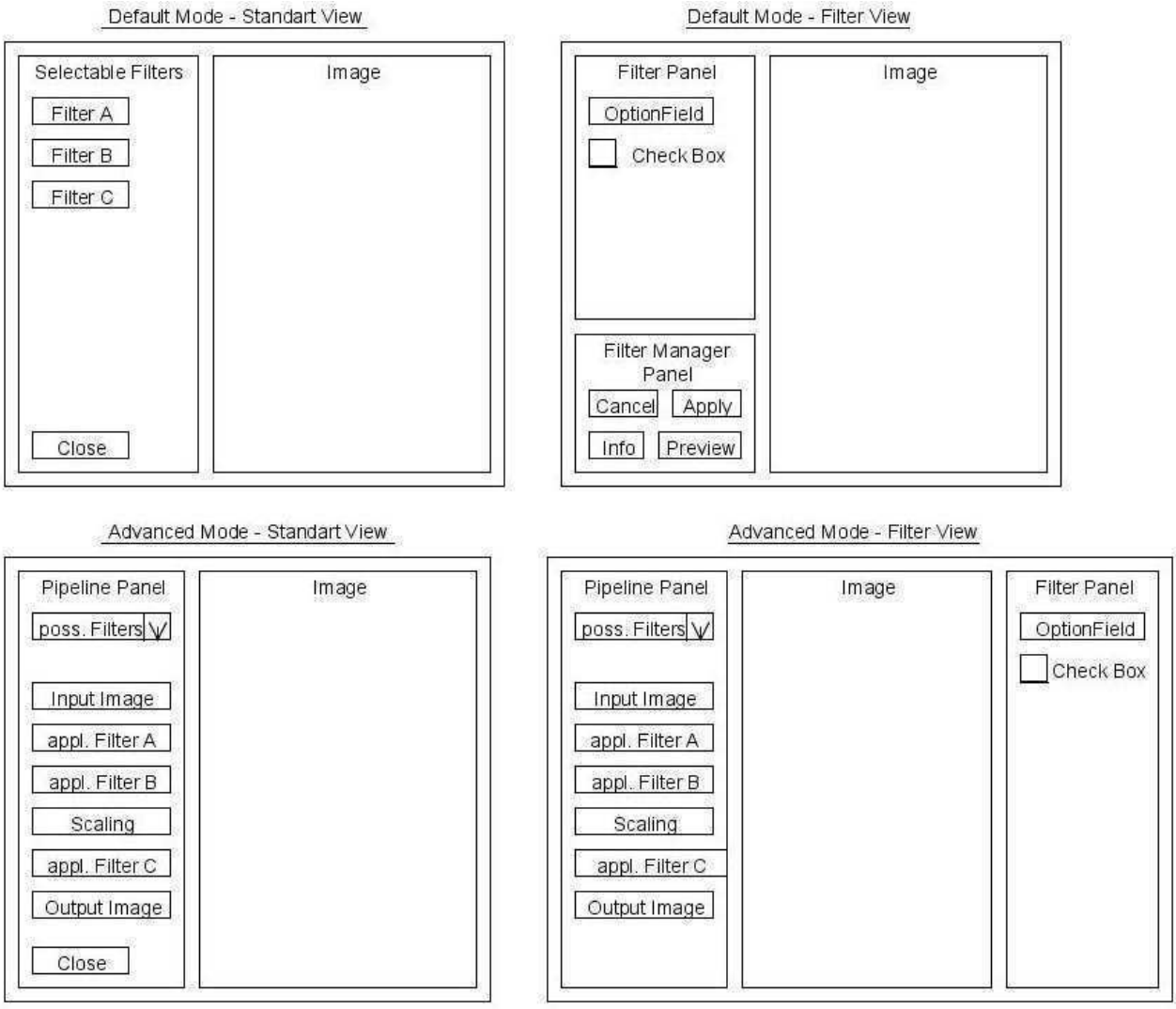

Figure 23: A schematic outline of the different views in the different modes

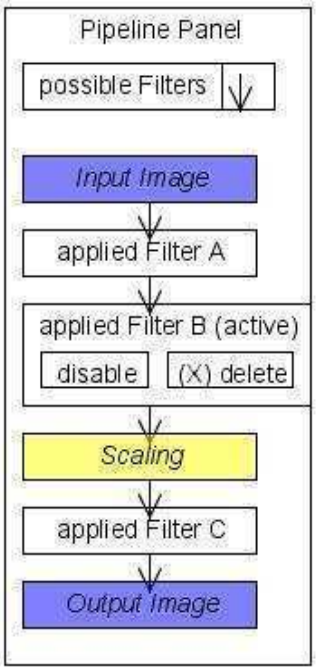

Figure 24: A schematic outline of the Pipeline Panel

The following paragraph explains the detailed requirements for the implementation of the pipeline concept briefly.

Pipeline Panel

- Show filters in the logical order in which they will be applied to the image. Differentiate between prescale and postscale filter. From now on the visual representation of a filter will be called “FilterPanel”.
- Show the scaling as an operation which is also applied to the image but also make it obvious to the user that the scaling is not a filter.
- Let the user change the filter order by moving the filters up and down via Drag and Drop. “Prescale only filters” should only be moved so that they are always before the rescaling where as “postscale only filters” can only be moved among the range of filters being applied after the rescaling. Filters which can be applied both, as prescale and as postscale filters, can be moved to any position in the pipeline.
- Individual filters can be disabled by clicking a button on the graphical representation of that filter. When they are disabled they still remain in the list at the same position but are not

rendered on the image anymore. They can still be moved. It should be obvious to the user which filters are disabled. When a filter is disabled, the same button it was disabled with can then be used to enable this filter again.

- Individual filters can be deleted. Once a filter is deleted it is completely removed from the list and can not be restored anymore. Disabled filters can be deleted as well.
- By clicking on one of the filters in the pipeline, this filter becomes the active filter. It should be obvious to the user that this is now the active filter. Buttons for disabling/enabling and deletion of the filter will appear. Also if the filter has a control panel this control panel will appear on the right side of the image. The image in the center will show the filter preview of the active filter. The filter panel should be wider than the other panels and reach on its left side to the image area to show that the image shown there belongs to this filter.
- By clicking on a filter while another filter is active the settings of the first filter will be saved and the control panel as well as filter preview of the second filter will be shown.
- By clicking on some other part of the pipeline panel with no specific functionality assigned to it while a filter is active the settings of this filter will be saved and the application returns to the “Standard View”.
- Above the graphical representation of the first filter should be a graphical representation indicating the input image. When clicking on that representation the input image will be shown as an image in the image area. It then should also be made obvious that the image shown in the center is the input image, similar to the indication of the active filters.
- Similar to the representation of the input image, a representation of the output image should be made. Both should be easily be distinguishable from the filters or the scaling. When no filter or other option like input image or scaling is selected this option should get selected automatically to indicate that the image shown in the image area is actually the output image.
- By clicking on the graphical representation of the scaling operation the user should get the option to set the scaling settings of the image, which would have been set via the album settings page in the regular JAlbum application as described in the chapter “The JAlbum Album Generation Process” on page 11. The scaling should then be simulated with the size for the CloseUp images.
- Above the graphical representation of the input image, the user should have the option to add new filters. In order to save screen real estate those filters can be put in a drop-down-menu with

a button next to it to apply a new instance of the filter belonging to the selected option.

- If there are more filters applied to the image than there is space on the screen, a scroll bar should appear to vertically scroll through the applied filters.

Graphical User Interface and Usability

The first version of the FilterManagerTester was created as a tool for the development of the *FilterManager* and for testing the filters. It also served as a demo tool to show the functionalities of the various filters. However it was created for internal purpose only and to be handled by the people who also developed it. Therefore not much attention was paid to the GUI of the test environment or its usability. Now that this software is used as a prototypic implementation of the pipeline concept which will be used more than just for internal testing purposes, some improvements of the GUI and the general usability need to be taken care of.

- The Default Mode should look as much as possible like the implementation of the Album Filters in JAlbum. It is part of this thesis to show that the existing implementation could be extended to support the pipeline concept. How those two concepts can be build on top of each other should not only be visible in the code but also to the regular user of the application.
- Have a clear separation between the Default Mode and the Advanced Mode. It should be easy to switch between those two modes either by some small button, by a link or by a key command. The current mode should be saved when exiting the program and then be restored when the program is started again.
- Give an option to open images. In the previous version a predefined set of images was used to test the *FilterManager* and the filters with. Now the user should be able to open and edit own images.
- Give an easy option to save the filters. The user should be able to save the applied filters any time and an exiting the program or changing to another image the user should be asked if the filters should be saved.
- Give an easy option to export the resulting image. Check if another image will be overwritten and warn the user. Check if the overwritten image is the currently opened image and warn the user in particular. If the user wishes to overwrite the image, load the image after the saving again as new image and delete all filters.
- Better handling of larger images. If the images were too large for the FilterManagerTester, the

filter menu of the selected filter will be shown off screen. A better handling is needed either by zooming out on larger images or by adding scrollbars.

- Open files on startup. Give the user the possibility to start the application with an own file via a start parameter.

Absolute Requirements

The new following requirements are a necessity and are not negotiable.

No Changes to the JAlbumImageFilter Interface

No changes should be made to the *JAlbumImageFilter* interface. The idea of this interface was to allow other developers to create their own filters. Even though it has not been officially released to outside developers yet it still can be that other developers already implemented it.

Localization

The JAlbum software is available in 32 different languages. So localization is a key part of the software. Any language strings should be stored in property files to be easily translated and exchanged. This effects in particular the classes which would be reused if the concept was to be implemented into JAlbum later on. For the classes of the test environment, localization is not that important.

Undo/Redo

The pipeline concept means opening up the chain of steps of the linear undo model. Now the user can change any setting of any filter previously applied. This makes a linear undo dispensable. However the software still offers the option of applying filters as it was possible in the previous version. There the only way for the user to change previously set filters is to undo them. Thus the software still needs to continue supporting linear undo. This should also work if the user applied some filters, changed into the Advanced Mode, edited some filters there, then changed back into the Default Mode and pressed undo there. So despite their order in the pipeline the filters should still be undone in the reverse order in which they where applied. If a filter is undone and then redone it should be inserted at the same position where it previously was and the resulting image should still be the same as it was before undoing the filter.

But if the user undoes a filter, changes to the Advanced Mode and does something which has an

effect on either the order or the number of the applied filters, goes back to Default Mode and tries to redo the undone filter, then it can not be guaranteed that the filter will be redone at the right position. Therefore it is legitimate to insert it at any suitable position.

Backward compatibility

Version 8.0 of the JAlbum software saved the entire filter list by parsing it in a XML document. No matter how much the classes change, it still needs to be insured, that settings saved with version 8.0 would still be readable in the newer version. Also any image rendered with filters should render exactly the same in the new version as in the previous one.

Since one of the ideas behind the filters was to give developers the option of creating their own image filters it needs to be assured that filters written for JAlbum 8.0 can still be applied to images even if there are changes in the requirements for the filters. It is acceptable if these filters do not offer all the functionality needed for the pipeline, like the possibility of adjusting the parameters later on but they should still be applicable and renderable.

Soft Requirements

Not all of the requirements are as specific and strict as the ones mentioned above. Here are some soft requirements.

- Change the existing filters as little as possible. As described above even though it is unlikely that filters of other developers already exist it can not be ruled out. Also any bigger changes would require to put those changes into any existing filter which calls for a lot of extra work.
- Performance sensitivity. Operations on images particularly on large images can require a lot of performance. So when changes are done by the user and the software needs to recalculate the resulting image especially when other filters are also effected, as little recalculation effort as possible should be done but as much as needed.
- A modular architecture should be retained. New functionality should be encapsulated in suitable classes for possible later extension and easy maintenance. The regular editing should remain independent from the pipeline editing so it could still be used if the the pipeline functionalities were to be removed again later on.

Ideas which will not be implemented

When working on the first implementation of the Album Filters as well as during the design phase of the Pipeline Concept some ideas came up on how to further improve the handling of the filters within JAlbum. Those ideas however will not implemented as part of this thesis either because they are only loosely connected to the pipeline concept or because the estimated time needed to implement those features is not justified by the expected results.

Showing Album Filters

Even when using Image Filters it is still possible to use Album Filters as well. The Album Filters are applied during the final rendering process as can be seen in Figure 7 on page 13. During the preview of the Image Filters they are not rendered on the image which can cause some confusion as the final rendered image will be different than in the preview. In the long term it would be useful to let the user set the Album Filters via a GUI instead of album variables. This however will not be done in this paper. Also it will not be indicated to the user in the pipeline panel if and what album filters will be applied. This would require important changes to the *FilterManagerFriend* interface which can not be easily simulated by the test environment.

Apply Image Filters on Thumbnails or CloseUps only

When the user applies album filters it is possible to specify if they should be applied on only the Thumbnail images or only on the CloseUp images. By default they are applied to both. Image filters however can only be applied to both kinds of images and will be applied when the image has the CloseUp size. Afterwards a copy of the images will be scaled to Thumbnails size. Letting the user apply specific filters to only one kind of image will be too much effort to implement and will will be hard to simulate in the test environment.

Automatic Import of Possible Filters

In JAlbum the possible Image Filters are checked on startup by looking for classes implementing the *JAlbumImageFilter* interface and offering them to the user. For the test application this feature would be dispensable as it needs to be done by the part simulating the JAlbum core. This will lead to duplicate code as this feature already exists within JAlbum. Taking the existing code from the part of JAlbum which takes care of loading the filters however is not appropriate as it was written by the JAlbum team and was not developed as part of this thesis.

6. Implementation

Figure 25 shows the class architecture after the implementation of the Pipeline Based Image Editing concept into the test environment.

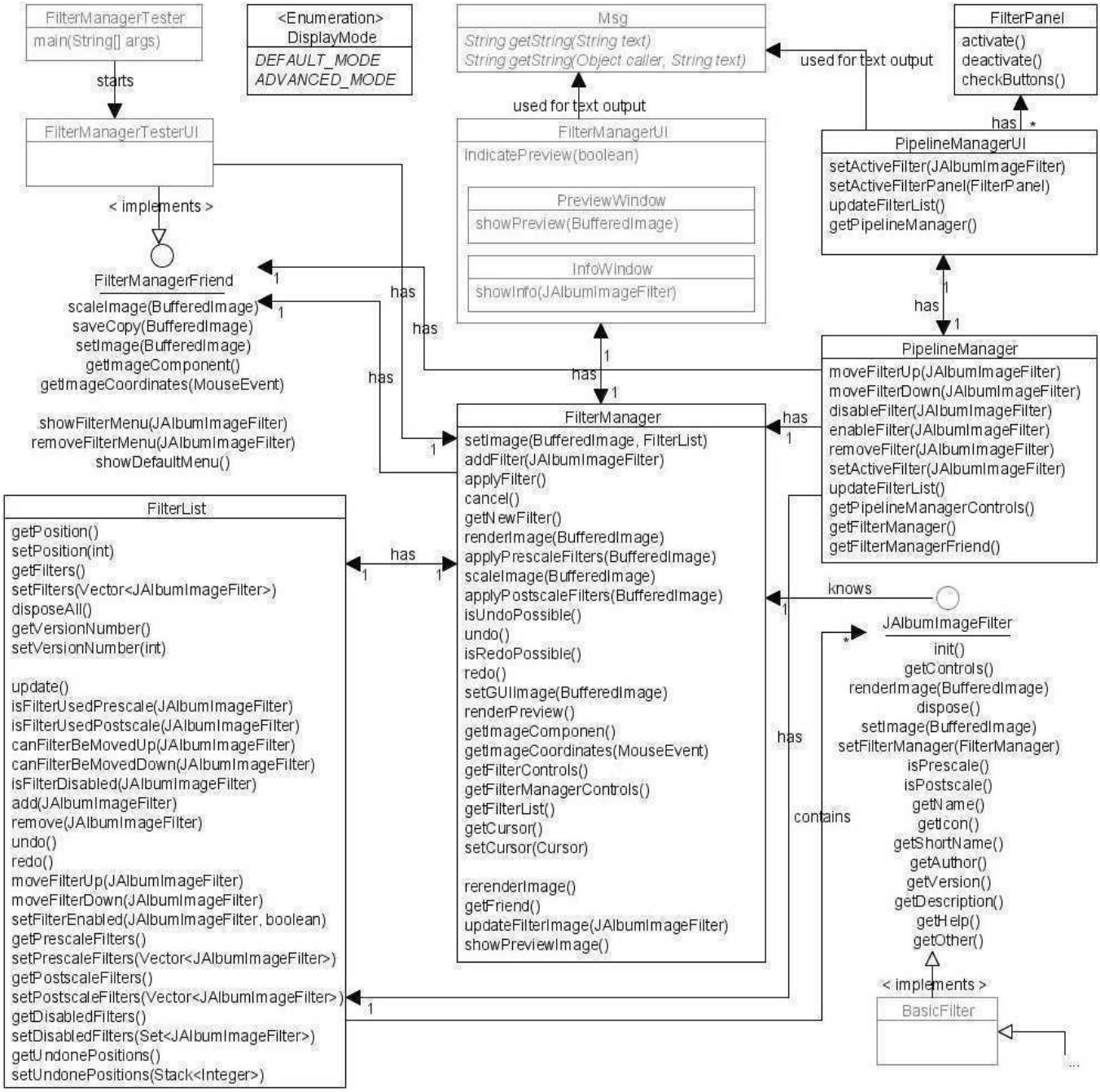

Figure 25: Class Diagram of the PipelineManagerTester application

Classes which have not changed at all are drawn in gray. New methods added to previously existing

filters are added after an empty line. In this diagram eventually implemented listener interfaces like *MouseListener* or *ActionListener* are not shown to keep a certain level of clarity. In general it can be said that all classes which end with *UI have the *ActionListener* interface implemented and most of them also the *MouseListener*. Other implemented listener interfaces are not worth mentioning here as they do not offer any vital functionality for the application.

One of the ideas behind this architecture was to keep the pipeline functionality independent from the basic filter functionality. Even though some of the original classes had to be changed to better incorporate the pipeline functionality, those classes are not aware of the newly added classes. So if one would just remove the classes *PipelineManager*, *PipelineManagerUI*, *FilterPanel* and *DisplayMode*, none of the remaining classes[11] related to the *FilterManager* would break and editing the images in the Default Mode would still work.

Changes to the FilterManagerTesterUI

A lot of the improvements to the GUI and the Usability of the demo application were done to the *FilterManagerTesterUI*. The program now shows in the default *Look and Feel* of the Operating System. Many of the standard operations which have previously been offered as buttons are now offered in the menu bar of the program. The menu items include “Open” to open a new image, “Save Changes” to save the filters applied to that image, “Save Image As” to export a copy of the image with the filter applied to it, “Exit” to close the software, “Undo” and “Redo”, and the possibility to switch between the Default Mode and the Advanced Mode. All of these commands also have keyboard shortcuts assigned to them which use de-facto standards if such exist, like “Control + O” (or “Cmd + O” on the Mac) for Open and “Control + Z” (or “Cmd + Z”) for Undo.

When the user tries to close the application either by using the close option in the menu or by closing the program window, the software asks if the user wishes to save the changes done to the current image. The same is done when opening a new image. While opening an image, the user is offered a file selector and can import any file with the file extension *.jpg, *.jpeg, *.gif, *.png or *.bmp. If filters are stored to the selected image they are loaded as well.

The Default View now looks a lot more like the GUI of JAlbum. The two different views are represented by the enumeration *DisplayMode*.

Since this test application is in this case used to demonstrate the pipeline managing possibilities it

11 The only class in the architecture which is aware of the PipelineManager and its related components is the FilterManagerTesterUI. Removing the PipelineManager would therefore also require changing the FilterManagerTesterUI to avoid compiling errors.

was considered renaming this class to *PipelineManagerTesterUI* and the class *FilterManagerTester* to *PipelineManagerTester*. This however was not done as it would imply that those old classes were replaced by the new ones which is not the case. The old classes where merely extended to now support the *PipelineManager* besides the existing *FilterManager*. Also the *PipelineManager* depends on the *FilterManager* but not vice versa, so the *FilterManager* is the more important structural part of this application.

Changes to the Filters

The filters were not changed much. The *JAlbumImageFilter* interface did not change at all as it was stated in the Absolute Requirements. In the previous version of the *FilterManager*, the `dispose()` method of the filters was called before they were exported. In this method the filters is able to delete all content which was not supposed to be exported. Just the necessary settings were saved for the filter to be rendered again later on. Since the user was not able to change those settings later on, the GUI was not needed anymore and therefore was not exported. Now that the filter can be adjusted at any time the GUI needs to be restored even when the filters were exported between the adding of the filter to the pipeline and the calling of the GUI. So each filter now restores its GUI within its `init()` method and sets it according to the saved parameters.

Changes in the FilterManagerFriend Interface

Previously the only time a control panel of a filter was displayed was when the user added a new filter. This button was part of the class which implemented the *FilterManagerFriend* interface and so this class always knew by itself when a control panel of a filter was supposed to be displayed. Therefore it was not necessary to have a method in the interface for redisplaying a filter panel. Now a control panel can also be displayed via the graphical representation of the filter pipeline. So the method `showFilterMenu(JAlbumImageFilter)` was added to the interface. With the methods `removeFilterMenu(JAlbumImageFilter)` the control panel is remove or with the method `showDefaultMenu()` the UI is reverted to the standard view of the currently selected mode.

Changes in the FilterList Class

Previously the *FilterList* just stored a list of filters in the order in which the user applied them to the image, the index to indicate which filters are undone and a version number. Now that the user can define the order in which the filters will be rendered onto the image this simple list will not be

suitable anymore. Internally the *FilterList* now stores two now additional lists for the filters: one for prescale and one for postscale filters. The list is sorted by order in which the filters will be rendered onto the image and not the order in which the user selected them. Because of the requirement that the old way of undoing/redoing filters should still be possible in the Default Mode the old list with the index is still kept. The class also holds a map for the disabled filters and a list of the positions where the undone filters should be inserted again if they were redone.

In the previous version it was the *FilterManager* which changed the list of filters stored in the *FilterList* as well as the index number. Now that functionality is embedded in the *FilterList* class and provided to the other classes by the methods **`add(JAlbumImageFilter)`**, **`remove(JAlbumImageFilter)`**, **`undo()`** and **`redo()`**. But the *FilterList* also provides services to the *PipelineManager* class with the methods **`moveFilterUp(JAlbumImageFilter`**), **`moveFilterDown(JAlbumImageFilter)`** and **`setFilterEnabled(JAlbumImageFilter, boolean)`**.

Additionally the class also has default getter and setter methods for exporting and importing the various lists since this class is exported by the XML parser when saving the applied filters. More information on the requirements of the parser can be found in the chapter “Restraints on the Image Filters” on page 28. When exporting the FilterList with the XMLEncoder the encoder prints out but several errors and exceptions does not throw them. This however does not effect the functionality of the export and can safely be ignored. The same happens when importing the list again especially when it is a FilterList of the previous version.

The PipelineManager Class

The *PipelineManager* class manages the commands coming from the *PipelineManagerUI* class. Many of them are just forwarded to the *FilterList* which holds the actual logic for disabling filters or moving them up and down. But the *PipelineManager* also handles the rendering by communicating with the *FilterManager* and the updating of the other GUI components other then the *PipelineManagerUI*.

The PipelineManagerUI Class

The *PipelineManagerUI* class is an instance of *JPanel*. It is the graphical representation of the filter pipeline and it corresponds to the “Pipeline Panel” in Figure 24 on page 43. The user sees the filters

and the order in which they are applied. The filters are shown as FilterPanels which are drawn on the panel of this class. By clicking on the FilterPanel belonging to a filter, the user can select that filter and it will then become the active filter. For this the method `setActiveFilterPanel(FilterPanel)` is called by the *FilterPanel* with itself as a parameter. For the active filter the user gets the options to move it up and down, if possible, to delete it or to enable/disable it.

Since an active filter can also be set by adding a new filter the method `setActiveFilter(JAlbumImageFilter)` exists to be called by the FilterManagerTesterUI. When any changes are done which have an effect on how the filters are displayed the method `updateFilterList()` can be called to redraw the panel.

The FilterPanel Class

The *FilterPanel* class is the graphical representation of an individual filter. It listens to the changes which are requested for that particular filter by the user and forwards them to the *PipelineManager*. With the methods `activate()` and `deactivate()` the filter can be shown as being the active filter or reset to a regular filter. If the filter is the active filter, its filter panel is drawn with a green background and it is wider so the user can see that the image currently displayed on the image area in the center is the filter preview of that particular filter.

The method `checkButtons()` for updating if the buttons are still valid options. This is used to disable or enable the buttons for moving the filter up or down. For example: the top filter can not be moved up anymore and therefore the button for moving it up should be disabled. But if the second button is moved up the previously first button can be moved up again and thus the button should be enabled again.

Changes in the FilterManager Class

As already mentioned many of the functionality previously done by the *FilterManager* is now done by the *FilterList*. But since the *FilterManager* is still the partner of communication for the individual filters as well as for the class implementing the *FilterManagerFriend* interface, those methods still exist, only that their calls are now routed to the *FilterList*.

Since it may be required to render the filters to the current image now more often than just after a filter was applied the method `rerenderImage()` was added so that the *PipelineManager* could

request the current image to be rendered again after the user did some changes to the filter pipeline, either by changing the settings of a specific filter or by changing the order or the amount of filters being applied. The method `showPreviewImage()` would cause the last output image to be displayed on the image area in the center of the application again. This is used when the user closes a filter menu and the user could not do any changes to the filter because it was disabled. This way less rendering is required.

In the previous version once a filter had its input image it was never changed again since the settings of the previous filters could not be changed. Now that those settings can change the input image of a filter can also change. So when a filter is selected and its image is shown in the image area again the method `updateFilterImage(JAlbumImageFilter)` is called to get an updated image as a new input image.

Not implemented Requirements

In general most of the previously stated requirements could be implemented as described. However not every single requirements was implemented. This was mostly due to the time constraints of this thesis. Some elements though could not be implemented for other reasons. The requirements which have not been implemented are mentioned in the following. If no additional explanation is provided on why they are not implemented it is because the time and effort required to implement them was not in a reasonable relation the the gained functionality. Since the application done as part of this thesis is merely a prototypic implementation of the pipeline concept and not supposed to be a full blown releasable software it seemed reasonable to focus on getting the basic functionality to work instead of spending too much time polishing details. The aspects are listed in the order in which they were mentioned in the requirements and not according to their importance.

Pipeline Panel

FilterPanels are not wide enough

The original plan called for the active FilterPanel to be wider than the other panels and reach to the right side till the image area. In the implementation the active FilterPanel is wider but not as wide as it was supposed to be. There is a gap of around 20 pixels between the right end of the FilterPanel and the image area. This is because of the offset caused by the different panels being put into each other and because of the scroll pane in which the filter panels are embedded. Usually this scroll

pane is not visible unless there are too many filters but it still forces some spacing between the filter panel and the image area.

Panel for input image not selectable

The panel which represents the input image is not selectable by the user as it is described in the requirements. This also means that the user can not see the input image without any filters applied to it, unless there are no filters in the pipeline and the output image is equal to the input image.

Panel for output image not selected by default

When the user clicks anywhere on the pipeline panel where there is no filter panel, the currently active filter panel is deactivated and the output image with all filters applied to it is shown. But the panel for the output image does still not get hightlighted as active.

Show Scaling Options when clicking on scaling panel

When the user clicks on the panel for the scaling option the active panel gets deactivated and the output image is shown as if the user had clicked somewhere on the background. In contrast to the requirements the scaling options do not get shown and can not be changed at this point. This is because this requirement was reevaluated during the implementation phase with the conclusion that even though such an option would make sense for this demo application if that option was to be implemented into JAlbum it could cause more confusion to the user as it would be helpful. The setting for the rescaling would effect all images but if the user were able to change the settings next to the Image Filters it would imply that this setting is image specific.

This option however would be suitable for a later implementation of a GUI for the Album Filters.

As an alternative for this dropped requirement it was considered building a way of setting the parameters for the rescaling via the menu bar. This however was not implemented due to time constraints.

Move Filters via Drag and Drop

Other then specified in the requirements the filters can only be moved up and down by pressing the according buttons on the filter panel and not by dragging and dropping them. An implementation of the Drag and Drop support would have required a lot of additional programming effort which was not reasonable as the filters could already be moved up and down even though not as easy as with

Drag and Drop.

If that implementation of the pipeline concept was ever to be made part of the JAlbum application however supporting the Drag and Drop for the filters is a must for the usability.

Graphical User Interface and Usability

No Scroll Bars or Zooming for Image Area

If an image is to large for the application to be display as a whole only the center part of the image gets displayed. There is no zooming and no scrollbars. But unlike in the previous version the control panels do not get pushed off screen if the image is to large.

Missing Graphics and improved GUI

Some of the basic functionalities would appear nicer to the user if they had additional graphics. This was partially not done because of time constraints but also because some of such graphic improvements had already been implemented by the JAlbum team before the release of JAlbum 8.0 but after the work on the implementation of the pipeline functionalities had started. So adding the graphics based on the changes done by the JAlbum team would not reflect the work done for this thesis. But implementing it again on the other hand would be redundant work if the functionalities were to be added to JAlbum.

The same goes for the GUI provided by the class *FilterManagerUI* including the Preview Window and the Info Window. It already is improved in JAlbum 8.0 and to avoid doing duplicate work or including code not done as part of this work, the GUI of the *FilterManagerUI* was not improved.

Missing Tool Bars, Mnemonics and Tool Tips

Many of the commands for the *FilterManagerTesterUI* have now been grouped and moved into the menu bar. For better usability it would be equitable to have the most often used commands also be displayed as icons in a tool bar. The commands in the menu bar itself are missing mnemonics for easy selection via the key board. But since the existing commands already have keyboard shortcuts assigned to them those other options were considered dispensable.

Most of the buttons are also missing tool tips explaining what the resulting operation does in more detail when the user hovers with the mouse over it.

Absolute Requirements

Localization

While all the components which might later be used by JAlbum if the pipeline concept were to be adopted, are completely localized, the text which is displayed by the class *FilterManagerTesterUI* is not localized. This however is not important as it is only used in this test application and presumably not going to be released outside the context of this thesis.

Possible Further Improvements

During the implementation process some more issues became obvious which were not part of the requirements and also were not implemented but should be build in if the software were to be extended later on.

Performance

The implemented architecture was designed with the constraint that the *FilterManager* should be independent from the *PipelineManager*. This approach might be modular and flexible but is has to be paid with performance losses. When the *PipelineManager* now changes the order of the filters the entire stack of filters has to be rendered again as the *FilterManager* is not aware of what changes can be done and are done by the *PipelineManager*. If the *FilterManager* would know which filters are changed only those could be rerendered. For larger images[12] with several filters the time needed to move a filter up for instance is definitely noticeable as the delay can take up to several tens of seconds. For smaller images or images with only view filters applied to them this delay is hardly noticeable or only takes a few seconds. The specific time needed to render an image also depends on what filters are used and in what order.

Export to other file formats

Currently the only file format supported to be exported to is JPEG even though other image formats can be opened. This however is not needed as JAlbum itself can only export to JPEG files.

12 Larger images in this context are images with more than 3 megapixel.

7. Result

The purpose of this thesis was to create a concept for software where the user can change the parameters of any previously done operation as well as the order in which those operations are performed. This concept was supposed to be developed on an abstract level so it would be valid for any kind of creative software. But in order to be implemented this concept then needed to be specialized to the general level of image editing software. On this level it would be possible to describe what specific problems and challenges will arise when implementing this concept for image editing software. As a next step the concept was supposed to be adopted in a prototypic application based on the image editing functionalities of the photo publishing software JAlbum.

Evaluation of the Concept

The developed concept shows that the user does not have to be bound to just undoing operations which did not lead to the desired result. By letting the user change the parameters of any applied operation the software gains a lot more flexibility. This might be especially interesting for advanced users. The concept also demonstrates what particular problems might occur when adopting this relatively abstract concept to the field of image editing. Many of those problems can easily be solved once one is aware of them while others might require trade offs of other functionalities. Not every software however is suitable for implementing this concept and not every problem can be solved easily.

It is hard to say why the Pipeline Concept is so rarely found in other software and in particularly in image editing software. Most other software which use such a kind of concept are used for batch processing of images or for videos. But since videos can be understood as a batch of images for every frame those two kinds of software do not differ so much. One of the advantages for videos would be that the application always guaranties the final size of each image or frame, since it is given by the settings for the current project. So at least some of the problems which can arise from the concept can be avoided by using coordinates relative to that output size. Also the effects which are applied at real time are only done to low quality preview images. So the performance issues which can arise at bigger images are not given here. The effects are applied at the real images when the final rendering process is started. There the user knows that might take longer and does not expect instantaneous results.

Evaluation of the Prototypic Implementation

The implementation of the Pipeline Concept into JAlbum demonstrates how easy existing image editing functionalities could be extended to implement the pipeline concept. It is even possible to keep the previous mode of operation working for the users who do not wish to use the Pipeline functionalities.

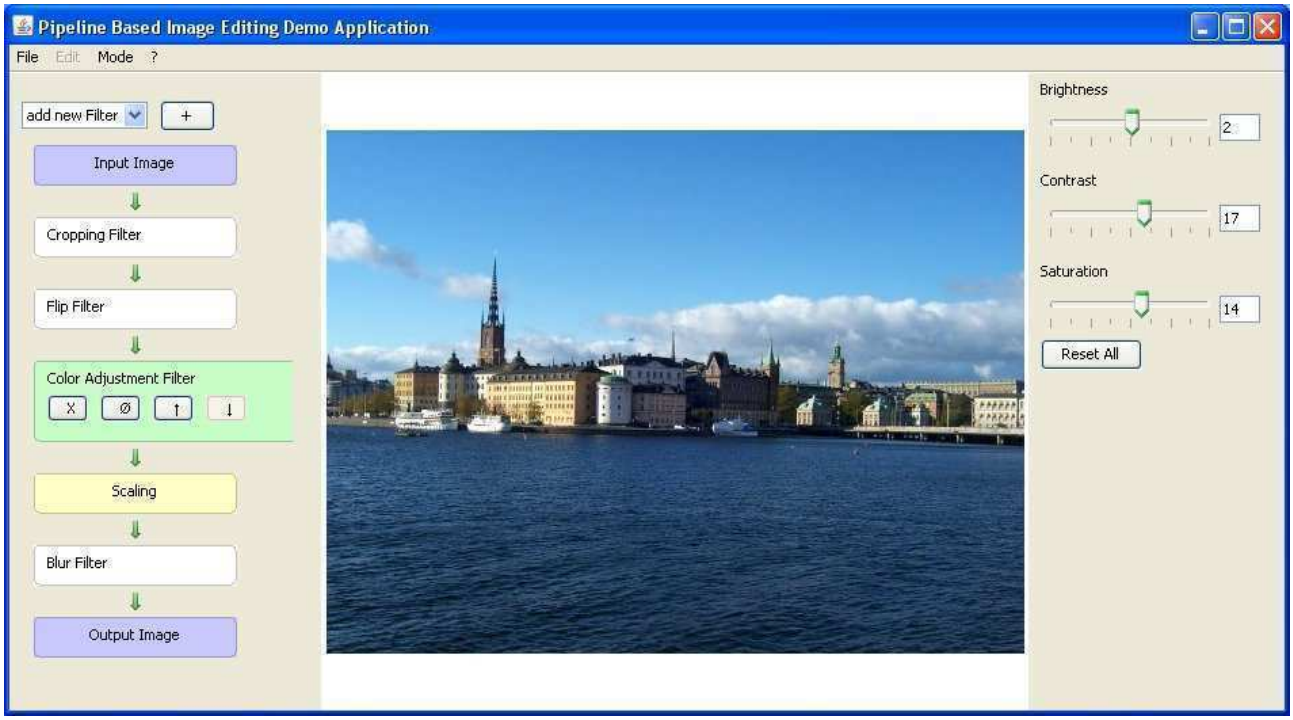

Figure 26: A screenshot of PipelineManager application in the Filter View of the Advanced Mode

The implementation of cause still misses some features one would expect from a regurlar image editing software but it is specifically only a prototypic implementation. Despite this it clearly demonstrates to the user how the Pipeline Concept works and how the user profits from this concept with more flexibility and a better understanding of the image generation process. This is especially important in the context of JAlbum where the filters are differentiated in prescale and postscale filters.

8. Outlook

Integration into JAlbum

As the concept and the software was developed to extend the image editing functionalities of JAlbum the obvious next step would be to build this kind of functionality into JAlbum itself. This however will raise some challenges as the classes for the general image editing, like the FilterManager, which are used by JAlbum have also been changed since they have been included. So they are not identical anymore with the classes the test application was build onto. With the development of this test application and the continuing development of the FilterManager as a part of the JAlbum software by the JAlbum team, the two development branches drifted apart from each other and caused a forking of the software. The challenge will be to merge those two branches back together if the new classes should be incorporated into the JAlbum software.

Integrating this concept into JAlbum is a big chance. When the version 8.0 of JAlbum was released it got a lot of positive feedback of the JAlbum community. The image filters where one of the major changes of that software. Implementing of the pipeline concept would strengthen the image filters even more especially for professional users. Of cause it would also be important to publish the API for Image Filters in order to get other developers involved and give the users even more tools to enhance their images.

Standalone Image Editing Software

As the test application shows it is also possible to work with the JAlbum image filters even without using JAlbum. It would be one option to further develop this test application to become a standalone light weight image editing software which can be released to the public. So users could edit their images independently from publishing them with JAlbum.

In order to make this test application usable for the general public it would be necessary to implement those requirements which have not been fulfilled yet as described in the chapter “Not implemented Requirements” on page 54 as well as the possibility to export to file types other then JPEG and an automatic import of available filters. For such an application it would also be good to remove the code which is taking care of specifics regarding the implementation into JAlbum. The most important of those parts would be the differentiation between prescale and postscale filters. Since there would be no implicit scaling required in an general image editing software any filter can

be applied at any time.

The application would also be of use for developers who are interested in making their own album filters as it would offer an easier way of testing them without JAlbum.

Future Improvements

Integration of Album Filters

As already mentioned, one of the points which was not planed to be implemented as part of this thesis is the integration of Album Filters. This would be one of the next logical steps. In the first place it means offering a nice way of adding Album Filters via a GUI and also setting their parameters that way. Also it would be useful to show the user what album filters are applied before and after the image filters so that the user can take their effect into account when setting the parameters for the image filters. The album filters however are not supposed to be set or adjusted at the same place where the image filters are. It is important to keep those two separated so that it is obvious to the user that they are two different kinds of filters. Otherwise some users might adjust their album filters to look good for on specific image and then wonder why other images have changed as well.

An important part of the integration of the album filters is that the user can see the effect of a filter immediately after adding it to the list of album filters and does not have to wait until the album is already generated.

Node Based Software

In this thesis all the filters take one image as input and produce one image as output. In the simple case when there is only one filter and no scaling operation, the input image of the filter is the image the user selected and the output of the filter is also the output image of the application.

One extension of this concept would be to have filters or operations in general which can have more than one input image and more than one output image. This concept is called “Node based” where each of the operations is referred to as node. Those nodes can be connected together to form complex nets of operations. This concept is commonly used for 3D modeling and for the rendering of visual effects.

There are also some small applications which try to implement this concept for image editing like

Image Effect Trees from *Optunis Imaging*[13] and *Image Genius Professional* from *Pixel Dragons*[14]. *Image Genius Professional* is explicitly designed for batch processing images.

This concept was intentionally not mentioned in the thesis before as it would have dramatically increased the complexity of this thesis or the application if it were to be implemented.

Extending the pipeline concept to a node based concept however would be the next obvious big step to give the user even more possibilities to edit their images. Implementing a node based image editing concept into JAlbum though would overshoots the purpose of image editing on within JAlbum as JAlbum is not primarily an image editing software.

13 http://www.optunis.com/imaging/image_effect_trees_info.html
14 http://www.pixeldragons.com/Products/ImageGenius/Index.ashx

9. Appendix

FilterList class parsed as XML

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.6.0_03" class="java.beans.XMLDecoder">
 <object class="net.jalbum.filterManager.FilterList">
  <void property="filters">
   <void method="add">
    <object id="CroppingFilterPlugin0"
class="net.jalbum.filters.CroppingFilterPlugin">
     <void property="resizableBox">
      <null/>
     </void>
    </object>
   </void>
   <void method="add">
    <object id="ColorAdjustmentFilterPlugin0"
class="net.jalbum.filters.ColorAdjustmentFilterPlugin">
     <void property="brightness">
      <double>5.0</double>
     </void>
     <void property="contrast">
      <double>1.1600000000000001</double>
     </void>
     <void property="saturation">
      <double>1.000625</double>
     </void>
    </object>
   </void>
   <void method="add">
    <object id="BlurFilterPlugin0" class="net.jalbum.filters.BlurFilterPlugin"/>
   </void>
  </void>
  <void property="position">
   <int>3</int>
  </void>
  <void property="postscaleFilters">
   <void method="add">
    <object idref="BlurFilterPlugin0"/>
   </void>
  </void>
  <void property="prescaleFilters">
   <void method="add">
    <object idref="CroppingFilterPlugin0"/>
   </void>
   <void method="add">
    <object idref="ColorAdjustmentFilterPlugin0"/>
   </void>
  </void>
  <void property="versionNumber">
   <int>2</int>
  </void>
 </object>
</java>
```

Content of CD

Folder Structure:

/software
    /FilterManagerTester
        FilterManagerTester.jar
        /images
            (various test images)
    /Pipeline Based Image Editing Demo Application
        PipelineDemo.jar
        /images
            (various test images)
    /JAlbum
/source code
    /FilterManagerTester
    /Pipeline Based Image Editing Demo Application
/documents
    Pipeline Based Image Editing with JAlbum – Bachelor Thesis – David Fichtmueller.pdf
        this document
    /images
        the images from this document
    /JavaDoc
        /FilterManagerTester
        /Pipline Based Image Editing Demo Application

Sources

[1] “Chronically Unemployed”. Gradual Undo. URL:
http://www.chronicallyunemployed.com/main_stuff/tech/undo.htm. [cited 10. July 2008]

[2] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[3] Chumbo Zhou and Atsumi Imamiya. Object-based nonlinear undo model. In *Proceedings of the 21st international Computer Software and Applications Conference* (August 11 - 15, 1997). pages 50-55. COMPSAC. IEEE Computer Society. Washington, DC, 1997.

Declaration of Authorship

I, David Fichtmüller, hereby declare that I created this Bachelor thesis and the work related to it independently and without illegitimate help. I did not use any other than the stated sources and resources.

Berlin, 15. July 2008 _____

(David Fichtmüller)