AN ABSTRACT OF THE PROJECT OF

Hien Le for the degree of Master of Science in Computer Science presented on June 14, 2007.

Title: Using Web Services for Image Processing in a Desktop Widgets Engine

Abstract approved: ______________________________

Ronald Metoyer

Desktop widget engines have emerged as an alternative for completing simple tasks without the need for a full-blown application or constant user interaction. Widgets can simply display data in a compact and visually appealing manner (such as stock tickers, weather forecasts, and news notifications), or go so far as to provide alternative interfaces to sites that expose their services via an API. This project aims to develop an image processing application where the key features are provided neither locally nor by a single host. The Yahoo! Widgets Engine is used to design a simple image processing interface where processing operations are mapped to widgets that can be connected together to create more complex operations. Actual data processing occurs remotely via calls to a REST-style Java web service to allow for a non-local and decentralized system.

©Copyright by Hien Le
June 14, 2007
All Rights Reserved

Using Web Services for Image Processing in a Desktop Widgets Engine

by

Hien Le

A PROJECT

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented June 14, 2007
Commencement June 2008

Master of Science project of Hien Le presented on June 14, 2007.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electric Engineering and Computer Science

---

Dean of the Graduate School

I understand that my project will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my project to any reader upon request.

Hien Le, Author

ACKNOWLEDGEMENTS

I would like to thank my committee members for their time today and over the past two years:

- Dr. Ron Metoyer
- Dr. Mike Bailey
- Dr. Eric Mortensen

TABLE OF CONTENTS

Page

1 Introduction 1
1.1 Related Work . . . 3
1.2 Overview . . . 5

2 Yahoo! Widgets Engine 8
2.1 Widget Format . . . 8
2.2 Capabilities . . . 9
2.3 Limitations . . . 11

3 Web Services with Java 13
3.1 WSDL and SOAP . . . 13
3.2 REST Services . . . 15

4 Services Implementation 17
4.1 Class Design . . . 17
4.2 Image Operations . . . 18
4.2.1 Mean Blur . . . 19
4.2.2 Gradient Magnitude . . . 19
4.2.3 Median . . . 20
4.2.4 Histogram Equalization . . . 20
4.2.5 Hue Shift . . . 21
4.2.6 Blend . . . 21
4.2.7 Alpha Masking . . . 21

5 Widget Implementation 25
5.1 Class Design . . . 25
5.2 Construction and Layout . . . 27
5.3 Messages and Events . . . 28
5.3.1 Connecting . . . 29
5.3.2 Executing . . . 29

TABLE OF CONTENTS (Continued)

Page

6 Conclusion 34

6.1 Summary . . . 34

6.2 Future Work . . . 34

Bibliography 35

LIST OF FIGURES

Figure		Page
4.1	Class diagram overview of how our image operations are exposed as services	23
4.2	Image blurring method with annotations	24
4.3	Publishing the image blurring method as a REST web service	24
5.1	Example flow network that blends two images (one of which will be blurred), then applies a blur to the results. The circles on the top edge represents input points, the color scheme is: gray = no input, yellow = widget input, green = file input.	26
5.2	Class diagram overview of our widget environment.	27
5.3	Messages sent and handlers activated when Widget A's input and Widget B's output are clicked (in that order) to create a connecting edge.	31
5.4	Messages sent and received to execute an entire chain of widgets. All inputs have been correctly set in this example and the network successfully executes to completion.	32
5.5	Messages sent and received to execute an entire chain of widgets. In this example an error occurs because one widget does not have all of its inputs set.	33

DEDICATION

To my fellow lab dwellers, may we meet again in another cubicle.

Chapter 1 – Introduction

This project is motivated by two emerging technology trends in the area of casual consumer computing:

1. The shift of common computing applications off of the desktop and onto a web-based environment.

2. The use of desktop widgets as a bridge between web-based and desktop-based computing.

We are interested in determining if graphics applications could utilize these two technologies. To this end, we develop an image editing environment that presents the client-side interface using desktop widgets that delegate to a web service to carry out the actual signal processing operations. Each image operation maps directly to one widget in the environment, and these widgets can be connected together in a dataflow manner for more complicated processing. We chose this particular domain with the hope that image processing operations are more conceptually accessible to a casual user than geometry or motion processing techniques. The remainder of this section details how our project tries to incorporate what we see as the key ideas of these technologies.

The shift to web-based applications is happening despite the ever-increasing amount of of raw processing power and storage space available on the modern

desktop. We see accessibility as the main motivation for users migrating to web applications, because the application is available even when their personal machines are not. Many web applications have taken the concept of portability one step further by exposing their core functionality to other applications as web services [10, 26]. To maximize portability and accessibility, the core signal processing operations of our application are provided by a remote host through a very simplistic set of HTTP-based calls.

With larger applications moving online it is surprising to see the increasing popularity of desktop widget engines, which potentially fill the local desktop with *small* applets that each have very limited scope and features. Desktop widgets are usually small enough to constantly occupy screen real estate without being obtrusive to the user's main applications. They serve mainly to display some form of constantly updated local or remote data. Since most widget engines provide network access (via HTTP) to retrieve information, widgets can also be used to call web services. The photo sharing service Flickr [7] exposes its functionality via an API[8] that enables its desktop widget[44] to retrieve photos for a slideshow and also act as a convenient drag-and-drop application for users to upload photos. Our project is implemented using the Yahoo! Widget Engine (YWE) [27].

While the move seems natural for network clients (such as e-mail and instant messaging[17]), mainstay personal computing applications (such as word processors[28] and spreadsheets[19]) also made surprisingly successful transitions to the web. The non-intensive computation and size requirements of office software and documents with respect to modern CPUs and high speed Internet connections

helped greatly in adapting them to remote environments. Could the image editor (another common desktop-centric application) undergo a similar transformation considering signal processing operators and raster image data typically require significantly more CPU cycles and storage space? Assuming server hardware at least as powerful as the client machine, the upload/download of data is the only difference in interactivity of remote versus local applications. If we define our casual users as those working with small images files (less than one megabyte in size) then the delays become manageable.

The abstraction of web applications into a publicly accessible web services layer provides additional customization opportunities for users when assembling their remote environment. Web services facilitate interoperability between web applications that were not designed for direct collaboration and additionally provide 3rd party applications (including desktop widgets) a means to create an alternative interface for a site. One of our project goals is to allow users to combine the image processing services of different providers. To accomplish this, we do not define a protocol for the services to talk to each other (choosing to keep their APIs minimal) and instead support service aggregation via simple message passing between the desktop widget front-ends.

1.1 Related Work

We did not aim to develop a novel user interface paradigm, choosing instead to mimic the interface of established graphics applications. Our early design im-

posed a single-input/single-output restriction on the processing nodes, which essentially imitates stack based computing models. While we ultimately abandoned this model (due to problems with representing multiple input operations such as blending and masking), it is successfully employed in commercial 2D and 3D applications. The latest incarnation of Adobe's Photoshop[1] refers to this as non-destructive Smart Filters for images, and AutoDesk's 3D Studio MAX[30] has a Modifier Stack system that allows sequential state-based operations on 3D models.

The paradigm we eventually settled on is commonly referred to as a data-flow execution environment[32], characterized as a visual program where data moves through a user-defined (potentially branching) pipeline of processing nodes. The open-source packages OpenDX[20] and VTK[23] use this kind of environment to facilitate fast construction of scientific visualizations. Apple Computer provides the Quartz Composer[24] application to simplify development of applications utilizing their Core Image[2] and Core Video frameworks. Professional 3D animation packages such as Maya[31] and Blender[4] employ node-based editors to manage the complexities of their materials and post-production compositing systems. While these systems allow nodes to be multi-input and multi-output, our current environment only employs single-output nodes.

Recent research into remote data processing has produced systems of varying scales. An early web-based prototype[39] let users specify a stack of image processing services to execute on remotely-sensed earth sciences datasets from the geographical image systems (GIS) domain. The Epoch project[43] enabled users to upload photos that were farmed out to its computing grid for the purpose of

3D reconstruction. Similarly, Green Building Studios[36] allows engineers working on commercial buildings to upload their CAD models and analyze a building's eco-friendliness. A more mature system, ImageGrid[45], focused on providing both efficient execution and storage of large-scale data sets. This system allowed designing flow networks, and used a layered service-oriented architecture (SOA) approach to efficiently schedule the execution of the flow networks on a highly parallelized C/MPI-based processing grid.

With regards to a commercial graphics web services standard, Sony Computer Entertainment has proposed COLLADA[29] as a common asset format for graphics applications. LEAD Technologies[16] sells both subscriptions and the server components for an image processing web service. Mental Image's RealityServer[18] provides an architecture where all aspects of 3D content creation occur on the server and clients interact with the content via streamed images and video. RealityServer allows full 3D content development from a web browser, but also allows 3rd party 3D packages to interface with it in a client role.

1.2 Overview

Our image processing environment has been designed with the following scenario as its primary use-case:

1. A user would like to edit an image but doesn't know how to use Photoshop, GIMP, or some other image manipulation package.
2. User loads the widgets representing the desired image operations into the

YWE, these could potentially be located by:

- going directly to the website of a known service provider and downloading their widgets
- using another widget that displays a gallery of registered image operation providers (essentially a toolbox widget)

3. Each operator's image inputs are set by dropping an image file onto the appropriate input node on the top-edge.
4. The image operators' settings are configured using the YWE's standard Preferences dialog or some other widget-specific custom user interface.
5. An output node of one widget is connected to the input node of another widget to create more complicated composite operations.
6. The user then executes either:
    - the last widget in the flow network to obtain final results
    - any other widget to see intermediate results for the purpose of fine tuning settings

A widget executes its associated image operation by sending the input image data and filter parameters to the web service as a standard HTTP POST request and receiving the resulting image in the same request. A widget's image inputs can either be a file or another widget (which will pass its output via a file). Every widget in the YWE runs in its own process (for stability reasons) but asynchronous

text messages can be sent between widgets. The widgets use this mechanism to request the execution of other widgets and to receive their output. This design simplifies service providers by keeping them oblivious to each other but requires the client-side to act an intermediary. As a consequence, the client must download and upload all intermediate values in the dataflow network.

The remote processing operations are executed by sending an HTTP POST request to the endpoint URL. These requests follow the standard MIME multi-part message format used by browsers for submitting form data and file uploads. Our operations also support two commands in the form of GET requests for the purpose of displaying progress in the widget. Currently we only monitor the progress of the upload because the operations are simple enough that the bottleneck of the remote call is the transmission of operands.

We do not use extensible markup language (XML)[41], SOAP[38], and web services description language (WSDL)[40] to implement and specify the server side of the image processing system, so our server side is not technically a web service as defined by W3C standards. Services that do not use these standards-based technologies are typically referred to as web APIS to distinguish this fact, but we will sometimes use the terms interchangeably. The decision to be non-standards based primarily had to do with the ease of building the clients (for a more in-depth discussion please see sections 2.3 and 3.1).

Chapter 2 – Yahoo! Widgets Engine

The Yahoo! Widgets Engine is a cross-platform (Apple Mac OS X and Microsoft Windows) runtime environment used for running small desktop applets referred to as widgets. Similar widget engines exist for a variety of platforms, such as: Dashboard[3] for OS X, SuperKaramba[22] for KDE, gDesklets[6] for GNOME, StarDock's DesktopX[21] for Windows, and the Java-based Glossitope[9]. After experimenting with the Google Desktop Gadgets[11] and Glossitope engines we selected YWE as our platform because of its maturity in terms of features, documentation, and availability of widgets for reference.

2.1 Widget Format

Widgets use a custom XML file format[15] to define their visual appearance and behavior, identifying metadata, and any preferences (such as user settings or window positions) to be persisted between sessions . A complete widget includes these configuration files as well as other resources such as images, sounds, and source files to scripts. Widget behavior is implemented in Javascript but there is the ability to use Component Object Model (COM)[5], AppleScript, or shell calls to access native code. Almost the entirety of the configuration file's Document Object Model (DOM) is available in Javascript, allowing for the entire widget layout to be

dynamically specified and modified. The Javascript runtime provides abstracted access to platform-specific details such as the file system, display resolution, hardware settings, and network access. Web developers used to working with Dynamic HTML should be able to easily transition to YWE as the event model is very similar to that of most browsers. It is helpful to think of YWE itself as a very compact browser (recent versions have even incorporated CSS into the layout engine) built for the purpose of hosting applets. Despite so many similarities to a web browser, the YWE DOM is not a direct copy of the HTML DOM.

2.2 Capabilities

Specific features of YWE that were integral to our image processing environment were support for: 2D vector graphics, runtime read-write of the widget's DOM, filesystem text IO, inter-widget message passing, and network access to HTTP resources.

Having access to the widget's layout at runtime and the ability to draw vector graphics helped us to avoid many of the tricks that usually have to be used by web designers to create specific user interface effects. We can create the glossy button appearance of our widgets in vector graphics simply by defining the path for a rounded rectangle and specifying a color gradient as the fill pattern. Supporting resizing of a similar button within a typical raster image setting would involve creating three separate images for the left edge, tileable center region, and right edge. The vector graphics approach also requires only one Canvas DOM element,

whereas the raster approach would require three Image DOM elements with one able to resize with the window. Runtime access to the DOM also allows us to programatically construct repeating UI elements in code instead having to create excessively large static XML configuration files.

Although each widget in YWE runs in its own process, we need inter-widget communication to enable construction and execution of our data flow network. YWE allows widgets to send text messages to other widgets that it knows by name, but there is currently no method provided by the platform for widgets to discover the names of other actively running widgets. We use these messages to allow one widget to invoke the methods of another widget, but the asynchronous and one-directional nature of this message passing system require a bit of work on our part to accept return values. In our system, these messages coordinate the connection, disconnection and sequential execution of widgets. To overcome the previously mentioned discovery limitation, all of our widgets use the filesystem text IO to create an on-disk registry that makes them visible to other widgets. There are potential race conditions if multiple widgets are simultaneously starting/exiting due to lack of file locking in YWE, but we have not encountered this situation in day-to-day usage.

Widgets are able to access the network via HTTP using either a YWE-specific URL object or a recently added XMLHttpRequest[42] object similar to that found in most web browsers. These objects allow for HTTP GET and POST requests, which in turn allow users to both retrieve data from and send data to web services. Both synchronous and asynchronous network access is supported and we make use

of both in our application. We use asynchronous POST to support sending larger images to a web service without locking up the widget UI, and utilize synchronous GET to check on the progress of those file uploads.

2.3 Limitations

The biggest limitation we encountered with the YWE is its lack of support for handling of binary data in Javascript and performing binary file system IO. File IO support currently only deals with text strings and the only way to write binary data (such as image data from the web) is to set the URL object to redirect its results to a file. Attempts to use the file system APIs to read a binary file typically results in only the first few printable characters of the file being read. Support for binary IO was vital to our initial plan of implementing standards-based web service using SOAP calls.

Further details about SOAP will be provided in section 3.1 so we will only describe the YWE-related issues here. SOAP is the format of the XML messages used to invoke remote methods and receive return values in a standards-based web service. There is support in YWE for constructing and parsing XML so there is no need for text scraping with unsightly regular expressions, but lack of binary support limits our ability to send and receive images with XML. It is possible to send binary data directly in XML if it has been base64 encoded to avoid introducing errors to the XML parser but, as previously mentioned, directly manipulating binary data is not possible in YWE (without the use of platform-

specific shell calls). While YWE's URL interface supports sending binary files, we have no control over the construction of the multi-part MIME message body and therefore cannot create proper SOAP attachments. It is primarily for these reasons that we avoided creating our image operations as standards-compliant web services.

One final limitation that is easily worked around is the need for every widget to be identified by a unique ID. This allows widgets to communicate and to launch widgets required for collaboration. As a side effect of this, the runtime will only allow one instance of a widget to be active. If an operation is to be used twice in a network then there must also be two copies of the widgets on disk with different IDs in their configuration file. Currently a user would either have to use the YWE tools to manually modify and duplicate their widgets, or service providers will need to provide a set of identical widgets for download with slightly mangled names.

Chapter 3 – Web Services with Java

The Java Platform has included support for remote method invocation (RMI) since the earliest versions of the Standard Edition distribution, but support for integration with software written in other languages has mainly been a feature of the Enterprise Edition platform. With the version 6 release of the Java platform, many of the facilities for distributed computing from the Enterprise Edition is available in the Standard Edition. The next sections will explain these features of JAX-WS 2.0 in Java 6 and our attempts to use (or reasons for not using) them in this project.

3.1 WSDL and SOAP

A true web service is built primarily on two XML-based language standards managed by the World Wide Web Consortium (W3C) [25]: web services desription language (WSDL) and SOAP (which is no longer an acronym). A WSDL file is analagous to the C/C++ concept of a header file in that it describes the public interface of a web service. Communication between the client and web service provider is in the form of SOAP messages. The Java Enterprise Edition platform has long provided tools (such as *wscompile*) and libraries to automate creation of these files and messages, and starting with the version 6 release these tools are also

available in the Standard Edition.

The *wscompile* tool takes the public interface of a Java class (and any dependent classes) and generates the WSDL file that can be used by clients to generate their platform-specific client stubs. The WSDL file uses XML to describe the web service's method signatures and data structure layout in terms of primitives common to most programming languages. Invocation parameters and return values are transferred between client and server using messages in another XML format called SOAP. By using Java Annotations[13] and the light-weight HTTP server API added in Java 6, it is possible to host web services without touching Servlets[12], Enterprise Java Beans[14], or the wscompile tool. Annotations were added in Java 5 to allow adding metdata to source coce that could be accessible at both compile-time and runtime (see our particular usage in section 4.1). Annotations are used extensively by Java's web services framework to allow tooling to automatically add functionality to classes during the compilation process.

Ideally the client and server platforms will transparently perform serialization when passing objects, and neither the client or server developer has to write code that is aware of the remote nature of its data. This automatic data serialization would be especially useful if both the server and client code base make extensive use of fairly complicated classes/objects and pass them between each other. However, the widget clients in our system will pass only a few numerical parameters to the remote image operations and only receives one piece of image data in return. Our numerical parameters are not encapsulated as classes because they are neither used locally nor complicated enough to benefit from auto-serialization. Likewise, the

image data is not wrapped as a class (or XML) because binary data is inaccessible at the Javascript level. Other client limitations discussed earlier (see section 2.3) would require us to write an additional non-WSDL/SOAP layer that translates messages between the client and services. For these reasons we decided to expose the image processing operations using the service architecture described in the next section. Conveniently, Java 6 allows us to develop services in this alternative fashion simply by using the previously mentioned annotations to request direct access to the lower level details of the HTTP message (such as request type, form data, and query strings).

3.2 REST Services

We design our services around the the Representational State Transfer (REST) architecture described by Fielding[33] for web based applications. Our services are considered RESTful because:

1. Every operation provided exists at a unique URL as opposed to a single endpoint for W3C web services, e.g., the image blurring service is invoked at *http://localhost/blur* rather than sending an XML message containing the verb 'blur' to *http://localhost/services*.

2. Invocation occurs using standard HTTP methods without a wrapper such as SOAP, e.g., parameters are passed in the form of GET queries or POST form fields and results are also returned without wrapping.

3. The parameters in a method invocation contain enough context to allow the server to be stateless, e.g., a user checks on the progress of an upload with ticket number 111 at *http://localhost/blur/getProgress?key=111* rather than just *http://localhost/blur/getProgress*, which would require some state data on the server to infer the ticket number from the IP address of the request.

In addition to the actual image processing operation, our services allow clients to check on the status of the file upload (if they made an asynchronous call to the operation) by obtaining an upload ticket beforehand. The notion of an upload ticket is not based on any particular standard but we have seen it offered as part of other web APIs that deal with large files. For a service located at *http://localhost/blur*, the order of calls by the corresponding widget would be:

1. Access *http://localhost/blur/getProgressKey* via a GET request to generate a ticket number that can be used to check on the upload status.

2. Asynchronously call *http://localhost/blur?key=ticketNum* with a POST request that passes the images and filter parameters. The upload ticket is passed as a query string for simplicity.

3. Periodically poll *http://localhost/blur/getProgress?key=ticketNum* with a GET request to find out the percentage of image data read by the server.

Chapter 4 – Services Implementation

We have designed our backend to minimize the amount of new code a developer has to write in order to publish a pre-existing code as a web service. The method to be published is only required to use our custom annotations (so there are no real changes to the code) to allow automatic extraction and parsing of parameters from the HTTP request.

4.1 Class Design

As the UML Class diagram in figure 4.1 shows, the *AbstractService* class provides much of the basic functionality that allows the raw image operation code to be decoupled from the HTTP-related code. By default this class will process all GET requests, and respond only to the *getProgressKey* and *getProgress* commands that enable file transfer monitoring. This class utilizes a modified version of the Apache FileUpload package to extract parameters and files from POST requests. There is no other default functionality implemented for POST requests, so AbstractService must be sub-classed to be used.

The *ProgressKeyManager* is used to map the previously mentioned upload ticket numbers to the current progress of the upload. These tickets are randomly generated upon request by AbstractService and completed tickets are periodically

removed by an automatic cleaner thread. Since we do not implement any sort of error status reporting in this prototype, the default response when a non-existent ticket's progress is requested is that it is "more than done".

SimpleService is constructed with a reflexive reference to any public static method and makes that method available as a web service. We define two custom runtime-accessible Java Annotations *IntParam* and *ImageParam* that are used to export the parameter names that are otherwise lost during compilation. Additionally, *IntParam* also allows the specification if minimum, maximum, and default values of integer parameters for validation purposes. The annotations allow us to programmatically match up the fields/files in an HTTP POST request with the published method's parameter list. This essentially replaces some of the auto-serialization functionality lost by not using the WSDL/SOAP approach.

In this diagram, the *Service* is simply a dummy place holder that shows where web services can be created. The dependency arrows signify that new web services can be created by wrapping an image processing operation (with annotations) inside a *SimpleService* object. Figures 4.2 and 4.3 show how to annotate a method and then publish it as a web service.

4.2 Image Operations

We have implemented a few basic image processing operations that might be of interests to users who are trying to correct flaws in their digital photos. For a more in-depth discussion of each operation, please refer to any standard text on image

processing[34].

4.2.1 Mean Blur

We implement the image blurring operator as the mean of the neighborhood pixels and allow the user to specify the radius of the neighborhood. For the sake of speed we perform two 1-D convolutions with kernels of height/width $(2r+1)$ rather than a single 2-D convolution with a $(2r+1)\times(2r+1)$ kernel. This operator can be used to remove small amounts of noise from an image, but in general will also remove fine details and cause very noisy points to bleed into the neighborhood.

4.2.2 Gradient Magnitude

Areas of rapidly changing intensity in an image typically correspond to the edge of objects so the gradient magnitude operator is useful for highlighting edges. We approximate the gradient magnitude from the partial derivatives in the X and Y direction obtained using the Sobel filter. The operator probably has no direct applicability to a casual user trying to fix some photos, but we've included it nonetheless because it produces interesting images that may be useful for creating artistic effects.

4.2.3 Median

The median operator is very effective at removing salt-and-pepper noise and at smaller kernel sizes does not cause as much loss of detail as a blurring filter. When using larger kernel sizes the effects of the median operation resemble smudging more than blurring. Whereas a straightforward comparison or histogram sort would have $O(r^2 logr)$ or $O(r^2)$ complexity with respect to the neighborhood radius, we use a sliding window histogram sort[35] that only requires $O(r)$.

4.2.4 Histogram Equalization

In images with poor dynamic range, the equalization operator attempts to stretch out the brightness histogram so that it spans the entire available range. Discrete histogram equalization is a point operation that determines a pixel's new intensity based on the amount of total pixels in the image with equal or lesser intensity, according to the formula:

$$f(i) = \frac{1}{A} \sum_{j=min}^{i} p_j$$

where A is the total number of pixels in the image, and p_j is the number of pixels with intensity j.

4.2.5 Hue Shift

This operator allows changing the color of all objects in an image while for the most part preserving lights and shadows. Since most file formats store images as RGB values, the first step of hue-shifting involves conversion into HSV space where hue and brightness are separated into channels that can be independently processed. An add-and-mod point operation is applied to all pixel values in the hue channel before converting back to RGB space for display.

4.2.6 Blend

This operator is different from the ones we have seen so far because it takes two image inputs instead of one. Two images are combined according to a user-provided alpha value that dictates which image will be predominantly preserved. The standard linear blending formula $p_{xy}(o) = \alpha p_{xy}(i_0) + (1 - \alpha)p_{xy}(i_1)$ is used, where $p_{xy}(i)$ is the value of the pixel (x, y) in the image i. This operation can be used for overlaying two different images into a montage or merging a filtered version of an image with the original to achieve more subtle filtering.

4.2.7 Alpha Masking

This is an advanced version of the blending operator. Instead of using two images and a blending constant, the constant is replaced with a grayscale image that allows a different alpha value to be specified for every pixel in the output. When

applied to two different images this has the effect of cutting and pasting portions of one image into the other. If the inputs are an image and its filtered version then the alpha mask essentially restricts the filter's effects to the specified areas of the image.

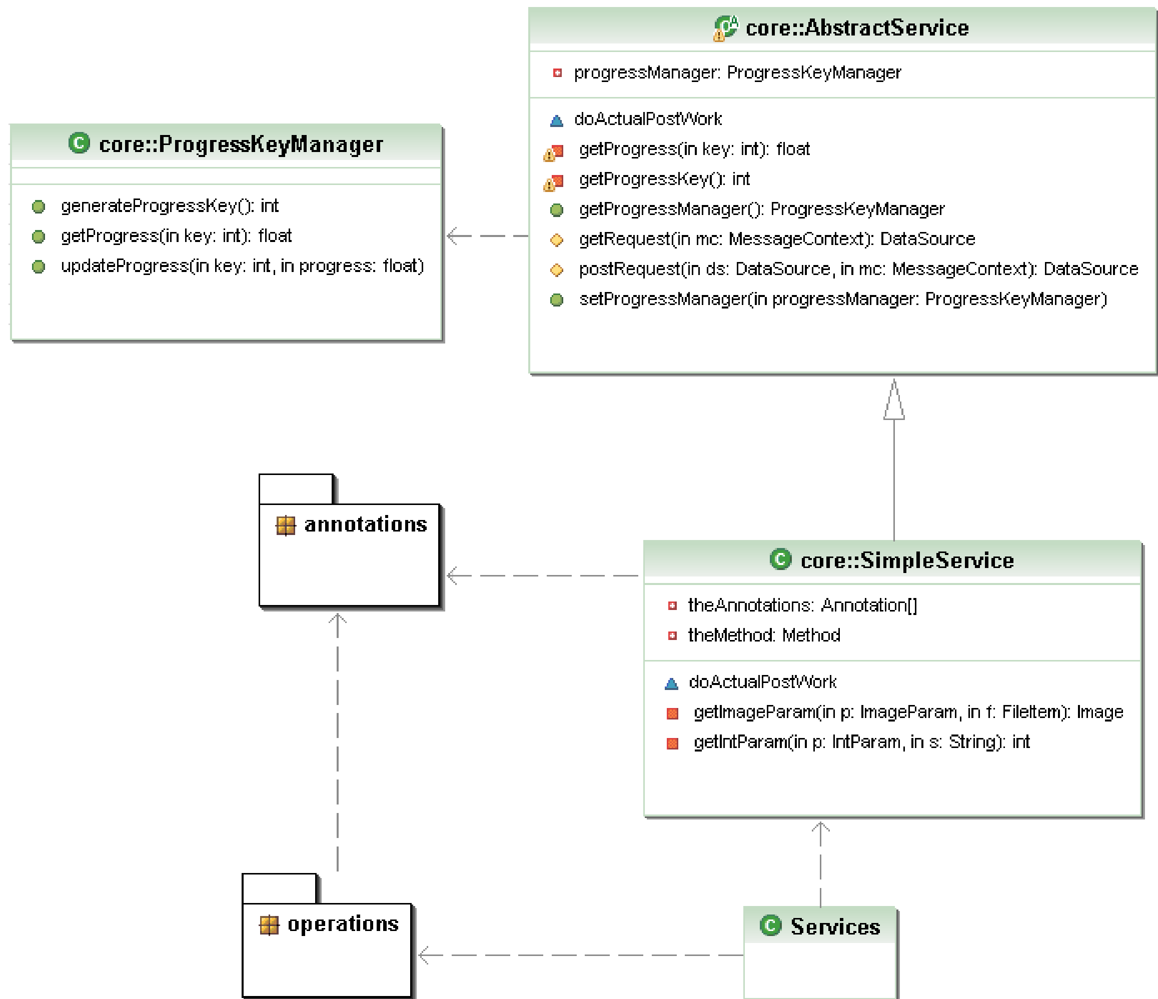

Figure 4.1: Class diagram overview of how our image operations are exposed as services

```
public class Blur {
  public static BufferedImage blur(
    @IntParam(name = "radius", min = 1, max = 16, def = 2)
    int radius,
    @ImageParam(name = "input0")
    BufferedImage input0) {
      ...
  }
}
```

Figure 4.2: Image blurring method with annotations

```
public static void main2(String[] args)
throws SecurityException, NoSuchMethodException {
  // use Reflection to get a reference to the method
  Method m = Blur.class.getMethod("blur", Integer.TYPE,
   BufferedImage.class);

  // wrap the method up so it can take parameters via HTTP
  AbstractService s = new SimpleService(m);

  // start the new service
  Endpoint e = Endpoint.create(HTTPBinding.HTTP_BINDING, s);
  e.publish("http://127.0.0.1/blur");
}
```

Figure 4.3: Publishing the image blurring method as a REST web service

Chapter 5 – Widget Implementation

Figure 5.1 shows an example of a filter chain created by connecting individual desktop widgets in our environment. The blur and blend widgets each call a different web service to carry out their image operation. The image operation is carried out by double clicking on a widget which will also execute all its input widgets as necessary, but only the widget that was double clicked will display its results.

5.1 Class Design

The class diagram shown in 5.2 represents the primary classes and interactions in our system. The base data structures are:

- *FileWrapper* - indicates that one of the widget's inputs is an on-disk file
- *WidgetWrapper* - indicates that one of the widget's inputs is another widget
- *WidgetIOPair* - refers to a widget and one of its specific inputs or output
- *IOPairEdge* - represents an edge connecting the input/output point of two widgets

The *ActiveRegistry* and *ScreenRegistry* are singleton classes used to coordinate sequential execution and drawing arcs between connected widgets. At the moment

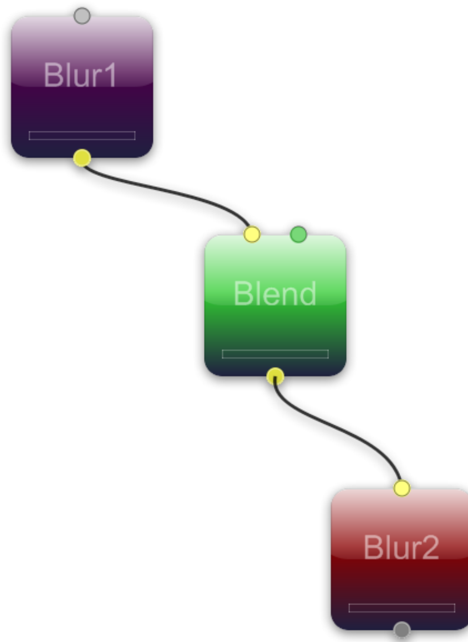

Figure 5.1: Example flow network that blends two images (one of which will be blurred), then applies a blur to the results. The circles on the top edge represents input points, the color scheme is: gray = no input, yellow = widget input, green = file input.

of load/unload all widgets register/unregister themselves with an on-disk file using the *ActiveRegistry* which also provides broadcast capability for sending messages. The *ScreenRegistry*'s purpose is track the position of all widgets and input/output nodes so that the edges drawn between them are correctly refreshed whenever one widget is moved. The *Controller* class is the primary user of *ActiveRegistry*, employing it to make sure all other widgets are aware of each other's special UI events (clicking on a node, or dragging the widget). It is also the *Controller*'s responsibility to keep the *ScreenRegistry* up to date so that the *View* can correctly resize the connector edges. *Model* is a rather uninteresting class so we will not

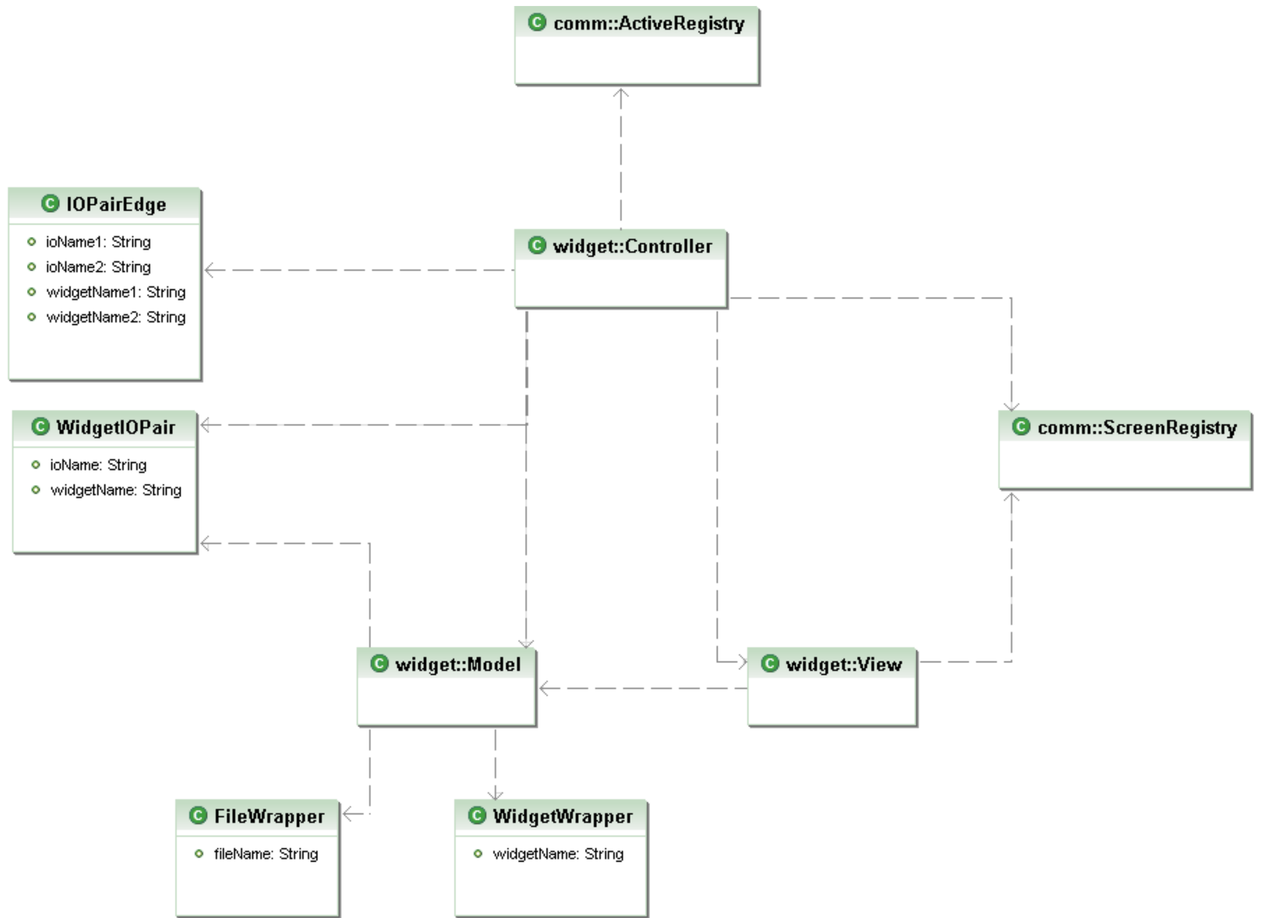

Figure 5.2: Class diagram overview of our widget environment.

offer a more in-depth discussion than to say it is primarily used to keep track of input/output connections and to make the actual web service calls.

5.2 Construction and Layout

Almost every graphical element composing the widget is created dynamically by JavaScript with the main window being the only object in the XML configuration

file (for the sole purpose of having the engine automatically save its last known location). Additionally, information specific to the web service, such as the URL and parameters, is stored in this configuration file as user preferences to allow decoupling of the Javascript from any specific image operation. We use YWE's ability to have hidden, non-persisted preferences to store a web service's specific parameters. Currently the 'file selector' preference type is used to represent images and slider preferences map to integer parameters. This allows the *Model* code to be re-used in any widget's remote procedure calls as long as the preferences are correctly setup.

Every widget is constructed as two YWE Windows each containing a Canvas element that allows for vector drawing (we do not use bitmaps in the UI). Users are able to directly manipulate the main window that contains the body of the widget along with the top and bottom-edge nodes used for defining inputs and connections. The second window is used to show the bezier curves representing connections, and will automatically move and resize to fill the space between widgets.

5.3 Messages and Events

All event handling (including message passing and receiving) is handled by *Controller*, making it by far the largest class in the system. We will describe here only the events and actions associated with connecting two widgets and executing a chain of widgets, as the rest of the UI event handling operates in a similar manner.

5.3.1 Connecting

Connecting two widgets involves clicking on the input/output node on one widget then on those of another widget. Once this occurs, both widgets need to be aware that an edge has formed between them. Since each widget is its own process, we use the textual message passing mechanism to coordinate this effort. When a widget receives a click it broadcasts an *ACTIVATE* message to all widgets (including itself) indicating the name of the widget and input/output node that was activated. Since all of our widgets execute the same event handler code, the broadcast has the same effect as if there was only one runtime and one handler overseeing all widgets. The first *ACTIVATE* simply causes that widget/node pair to be recorded as a potential endpoint. When the second *ACTIVATE* arrives then each widget checks to see if it was involved in either event, if not then it forgets that any of those events occurred. The widgets involved in the connection will set the input/output references correctly in its *Model* object (see Figure 5.3).

5.3.2 Executing

When a widget is double clicked to request execution, it will first check to see that all of its inputs are set (either to a file or another widget) or else it raises an error. Widgets with only file inputs will execute the web service call immediately, while widgets depending on other widgets for input will send *EXECUTE* messages to their dependencies. The messages will be recursively passed up the chain until either all dependencies are resolved or one widget returns an *ERROR* message. A

dependency is resolved when one widget is able execute its web service call and pass the temporary location of the resulting file to its output widget using the *RETURN* message. Upon receiving a *RETURN* message a widget will either try to call its web service or continue waiting for more messages. Each widget passes its result down to its output widget until eventually the bottom-most widget is reached and the final results displayed to the user in the native system editor. In the event of a web service error (bad input or server error) or widget error (missing inputs), the *ERROR* messages trickle down to the bottom-most widget in the same manner as a normal result. When the root node is notified of the error it recursively sends a *CANCEL* to itself and all widgets above it. This serves to abort any currently active or pending web service calls so as to not perform any unneeded work. Figures 5.4 and 5.5 illustrate the success and error scenarios for the network shown in Figure 5.1 .

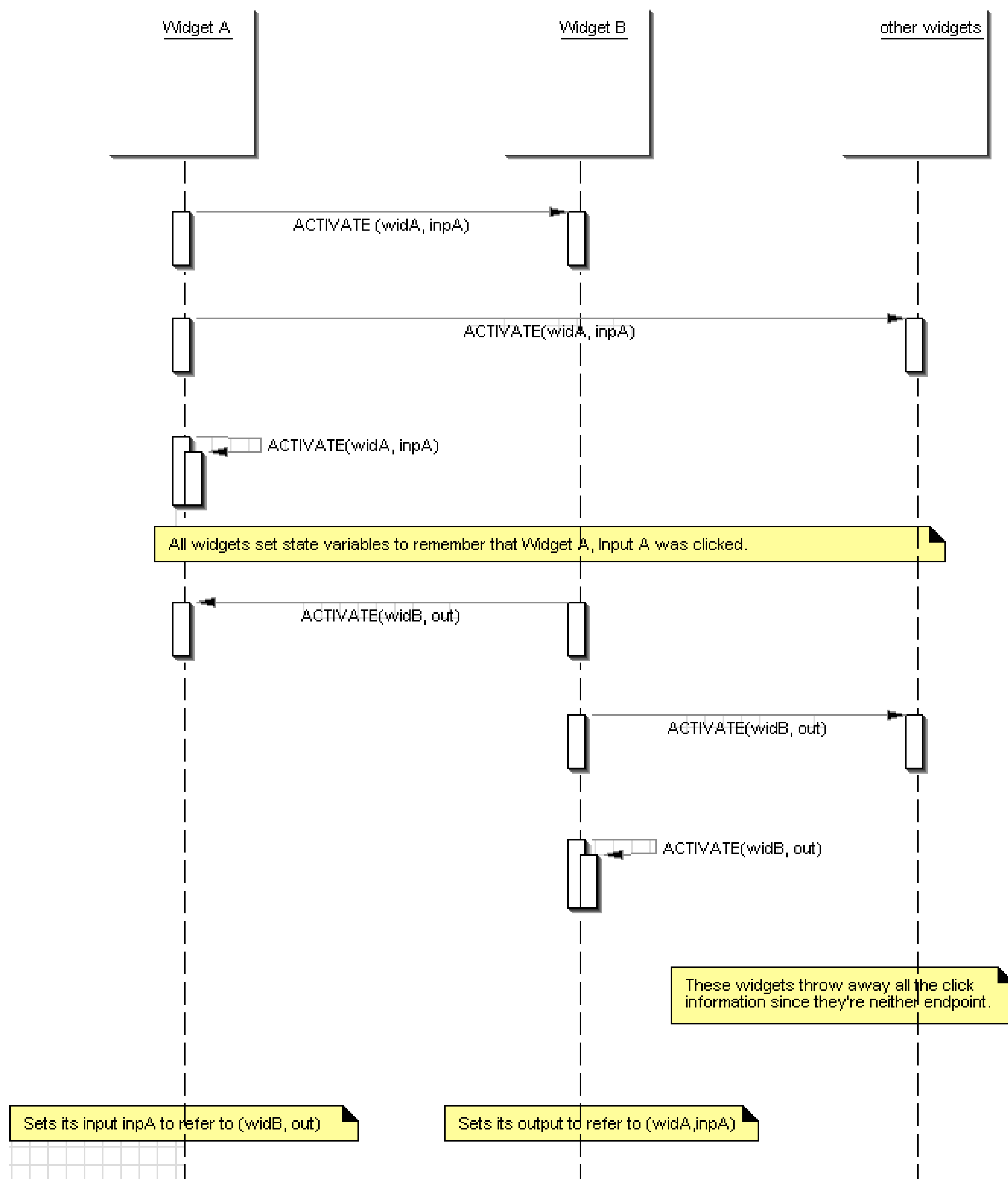

Figure 5.3: Messages sent and handlers activated when Widget A's input and Widget B's output are clicked (in that order) to create a connecting edge.

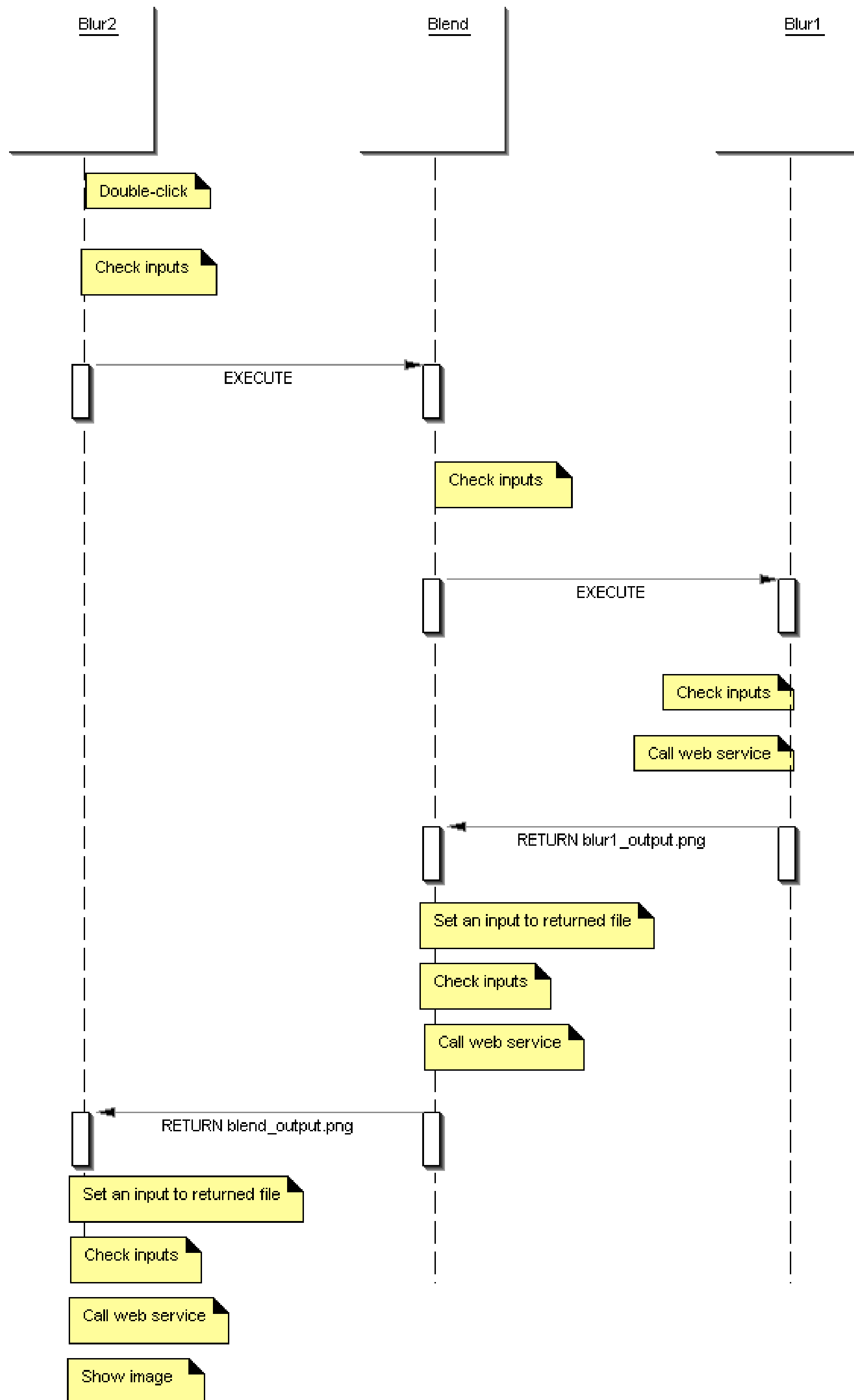

Figure 5.4: Messages sent and received to execute an entire chain of widgets. All inputs have been correctly set in this example and the network successfully executes to completion.

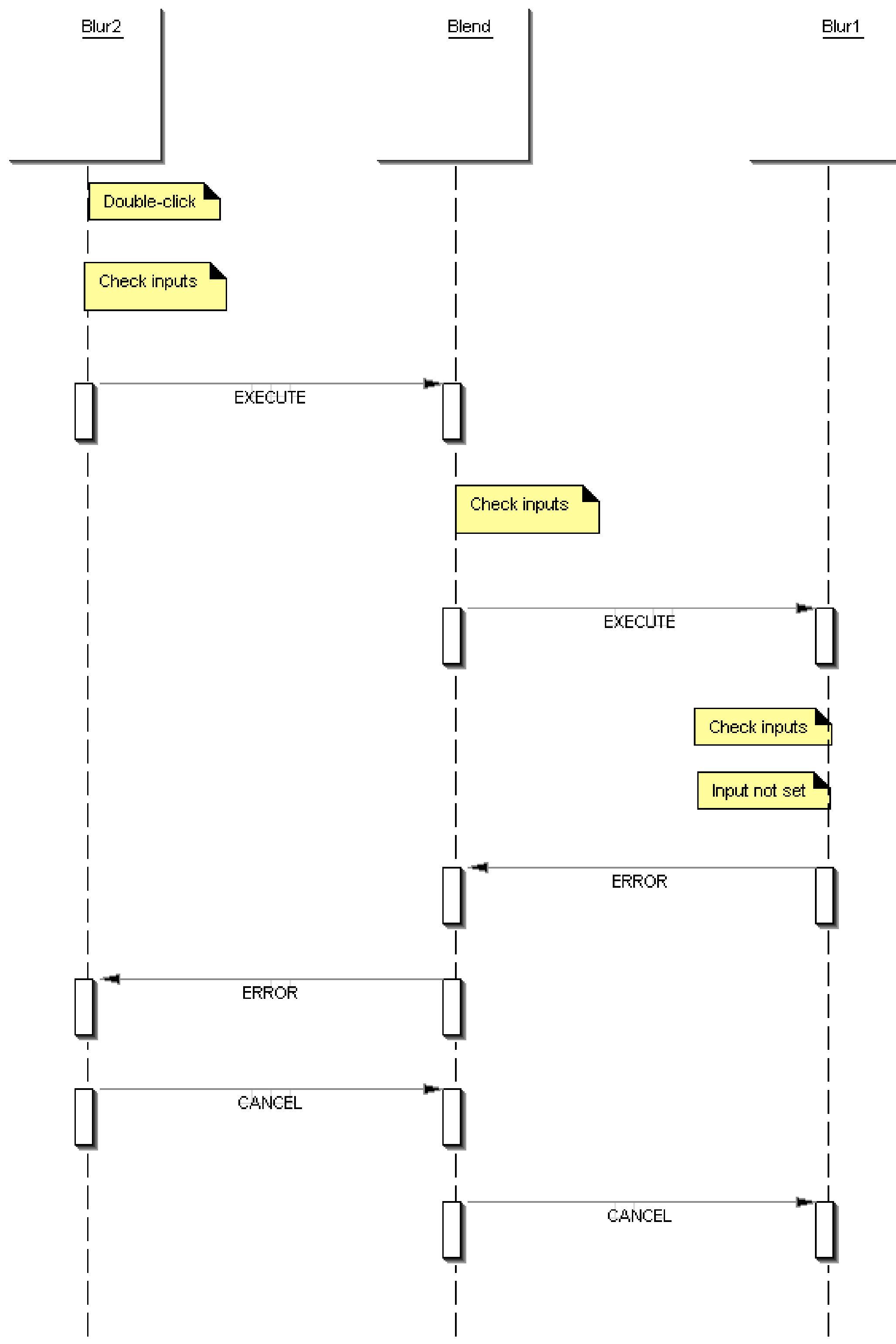

Figure 5.5: Messages sent and received to execute an entire chain of widgets. In this example an error occurs because one widget does not have all of its inputs set.

Chapter 6 – Conclusion

6.1 Summary

We have presented a system that allows the user to custom assemble a very simple (and hopefully intuitive) environment for processing digital images. While we have only implemented some very simple operations, the system is extensible enough that it could be extended to rival other online photo editing services. Our proof of concept has shown that with access to the right web services a widget can perform almost any task, even those that the widget runtime is ill-suited for (binary data processing in this case). Towards a larger purpose, this project has shown that it may be possible to expose casual users to other forms of graphics data processing (such as geometry and motion). Such widgets will achieve more relevance in the coming years when even casual users may have the need for customizing motion and 3D content.

6.2 Future Work

Some of the more basic tools found in most desktop image editing packages are missing from our prototype. Selection, painting and transformation tools are absent and not easily mapped to numeric sliders or the other built-in UI objects of the YWE. These tools require that the user be able to see the image to edit, so a

direct mapping to the dataflow network is unclear since operands may not always be available.

We mentioned earlier that the runtime does not allow more than one running instance of a widget. For most users it doesn't make sense to have multiple running instances of a widget but in a dataflow network this could be a very common use case. In our environment there would be conflicts when sending messages between widgets based only on a static ID. This limitation can be worked around if the user or provider creates copies of the widget with slightly modified IDs, but it would be interesting to see if the annotations we already use for parameter binding in the *SimpleService* class provide sufficient information for creating another service that would generate widgets (with unique/random IDs) on request. This feature would also free developers from having to create the widget XML files, further simplifying the process of creating new image processing services from existing code.

Finally, there are still the previously mentioned areas of graphics processing left to explore. While there is currently no cross-platform way to view motion files or meshes in the YWE, it would be interesting to explore the possibility of writing an OpenGL API on top of the Canvas object. This is not too far fetched considering successful prototypes of simple ray casting engines have been built using the Canvas[37]. Another possibility is to look into packing platform-specific binaries into the widgets to enable access to native 3D capabilities.

Bibliography

[1] Adobe Photoshop CS3 editions.
http://www.adobe.com/products/photoshop/index.html.

[2] Apple - Mac OS X - Core Image.
http://www.apple.com/macosx/features/coreimage/.

[3] Apple mac os x - dashboard. http://www.apple.com/macosx/features/dashboard/.

[4] Blender. http://www.blender.org.

[5] Component Object Model Technologies. http://www.microsoft.com/com/default.mspx.

[6] Desktop eyecandy. http://www.gdesklets.org.

[7] Flickr. http://www.flickr.com.

[8] Flickr services. http://www.flickr.com/services.

[9] Glossitope. http://www.glossitope.org.

[10] Google code. http://code.google.com.

[11] Google Desktop Gadgets. http://desktop.google.com/plugins/.

[12] JSR 154: Java Servlet 2.4 Specification.
http://www.jcp.org/en/jsr/detail?id=154.

[13] JSR 175: A Metadata Facility for the Java Programming Language.
http://www.jcp.org/en/jsr/detail?id=175.

[14] JSR 220: Enterprise JavaBeans 3.0.
http://www.jcp.org/en/jsr/detail?id=220.

[15] Konfabulator 4.0 reference manual. http://widgets.yahoo.com/manual/.

[16] LEADTOOLS Web Services. http://www.leadtools.net.

[17] Meebo. http://www.meebo.com/.

[18] Mental images reality server.
http://www.mentalimages.com/2_3_realityserver/.

[19] Num sum - web spreadsheet. http://numsum.com/.

[20] OpenDX. http://www.opendx.org.

[21] Stardock DesktopX - Build a better desktop.
http://www.stardock.com/products/desktopx/.

[22] SuperKaramba. http://netdragon.sourceforge.net.

[23] The Visualization Toolkit. http://www.vtk.org.

[24] Working with quartz composer.
http://developer.apple.com/graphicsimaging/quartz/quartzcomposer.html.

[25] World Wide Web Consortium. http://www.w3.org.

[26] Yahoo! developer network. http://developer.yahoo.com.

[27] Yahoo! widget engine. http://widgets.yahoo.com.

[28] Zoho Writer. http://writer.zoho.com.

[29] Rémi Arnaud and Mark Barnes. *COLLADA - Sailing the Gulf of 3D Digital Content Creation*. A K Peters Ltd, 2006.

[30] Autodesk. 3ds max. www.autodesk.com/3dsmax.

[31] Autodesk. Maya. http://www.autodesk.com/maya.

[32] Philip T. Cox, F. R. Giles, and T. Pietrzykowski. Prograph: A step towards liberating programming from textual conditioning. In *Proc. IEEE Work. Visual Languages, VL*, pages 150–156, Los Alamitos, California, 4–6 October 1989. IEEE CS Press.

[33] R. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of Califormia, Irvine, USA, 2000.

[34] R. C. Gonzalez and R. E. Woods. *Digital Image Processing, Second Edition*. Prentice Hall, 2002.

[35] T. S. Huang, G. J. Yang, and G. Y. Tang. A fast two-dimensional median filtering algorithm. In *International Conference on Pattern Recognition and Image Processing*, pages 128–30, 1978.

[36] Green Building Studios Inc. GBS Web Service. http://www.greenbuildingstudio.com/gbsinc/gbs-web.aspx.

[37] Benjamin Joffe. Canvascape - 3D Walker. http://www.abrahamjoffe.com.au/ben/canvascape/.

[38] Anish Karmarkar, Henrik Frystyk Nielsen, Marc Hadley, Jean-Jacques Moreau, Noah Mendelsohn, Martin Gudgin, and Yves Lafon. SOAP version 1.2 part 1: Messaging framework (second edition). W3C recommendation, W3C, April 2007. http://www.w3.org/TR/2007/REC-soap12-part1-20070427/.

[39] Ken Keiser, Rahul Ramachandran, John Rushing, Helen Conover, and Sara J. Graves. Distributed services technology for earth science data processing. *American Meteorological Society's (AMS) 19th International Conference on Interactive Information Processing Systems (IIPS) for Meteorology, Oceanography, and Hydrology*, 2003.

[40] Canyang Kevin Liu and David Booth. Web services description language (WSDL) version 2.0 part 0: Primer. W3C proposed reccommendation, W3C, May 2007. http://www.w3.org/TR/2007/PR-wsdl20-primer-20070523.

[41] Eve Maler, François Yergeau, C. M. Sperberg-McQueen, Tim Bray, and Jean Paoli. Extensible markup language (XML) 1.0 (fourth edition). W3C recommendation, W3C, August 2006. http://www.w3.org/TR/2006/REC-xml-20060816.

[42] Anne van Kesteren. The XMLHttpRequest object. a WD in last call, W3C, February 2007. http://www.w3.org/TR/2007/WD-XMLHttpRequest-20070227/.

[43] Maarten Vergauwen and Luc Van Gool. Web-based 3d reconstruction service. *Machine Vision Applications*, 17:411–426, 2006.

[44] Yahoo! Flickr widget. http://widgets.yahoo.com/gallery/view.php?widget=41574.

[45] Ran Zheng, Hai Jin, Qin Zhang, and Ying Li. Workflow-based remote-sensing image processing application in ImageGrid. In *PDCAT*, pages 390–394. IEEE Computer Society, 2005.