processWave.org Editor Architecture and Deployment

processWave.org Editor Architektur und Bereitstellung

Author
Martin Kreichgauer

Prof. Dr. Mathias Weske
MSc. Alexander Großkopf
MSc. Andreas Meyer

Lehrstuhl für Business Process Technology

Datum der Abgabe: 25. Juni 2010

I hereby affirm that I have written this bachelor's thesis independently, without using any sources other than those stated.

Potsdam, 25th June 2010

Martin Kreichgauer

Ich erkläre hiermit, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Potsdam, 25. Juni 2010

Martin Kreichgauer

The processWave.org Editor is a graphical modeling editor for Google Wave which allows multiple people to edit the same diagram at the same time. It is built on the open source modeling platform Oryx and supports a variety of modeling languages. This bachelor's thesis gives an overview to the architecture of the application and describes its deployment process. It therefore is an introduction for new developers who want to continue development of the processWave.org Editor.

Der processWave.org Editor ist ein graphischer Modell-Editor für Google Wave, der es ermöglicht, mit mehreren Personen gleichzeitig an demselben Diagramm zu arbeiten. Die Anwendung ist eine Weiterentwicklung der Open-Source-Modellierungsplattform Oryx und unterstützt mehrere Modellierungssprachen. Diese Bachelorarbeit beschreibt die Architektur und den Bereitstellungsprozess des Editors. Sie ist als Ausgangspunkt für weitere Entwicklungen am processWave.org Editor gedacht.

Contents

List of Figures

1 Introduction

The processWave.org Editor is a graphical modeling editor that allows multiple people to create and edit diagrams at the same time. It uses the source code of Oryx, an open source modeling platform, and currently supports six modeling languages from the Business Process and Software modeling area: The Business Process Modeling Notation (BPMN) [1] and its subset Simple BPMN, Unified Modeling Language (UML) class diagrams [2], Fundamental Modeling Concepts (FMC), Event-Driven Process Chains (EPC) and Petrinets.

The application is realized as an extension to Google Wave, a collaboration and communication tool that runs in the browser, which Google announced in 2009. [3] Together, the processWave.org Editor and Google Wave provide an integrated platform for graphical model creation and discussion, which aims at very quick and easy iterations on models.

This Bachelor's thesis should give you an introduction to the architecture of the processWave.org Editor, so new developers get started with the application's source code and deployment. Additionally, developers with experience in Oryx development get an understanding of how the processWave.org Editor's code base differs from that of Oryx.

The thesis first gives an overview of Google Wave, its architecture and how it can be extended by third-party developers. Then there is a short introduction to the Oryx architecture in section 3, in order to highlight the most important parts of the application that had to be mofified. Where applicable, there are references to other theses of the project that provide more detailed descriptions of the changes and newly added features. Afterwards, I explain the architecture of the processWave.org Editor in section 4 and how it can be deployed in section 5. In the conclusion the status of the application and the project is summarized.

2 Overview of Google Wave

Google Wave is an online collaboration platform that allows multiple people to work on an artifact, for example a letter, a piece of code or a mind map, at the same time. It was first introduced as a developer preview at the Google I/O conference in 2009, and released to the public one year later. [4]

With Google Wave users can have threaded conversations called *waves*, which Google describes as "equal parts conversation and document". [5] Inside a wave all participants can reply and edit everywhere in all messages. While they type every single keystroke is transmitted to all other participants in real-time. Users can also add collaborative applications, called gadgets, or automated participants, robots, to a wave. This makes Google Wave an interesting platform for all sorts of collaboration tasks, not just text editing.

In the following sections, I explain the most important concepts and terms behind Google Wave. Afterwards, I introduce different ways how third-party developers can extend Google Wave with their own applications, like we did with the process-Wave.org Editor.

2.1 Architecture

Waves are often described as "hosted documents" [6]: Unlike email conversations, which are sent back and forth between different mail servers, a wave is hosted on a server. When you edit a wave, the wave server sends a copy of it to the wave client running in your browser. Your wave client transmits all operations you made to the wave back to the server. Concurrent edits by multiple people are merged by the server into one consistent state using an algorithm called Operational Transformation (OT). [7] The resulting version of the wave is in turn pushed back to all clients.

This does not mean that access to a wave is limited to users of a single provider and its wave server. As figure 1 shows, wave servers of different providers can communicate with each other using the Wave Federation Protocol, an extension to the Extensible Messaging and Presecence Protocol (XMPP) [8]. This enables wave users

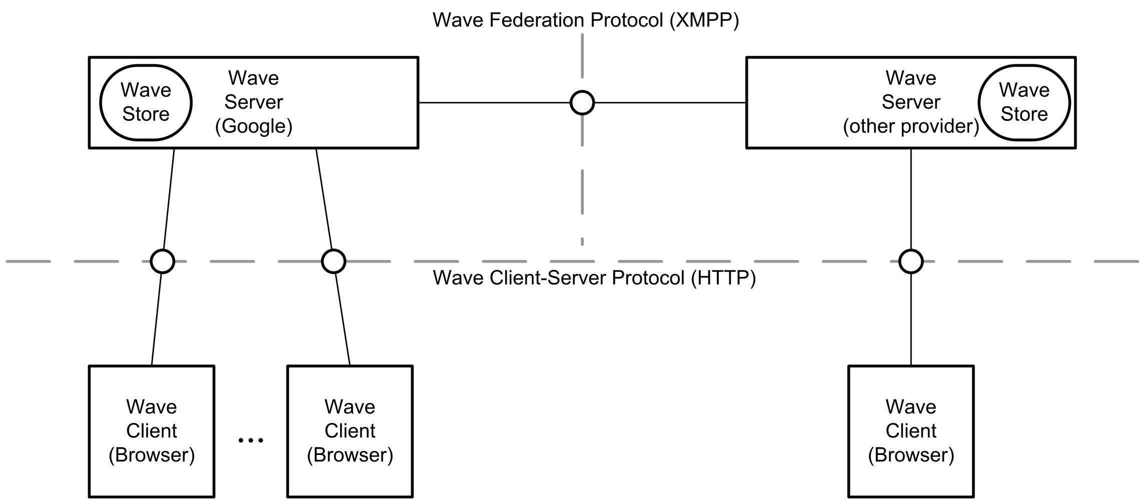


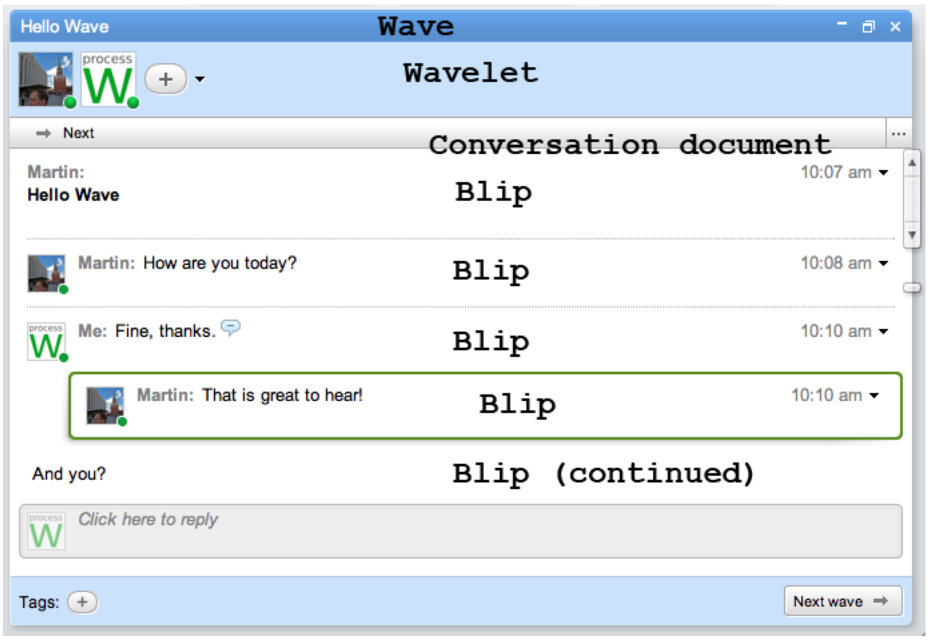
Figure 1: Wave architecture and environment

from different providers and domains to participate in one wave just as different users from a single provider would. Google decided to release parts of Google Wave under an open source license and standardize all communication protocols in an open process in order to support this distributed approach in the Wave architecture. [9]

2.2 Wave Conversation Model

One of Google Wave's most interesting properties is its threaded conversation model which allows users to edit all parts of the document and spawn new messages anywhere in the text. [10] If you want to write extensions for Google Wave, it is crucial to understand this model, so let us have a closer look at it.

When you start a new Wave your cursor is automatically put into an empty message box in which you typically enter the title of the wave and a short introduction for other participants. Anyone who wants to add content to the wave creates another one of these message boxes. In the wave data model, the entities these boxes represent are called blips. Blips can contain rich text, images, attachments or third-party extensions, called gadgets. Blips can also contain other blips thus forming the hierarchical structure of the threaded conversation.


Hello Wave

Martin:
Hello Wave
10:07 am

Martin: How are you today?
10:08 am

Me: Fine, thanks.
10:10 am

Martin: That is great to hear!
10:10 am

And you?

Figure 2: The wave conversation model

All blips are contained by the *conversation document*. Apart from the conversation document there can be other documents, for example one that stores all tags of a wave or an annotation document. Generally, documents store all the content that belongs to a wave. Several documents, in turn, form a *wavelet*. A wavelet defines a list of *participants*, who can access its contained documents, and what they are allowed to do with it. Usually, waves consist only of a single wavelet, which was created together with the wave. And if you add participants to the wave, you automatically grant them access to this initial wavelet. But if you spawn a private reply inside this wave to a subset of the wave's participants, a new wavelet, which allows only the recipients of the reply to see and edit its contents, is created.

2.3 Extensibility

Google provides a set of application programming interfaces (API) that enable developers to write their own applications to extend Google Wave. [11] The two most important interfaces are the Gadgets API and the Robot API. The processWave.org Editor itself is implemented as a Google Wave Gadget. Thus, the next section gives you an introduction to the possibilities of the Gadgets API and where its limitations are. Afterwards, I give a short explanation of the Robot API.

Other Google Wave APIs are

- the Wave Data API, which enables external applications to access and modify a user's waves,
- the Wave Embed API, which allows to augment other web applications with Google Wave
- and the WaveThis service, which can be used to programmatically add content to a new Wave.

More information on them can be found in the Google Wave API documentation [11].

2.3.1 Gadgets

The Wave Gadgets API is an extension to the Google's Gadgets API which already existed before Google Wave. [12] With the Gadgets API you can write small web applications that can run in other applications or websites to extend their functionality (gadgets). A really simple gadget looks like this:

```
<?xml version="1.0" encoding="UTF-8" ?>
<Module>
  <ModulePrefs title="Simple Gadget" />
  <Content type="html">
    <![CDATA[
       // Gadget HTML and JS goes here
       Hello, World!
    ]]>
  </Content>
</Module>
```

Listing 1: Hello World Gadget

As you might have guessed, this gadget displays "Hello, World!" where it is embedded.

Inside the `<![ CDATA[...]]>` element of the `<Content>` section you can write the HTML and JavaScript code for your gadget. From there the application code can call all JavaScript functions that the Gadgets API provides. They are documented at [13].

You then host this XML file on a web server and pass the URL to a web service that supports gadgets, e.g. iGoogle. A Gadget Server will parse the XML document and render the application code from the `CDATA` section into an HTML iframe on the page of the Google application. There it runs securely isolated from the rest of the web service with access only to the functionality that is provided by the gadget API. [14]

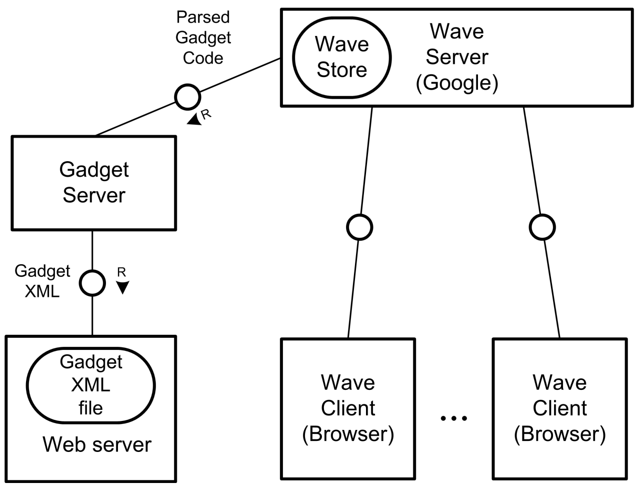


Figure 3: Wave gadget architecture

The Wave Gadgets API extends the normal Gadgets API with distinct functionality that helps you write multi-user applications for Google Wave. The most important of these features is the *gadget state*.

The Wave Gadget State The Wave Gadget state is a key-value store that belongs to a gadget. Each instance of the gadget has its own state object, but the state is shared between all participants of the wave. [1] Using the API, the gadget can modify

[1] More precisely, private state objects for each participant were added to the API recently. However, they were not used in our project. Further information on the private state can be found at [12].

the state and register callbacks to react to state changes. In the following example you see a basic gadget that uses the gadget state:

```
<?xml version="1.0" encoding="UTF-8" ?>
<Module>
<ModulePrefs title="Simple Wave Gadget" height="120">
  <Require feature="wave" />
</ModulePrefs>
<Content type="html">
  <![CDATA[
    <script type="text/javascript">
    function stateUpdated() {
      if(wave.getState().get('message')) {
        document.getElementById('messageDiv').innerHTML =
            wave.getState().get('message');
      }
    }

    function init() {
      if (wave && wave.isInWaveContainer()) {
        wave.setStateCallback(stateUpdated);
      }
    }
    gadgets.util.registerOnLoadHandler(init);

    function setMessage() {
      wave.getState().submitDelta({
          'message': document.getElementById('messageText').value
      });
    }
    </script>

    <div id="messageDiv"></div>
    <input id="messageText" type=text>
    <input type=button value="Submit" onClick="setMessage()">
  ]]>
</Content>
</Module>
```

Listing 2: Simple wave gadget

This gadget has an input text field and a button. Whenever a user clicks the button, the `innerHTML` property of a `div` element is updated with the content from the text box. However, note that the property is not updated directly. Instead, the `setMessage` function writes the text from the input box to the shared key-value store under the key `'message'`. When the state change has been performed, the `stateUpdated` function gets called because it was registered as a state callback upon gadget initialization. The `stateUpdated` function reads the value that belongs to the `'message'` key from the gadget state and writes it to the `div` element. This happens on all participants of the wave because everyone registered the same callback and has access to the same shared state. Thus, all users see the same text whenever one of them clicks the submit button.

The shared gadget state is the basic concept that allows developers to write collaborative Google Wave Gadgets. However, the underlying model has a serious drawback: When two participants submit a value to the gadget state using the same key at the same time, one value will immediately overwrite the other, an effect known as *Lost Update*. Thus, writing applications that allow concurrent actions can be quite a challenging task. For example, if you wanted to modify the gadget in listing 2 so that each time a user clicks the submit button the message is *appended* to the `div` element rather than replacing it's complete content, the Lost Update problem would make it complicated to maintain a consistent list of messages on all clients when two users click the button simultaneously.

More Gadgets API features Besides the gadget state, a gadget can access more properties of a wave. For instance it is possible to retrieve information on all participants who can access the wave: their wave id, their name and the url of their profile image. The Gadgets API also provides methods to find out which participant created the wave or views the current instance of the gadget. Furthermore, a gadget can be notified when the containing blip changes its state from read mode to write mode. This is often used to implement different read and edit modes for the gadget that toggle with the blip. More information on writing Gadgets and a complete reference can be found at [12].

One thing that gadgets cannot do is accessing a wave's content other than the gadget state. For example, a gadget cannot access any of the wave's blips or create a new blip. To accomplish this, there is a different type of extension called *Robots*, which I will explain shortly in the following section.

2.3.2 Robots

Robots are automated participants on a wave. They are written using the Robot API in either Python or Java. Robots are able to access a Wave to which they are added, modify its content, add and remove participants or even create new waves. As shown in figure 4 they communicate with the Google Wave servers over HTTP using the Robot Wire Protocol. The Robot API abstracts from this protocol to simplify the development.

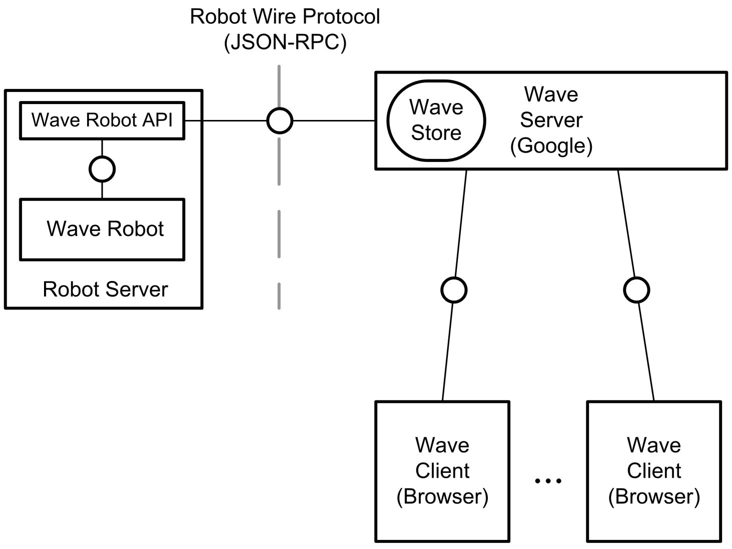


Figure 4: Wave robot architecture

Unlike Gadgets, Robots do not necessarily have a user interface. Instead, the Robot API provides an event system that lets Robots react to certain events, like being added to or removed from a wave, a change of the wave's content or a new participant joining the wave. A Robot, however, can interact with a Gadget that acts as its user interface by reading and modifying the gadget state. An example for this is the group management Robot *Invity* which Markus Götz and Thomas Zimmermann developed in an early phase of the project.[2]

[2]`http://www.processwave.org/2009/11/invity-group-management-in-wave.html`, accessed on 24 June 2010.

A more thorough introduction to robots including a complete reference of the API is available at [15].

3 Oryx in the processWave.org Editor

The processWave.org Editor builds on Oryx, an open source modeling platform started as a bachelor's project in 2006.[3] To create the processWave.org Editor, we embedded the source code of the Oryx Editor into a Google Wave Gadget. Then we modified it heavily so it supported edits from multiple users in a distributed way, that is, without the help of a server that manages collaboration by resolving conflicting edits or similar.

This section starts with an introduction to the architecture of the Oryx Editor. Afterwards, I explain how our modifications affected this architecture and the code base. The bigger picture, how Oryx integrates with the rest of the components of the processWave.org Editor, follows in section 4.

3.1 Oryx Architecture

Oryx is a web application written in HTML, JavaScript and SVG. It relies on two JavaScript frameworks, Ext JS[4] for the user interface implementation, and Prototype[5]. Oryx has two different back-end components written in Java and J2EE: The *Editor back-end* serves all content that is loaded dynamically, like the editor stencil sets and profiles, and provides a set of stateless plugins that implement server-side functionality, such as model export and transformation. The *Repository back-end* implements model persistance and management.

Figure 5 shows an overview of Oryx and its environment. [16] As you can see in the diagram, the Oryx Editor front-end consists of three important parts, the Core, the Editor and a large set of plugins.

The Oryx Core is a collection of the most integral components and functionality. These include shapes, the modeling canvas and stencil sets, among others. The Core also contains the command base classes that are used for the command pattern implementation, on which the undo and redo features are built. [17]

[3]`http://code.google.com/p/oryx-editor/`, accessed on 24 June 2010.
[4]`http://www.extjs.com`, accessed on 24 June 2010.
[5]`http://www.prototypejs.org`, accessed on 24 June 2010.

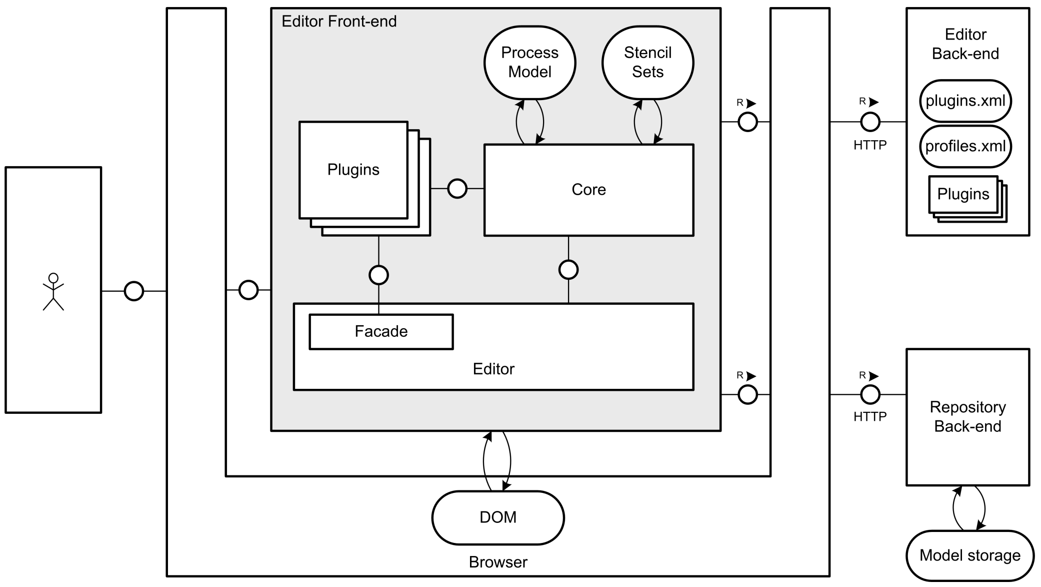


Figure 5: Oryx architecture

Plugins implement all additional functionality that is not included in the Core, such as the Shape Repository, Drag-and-Drop, Copy-and-Paste, or Undo and Redo. Since much functionality has been added to Oryx over the course of various research projects, the set of plugins has grown very large, with over 100 plugins today.[6] All available plugins have to be registered in the `plugins.xml` file. Different sets of plugins can be loaded for different editor configurations. All of these configurations, also called profiles, are maintained in the `profiles.xml` file

The Editor component is used to load and initialize the application. It also contains the Plugin Facade, an implementation of the Facade Pattern [18] that manages all communication with and between plugins. Since Oryx' plugins are losely coupled and do not hold references to each other, one plugin cannot call a function of another. Instead both hold a reference to the Plugin Facade and use the facade's event system: A plugin can register a callback function for an event using the `facade.registerOnEvent` function. Every time another plugin calls the `facade.raiseEvent` function to raise that particular event, the facade will invoke the callback.

[6] `http://code.google.com/p/oryx-editor/source/browse/trunk/editor/client/scripts/Plugins/`, accessed on 24 June 2010.

3.2 Oryx as a Collaborative Application

When Oryx was designed, compatibility with a multi-user environment and concurrent edits from different users on one model were not among the requirements. Thus, we had to modify Oryx' source code heavily in order to turn it into a distributed, collaborative application: First of all, we needed an architecture that supported concurrent edits. Additionally, many existing components did not behave as required once collaboration was enabled.

The exact implementation details of collaboration support and other changes to the Oryx Editor are described in other theses of the project: Michael Goderbauer explains collaboration support and multi-user editing in [19]. Undo and Redo features in the collaborative environment are outline by Thomas Zimmermann in [20]. Christian Reß' thesis describes changes to the User Interface like the reworked Shape Repository [21].

To give you an overview of the necessary modifications, the next section highlights a selection of them.

3.2.1 Command Serialization

In order to let different users edit the same model at the same time, we use the Wave gadget state as a distributed command stack that contains JSON serialized Oryx commands. This way, all participants are notified via a `stateUpdatedCallback` whenever a new command has been executed and put onto the stack by a remote user. The clients can then apply the same command in order to see the same, consistently updated model on all clients.

Implementing the distributed command stack on top of the gadget state, which is only a key-value store, was non-trivial: A JavaScript implementation of the Lamport clock algorithm was necessary in order to synchronize concurrent edits among all clients in a distributed way and resolve conflicts seamlessly. The product of this effort is an open source JavaScript library for command based Google Wave Gadgets called "syncro".[7]

[7] `http://www.bitbucket.org/processwave/syncro/overview`, accessed on 24 June 2010.

Furthermore, we needed to refactor the Oryx command architecture completely. Afterwards, all implementations of commands inside the plugins had to be adapted to the new architecture. Only plugins for which we carried out these modifications are included in our version of the application.

The exact details on these two topics can be found in Michael Goderbauer's aforementioned thesis [19].

3.2.2 Changes due to multi-user editing

Once collaborative editing had been enabled, parts of the existing code that worked perfectly normal in a single user environment suddenly caused strange, unwanted effects. These ranged from minor user experience glitches to serious faults, which resulted in model inconsistencies among different clients. Therefore, we adjusted implementations of various parts of the editor. I will describe a few examples in this section.

Concurrent Moves Drag-and-Drop functionality, which lets you move a shape, is implemented in the `DragDropResize` plugin. More specifically, the plugin contained a command class called `ORYX.Core.Command.Move`, which was moved to `ORYX.Core.Commands["DragDropResize.MoveCommand"]` during the command refactoring. Let's have a look at the original command's `execute` function:

```
execute: function(){
    this.dockAllShapes();
    // Move the shapes
    this.move(this.offset);
    this.addShapeToParent(this.newParents);
    // Reset the selection
    this.selectCurrentShapes();
    this.plugin.facade.getCanvas().update();
    this.plugin.facade.updateSelection();
}
```

Listing 3: Implementation of `execute` in `ORYX.Core.Command.Move`

In the `move` method implementation, which is being called in the above listing, an offset from the command instance gets added to the shape's position. This is a valid implementation strategy for the standalone version of the Oryx Editor. In a collaborative version of the editor, however, the implementation caused very surprising behavior: As soon as two users moved the same shape simultaneously, the two resulting commands were applied after each other so that both offsets added up. That way, the shapes appeared in a position where none of the participants really expected them. Or if both participants moved the shape in the same direction, adding the offsets even caused the shapes to be moved out of the viewport eventually.

As a solution we decided to refactor the move method, so it accepted a target position for each shape to be moved as a parameter. This implementation strategy is clearly less easy to understand for the developer and thus not preferrable in a single user environment. The usability concerns, however, prevailed on us to do the refactoring.

Remote Selection Another problem in the above implementation is the `selectCurrentShapes` function call. In the implementation of `selectCurrentShapes` the user's selection changes so it contains exactly those shapes that have been moved. Presumably, this code is responsible to highlight all moving shapes in cases where they are not already selected, for example when a user clicks the redo button to move shapes. In a multi-user environment however, the same code makes a user lose their selection as soon as a remote user moves a shape. This makes it hard to move any shapes at all while another user is editing the diagram. For this reason, we had to make sure actions by one user could only change the model and not affect another user's selection.

These problems are just examples for a whole series of issues that existed because the original Oryx code was not designed to work in a distributed multi-user environment. In fact, problems like this made up a substantial amount of the adjustments that we made to the Oryx source code. This class of issues also demonstrates that a possible future integration of the Oryx Editor and processWave.org Editor into a single, unified code base would presumably be a difficult task and make further development on both products significantly more complex.

3.2.3 Modified Plugins

To give you a better overview of the changes that we made to the Oryx source code, table 1 lists all plugins that are currently included in our version of Oryx. It also visualizes what sort of modifications were necessary:

Command serialization: This means, that the plugin's functionality uses the command architecture, and therefore it needed to be adjusted to the new command architecture.

Collaboration adjustments: These plugins required changes, similar to those highlighted in the previous section, where code had to be adapted in order to work in a distributed environment.

Other changes: This category includes Oryx bug fixes and minor modifications caused by other than the already listed reasons.

New plugin: All plugins that are not included in the standard Oryx release. However, these might heavily borrow code from existing plugins, like the reworked shape repository does, for example.

Note that the majority of the around 100 plugins from the Oryx source code repository[8] are not included in our release. Most of them contain functionality which was not of much interest to our project and was therefore not adapted to and tested with our version of Oryx.

3.2.4 Oryx Server Components

It should also be noted that the two Oryx server applications, the Editor back-end and the Repository back-end are not included in our application.

Most of the features that the Oryx Editor back-end provides were not really interesting for the processWave.org Editor and integration with the multi-user environment would have been non-trivial in some cases.

[8]`http://code.google.com/p/oryx-editor`, accessed on 24 June 2010.

Plugin name	Command serialization	Collaboration adjustments	Other changes	New Plugin
AddDocker	x	x		
ArrangementLight				x
BPMN2_0				
CanvasResize	x	x		
Changelog				x
DockerCreation	x		x	
DragDocker	x	x		
DragDropResize	x	x		
Edit	x	x		
Farbrausch				x
FarbrauschAfterglow				x
FarbrauschLegend				x
FarbrauschShadow				x
KeysMove	x		x	
NewShapeRepository	x			x
Overlay			x	
Paint				x
PropertyTab				x
RenameShapes	x		x	
Schlaumeier				x
SelectionFrame		x		
ShapeHighlighting		x		
ShapeTooltip				x
ShapeMenu	x	x	x	
SideTabs				x
Syncro				x
SyncroOryx				x
Toolbar			x	
UML				
View			x	
Wave				x
WaveGlobalUndo				x

Table 1: Modifications to Oryx Plugins

Integration with the Repository is an interesting idea because it would provide an additional persistence layer outside Google Wave. In the mean time, there is an Import from and Export to the Oryx Editor, which uses a specially designed interface of the Oryx Editor back-end that is deployed at `http://www.oryx-project.org`. Thomas Zimmermann further explains this integration in his thesis [20].

4 processWave.org Editor Architecture

While Oryx is one integral part of the processWave.org Editor, there are more components that allow the processWave.org Editor to seamlessly integrate with Google Wave and make use of its features. This section provides a thorough introduction to the software architecture of the processWave.org Editor and explanation of these components.

4.1 General Architecture

The processWave.org Editor is entirely written in HTML and JavaScript using the Google Wave Gadgets API and jQuery UI[9]. As you can see in the architecture overview that is depicted in figure 6, the application consists of two parts: The specially modified version of Oryx, which was introduced in the last section, and a Google Wave Gadget.

Oryx is embedded into this gadget using an HTML inline frame element (iframe), but logically, Oryx and the gadget are two separate units. Both use the Cross-Document Messaging mechanism of HTML5 [22] to communicate with each other. Two components are responsible to handle this communication: The Wave Communication Adapter, which runs as a plugin in Oryx, and the Oryx Communication Adapter, which belongs to the gadget. All other components use the interfaces of these communication adapters when they need to pass data from Oryx to the gadget or back.

Currently, two components use this communication channel in the gadget: *SyncroWave* accesses the gadget state for the Syncro plugin, which is responsible for model synchronization on the Oryx side. *Farbrausch*, on the other hand, passes information about the wave's participants to the Farbrausch Oryx plugin, which assigns them colors in order to highlight changes to the model. It also writes back this user-color mapping to the gadget state. More information on the implementation of Syncro and Farbrausch can be found in the theses of Michael Goderbauer [19] and Martin Krüger [23].

[9]`http://www.jqueryui.com/`, accessed on 24 June 2010.

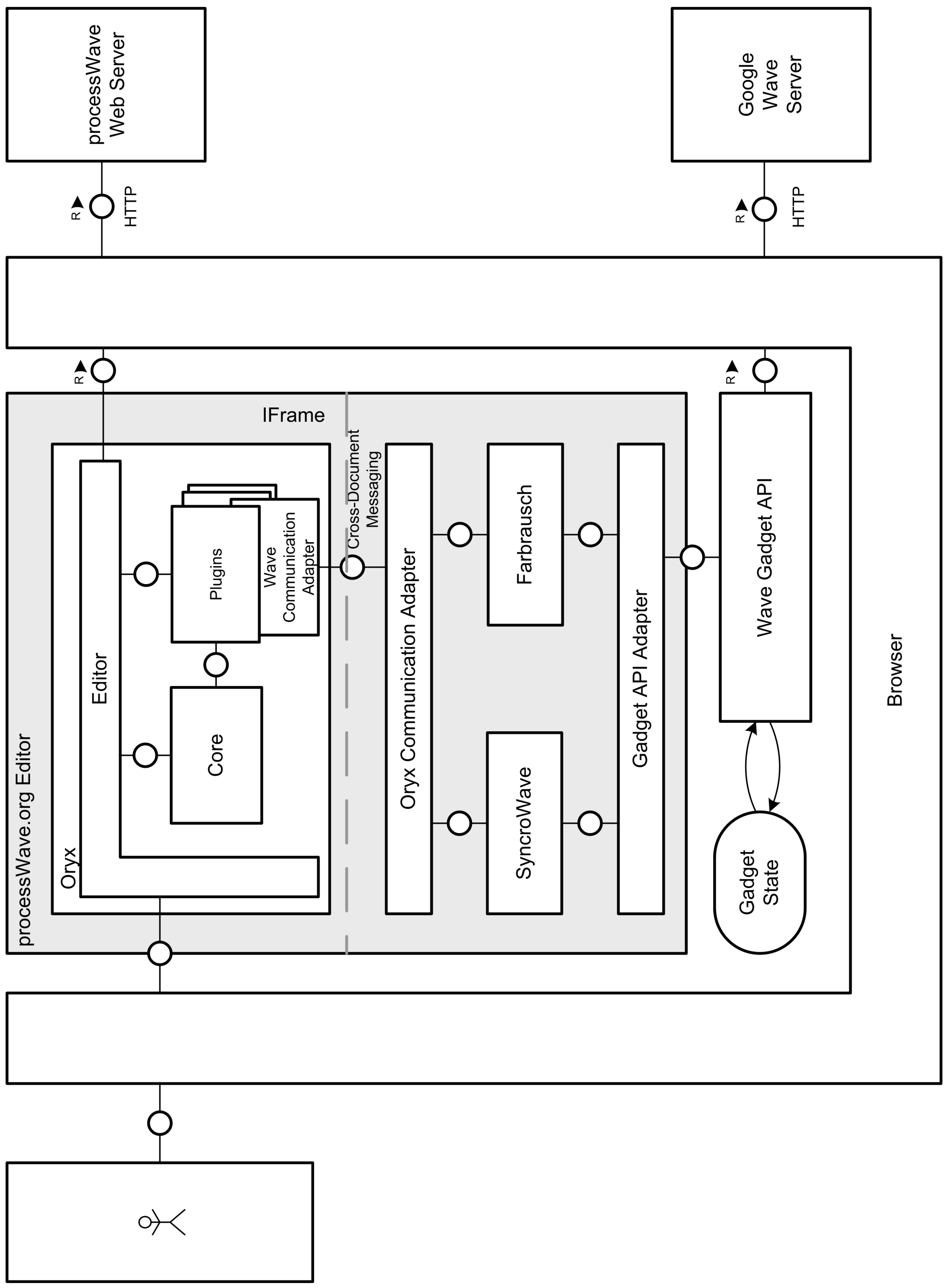


Figure 6: Architecture of the processWave.org Editor

The Gadgets API adapter is a proxy-like component that manages all communication with the Google Wave Gadgets API and adjusts behaviour of the API to our needs.

The next sections describe the gadget's components in more detail and point out the design decisions behind them.

4.2 Cross-Document Messaging

To understand why we separated the actual editor from the rest of the gadget and put it into it's own iframe, we have to recall how gadgets work: As explained earlier in section 2.3.1, gadgets are defined in a single XML file that contains both metadata and the HTML and JavaScript code of the gadget. This file is interpreted by a Gadget Server, which then renders the gadget into an iframe in the Wave client.

This means, that the window location[10] of a gadget's content is not the URL of the web server that hosts the gadget XML file, but the URL of Google's Gadget Server. This was a problem when we first tried to integrate the Oryx Editor into a gadget.

As explained in the Oryx Architecture overview of section 3.1, Oryx loads certain data dynamically during its initialization, for example the stencil set. It does so by issuing XMLHttpRequests[11] to the processWave web server.

If you simply put the Oryx source code into a Google Wave gadget, these XML-HttpRequests would fail: The *Same Origin Policy*[12] in modern browsers' security models prohibits XMLHttpRequests to a different host than the one of the requesting document. And since the gadget's document location is not the processWave web server, the browser would block all of these requests. The Oryx initialization would fail.

This was a major road block when we tried to embed Oryx into a wave gadget, so we investigated three different strategies to solve this issue.

gagets.io.makeRequest Since issuing XMLHttpRequests to other domains is quite a common scenario for a gadget, Google's Gadgets API provides a method

[10]`https://developer.mozilla.org/en/DOM/window.location`, accessed on 24 June 2010.
[11]`http://www.w3.org/TR/XMLHttpRequest/`, accessed on 24 June 2010.
[12]`http://www.w3.org/Security/wiki/Same_Origin_Policy`, accessed on 24 June 2010.

called `gadgets.io.makeRequest`[13], which proxies XMLHttpRequests through Google's servers in order to circumvent the Same Origin Poilicy. However, this interface only allows to make asynchronous requests and Oryx performs its requests synchronously. Rewriting the Oryx code to use asynchronous requests was estimated by the team to be very time consuming. Therefore, `gadgets.io.makeRequest` was not really a solution to the problem.

Cross-Origin Resource Sharing A second idea how to solve the problem was using a preliminary HTML5 standard called *Cross-Origin Resource Sharing.* [24] The standard describes a protocol that gives web servers the possibility to opt-in to cross-domain requests. According to the protocol, the implementing browser adds the source origin[14] to the `Origin` header of the XMLHttpRequest. If the server that receives the requests understands the protocol and wants to allow the cross-origin request, it sets the `Access-Control-Allow-Origin` header of the response to either "*" or the source origin that was specified in the request's `Origin` header. Upon receiving the response, the browser checks whether the `Access-Control-Allow-Origin` header is "*" or a case-sensitive match of the source origin string. If this is the case, the cross-origin request succeeds. Otherwise the browser will let the request fail.

The changes to the webserver configuration that would have been required to set the `Access-Control-Allow-Origin` header, would have made deployment of our application more complicated. Thus, we finally decided to embed Oryx into the the gadget using an iframe.[15] This way, the document location of Oryx belongs to our web server and all XMLHttpRequests work as usual. However, the iframe behaves like a sandbox to the JavaScript code inside it: JavaScript namespaces that are available in the gadget, like the Wave Gadgets API, for example, cannot be accessed from the document inside the iframe. For this reason, we use the Cross-Document Messaging API of HTML5 for communication between Oryx and the gadget.

[13]`http://code.google.com/apis/gadgets/docs/reference/#gadgets.io`, accessed on 24 June 2010.

[14]`http://www.w3.org/TR/cors/#source-origin`, accessed on 24 June 2010.

[15]Later we learned that the Gadgets API is incompatible with the Prototype JavaScript framework (cf. `http://code.google.com/p/google-wave-resources/issues/detail?id=143`, accessed on 24 June 2010), which Oryx uses. So directly putting the Oryx code into a gadget would not have been possible without removing Prototype anyway.

Cross-Document Messaging The Cross-Document Messaging API allows to exchange String objects between documents of two related windows, like an iframe and its parent window, in a secure way, without the possibility of cross-site scripting attacks. [22] The communication is established in two steps: First, you have to register an event handling function that receives all incoming messages. To do that, you call `document.addEventListener("message", handler)` from the receiving document. The first parameter denotes that you register a listener for the `"message"` event type. The second parameter is a reference to the event handling function. This function will receive one parameter, which has a parameter `data` holding the message. To send a message, you call `postMessage` on the window object that is to receive the message inside the sending document. For example, `window.parent.postMessage("Hello, World!", "*")` sends *Hello, World!* to the parent window of an iframe. For security reasons, the message will only be sent if the target window's origin matches the second parameter, "*" being a wildcard.

In the next section, I will describe, how exactly Cross-Document Messaging is implemented in the processWave.org Editor.

4.3 Communication Adapters and Protocol

As previously mentioned, the Oryx and Wave Communication Adapters are responsible to perform communication between Oryx and the gadget. That means, they are the only components that use the Cross-Document Messaging API directly. Both use a basic protocol in their communication, in order to simplify message dispatching to other components. Messages in their protocol are JSON serialized objects that contain three string properties:

- `target` denotes the component that is meant to handle the message.
- `action` specifies what the receiver should do with the message, e.g. "write this to the gadget state".
- `message` contains the message itself, for example the data that should be written to the gadget state.

The Oryx and Wave Communication Adapters provide interfaces that allow to send such messages and register a handling function for a certain target. In the next sections, I will describe these interfaces.

Oryx Communication Adapter The Oryx Communication adapter belongs to the gadget and manages communication with the Oryx component. It is defined as the JavaScript object `oryx` in the `gadget/oryx.js` file. To register a handling function for incoming messages, you call `oryx.addMessageDispatcher(target, dispatcherFunction)`, where `dispatcherFunction` is the function that is called when a message arrives and `target` is a string that specifies, for which message target the function should be invoked. With `oryx.sendMessage(target, action, message)` you can send a message to the Oryx component. The three parameters correspond to the three properties of the previously defined message protocol.

Wave Communication Adapter The counterpart on the Oryx side, the Wave Communication Adapter, is implemented as the plugin `ORYX.Plugins.Wave` in `oryx/editor/client/scripts/Plugins/wave.js`. Its interface works differently because it uses the event system of Oryx, which was shortly introduced in section 3.1: Every time, the adapter receives a new message, it raises an `ORYX.CONFIG.EVENT_NEW_POST_MESSAGE_RECEIVED` event. Other plugins can listen to this event type and will receive the parsed message in the `data` property of the event object. Unlike on the gadget side, all event handling functions will receive all messages. They have to inspect the message target themselves, in order to find out if they are the designated receiver. Messages with the target `"oryx"` are an exception: These are messages related to Core functionality and handled inside the Wave Communication Adapter.

4.4 Gadgets API Adapter

Of course there are many different instances where the gadget makes use of the Google Wave Gagets API. For example, the shared gadget state is not only used to synchronize the model across all clients, but also to store the stencil set name or the user-color mapping of Farbrausch. The Gadgets API Adapter proxies all calls to the

Wave Gadgets API, in order to manage the use of the gadget state and coordinate access to callbacks. It is defined in the file `gadget/adapter.js`.

Other components can retrieve an instance of the adapter by calling `adapter.connect(prefix, swappable)`. The `prefix` parameter is a string that is used as a prefix to the keys that the connecting component writes to the gadget state. For example, all user-color mappings of Farbrausch are prefixed with “fr”, so they can easily be distinguished from Oryx Commands, which are prefixed with “pw”. The second parameter is a boolean value that indicates if key-value pairs with the given prefix should be written to an external data store by “swappy”. Swappy was written to circumvent a 100k size limitation of the gadget state that has now been lifted.[16] Its implementation is discussed in Markus Götz’ thesis [25].

The adapter instance returned by `adapter.connect` offers a similar interface like the `wave` namespace of the Wave Gadgets API. For example, if `_adapter` is the adapter instance, `_adapter.getParticipants()` would return a list of the wave’s participants. Some of the functions, however, show slightly modified behaviour, so it is a bit easier to use the API in the gadget:

- `_adapter.submitValue` and `_adapter.submitDelta` automatically add the prefix that was used for the proxy initialization to all keys, before writing them to the gadget state.
- `_adapter.getState` filters for this prefix. It returns a state object that only contains key-value pairs which were written to the state by the connecting component.
- The Wave Gadgets API only allows to register a single function for each of the participants, mode and stateUpdated callbacks. [12] If you set the stateUpdated callback repeatedly, for example, by calling `wave.setStateCallback` twice with two different callback functions, only the last function will be invoked when the state changes. The proxy’s `proxy.setStateCallback`, `proxy.setParticipantCallback` and `proxy.setModeCallback` functions, however, allow you to set as many callback functions as you like.

[16]`http://wave-api-faq.appspot.com/#sizelimits`, accessed on 24 June 2010.

5 Infrastructure and Deployment

This section provides an overview of the tools and servers that we used to develop and deploy the processWave.org Editor. I will start with an introduction to the source code management basics required for development and testing. Afterwards, you will learn how to build and deploy a version of the editor that is ready for production on the processWave server. And finally, production deployment on Amazon CloudFront is explained.

5.1 Source Code and Infrastructure

We used the distributed revision control tool *Mercurial* (hg)[17] to manage all our code. New Mercurial users can find a good introduction at `http://www.hginit.com`. All new features were developed in separate branches and merged into the `default` branch upon approval.

Because you have to make a gadget available on a web server in order to test it, all source code was pushed to our development server with the public IP address `141.89.225.29`. On this machine an nginx webserver [18] makes the source code from the Mercurial repository in the `/home/hg/repos/oryx-frontend` directory available at `http://code.processwave.org`, using the repository browser that is built in to hg.

To test any version of the gadget in the repository's version tree you can add the development version of the gadget from `http://code.processwave.org/raw-file/default/gadget/oryx.xml` to a wave. You can also use its gadget installer, which can be found at `http://code.processwave.org/raw-file/default/gadget/installer.xml`. This development gadget lets you load the editor from a given hg branch name, thus making testing a feature branch more convenient.

The processWave Google code project at `http://code.google.com/p/processwave/` contains a clone of the source code, which we have released to the public. There is no automatic synchronization between both repositories.

[17]`http://mercurial.selenic.com/,accessedon24June2010.`

[18]`http://www.nginx.org/,accessedon24June2010.`

5.2 Build process

Since the JavaScript source code of the processWave.org Editor alone is over 500 kilobytes in size and distributed across many files, it is absolutely necessary to compile the source code before deploying it for the public. Otherwise, the gadget's loading time easily exceeds ten seconds in our tests.

To automate the various compilation steps and simplify deployment, we have created an automated build process using the Apache Ant tool[19]. The `build.xml` configuration file for this process can be found in the application's root directory.

To start a new build, create a new clone of the editor's repository somewhere on the development server or simply use the existing one at `/home/hg/editor-build`. Afterwards, run the `compile` target by executing `ant compile` in the editor's root directory.

The compile target of the ant file executes several compilation steps: First of all we use Oryx' stencil set compiler to compile all SVG and JSON files of each stencil set into single JSON files. Afterwards, we create the Oryx profiles, JavaScript files that contain specified sets of plugins for certain stencil sets as specified in the `profiles.xml` file, by using Oryx' profile creator. Finally, the Google Closure compiler[20] is used to compile JavaScript files from the Oryx Core and the editor's gadget component into single files.

5.3 Deployment

When everything has been built, the `deploy` target can be run to deploy the new version of the editor. This target copies all required files to the directory `/var/www/oryx.processwave.org/htdocs`. The nginx web server serves these files at `http://oryx.processwave.org`. The URL for the gadget is `http://oryx.processwave.org/gadget/oryx_stable.xml`, its gadget installer can be found at `http://oryx.processwave.org/gadget/installer_stable.xml`.

[19]`http://ant.apache.org/`, accessed on 24 June 2010.
[20]`http://code.google.com/closure/compiler/`, accessed on 24 June 2010.

5.4 Deploying to Amazon CloudFront

When the processWave.org Editor was launched for the public during the Google I/O 2010 conference in San Francisco, we decided to use Amazon CloudFront as a deployment system for the application because it offers lower latency for connections from nearly everywhere in the world at comparably low cost.

Amazon CloudFront is a so-called *Content Distribution Network*, which automatically replicates content such as static files, from the Amazon Simple Storage Service (Amazon S3), across a network of servers that are distributed strategically around different places in the world. If you request a resource that is hosted by the network, your request will automatically be directed to the server that is closest to you, thus delivering the file with the shortest possible delay.

Deploying to Amazon Cloudfront slightly varies from the steps described in the last sections: Before you start a build process, the version you would like to deploy has to be merged into the "cloudfront" branch of the hg repository. It contains a modifed build file and different URLs in the source code and therefore must not be merged back to the default branch. You then build and deploy all files from this branch as described in the last section. However, the deploy step will copy all required files to the `../cloudfront` directory (relative to the `build.xml` file) instead of putting them in the web server's directory.

Once the files are copied you can deploy them to the Amazon S3. The simplest way to do this, is to use one of the many free or commercial uploading tools. When uploading the files, you have to make sure each file is uploaded with the correct Content-Type Header[21], since S3 will deliver the file with exactly this Content-Type. The only tool we could find that does this automatically, is the CloudBerry S3 Explorer PRO[22]. After all files have been uploaded, you have to make sure, to set ACLs on all files, so that everybody is allowed to read all files in the S3 bucket. CloudFront is configured to automatically refresh its cache from the S3 bucket every 24 hours. After that, the deployment is available at `http://ddj0ahgq8zch6.cloudfront.net/`.

[21]`http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html`, accessed on 24 June 2010.
[22]`http://cloudberrylab.com/`, accessed on 24 June 2010.

6 Conclusion and Outlook

The bachelor's project team has released the processWave.org Editor to the public on May 18, 2010. [23] The source code was made freely available under the terms of the MIT license from its Google Code project at `http://code.google.com/p/processwave/`. One day after the launch, the editor was presented by Michael Goderbauer at Google I/O 2010 in San Francisco in a talk called "Google Wave API design principles Anatomy of a Great Extension" by Pamela Fox from the Google Wave team.[24]

The processWave.org Editor is the only freely available graphical modeling tool for Google Wave at the moment. The only possible alternative that we know of is called „Gravity", a collaborative process model editor by SAP for their proprietary collaboration platform StreamWork.[25] However, Gravity is still a prototype and therefore currently not available to the public.

Of course it was not possible to implement all features that we would have liked to see in the final product. For example a seamless integration with existing Oryx repositories is still missing. But although there is room for improvements and some known issues exist, the editor seems to be in active use: Since its launch until the publication of this document the application has been used over 3,000 times by over 1,500 users from 74 countries. [26] Therefore, continuing its development and investigating a possible synchronization with the Oryx code base would be reasonable.

[23]`http://www.processwave.org/2010/05/processwaveorg-editor.html`, accessed on 24 June 2010.

[24]`http://code.google.com/events/io/2010/sessions/wave-api-design-extensions.html`, accessed on 24 June 2010.

[25]`http://www.sapweb20.com/blog/2009/10/sap\T1\textquoterights-gravity-prototype-business-collaboration-using` accessed on 24 June 2010.

[26]According to the application's usage statistics, collected by Google Analytics.

Bibliography

[1] A. Grosskopf, G. Decker, and M. Weske, *The Process: Business Process Modeling Using BPMN*. Meghan Kiffer Pr, 2009.

[2] J. Rumbaugh, I. Jacobson, and G. Booch, *The unified modeling language reference manual, Volume 1*. Addison-Wesley, 2005.

[3] "Google Wave Overview." Youtube video. Available online at `http://www.youtube.com/watch?v=p6pgxLaDdQw`, accessed on 24 June 2010.

[4] "Google Wave Available for Everyone." Blog post. Available online at `http://googlewave.blogspot.com/2010/05/google-wave-available-for-everyone.html`, accessed on 24 June 2010.

[5] "Google Wave website." Website. Available online at `http://wave.google.com/about.html`, accessed on 24 June 2010.

[6] S. Lassen and S. Thorogood, "Google Wave Federation Architecture." Whitepaper. Available online at `http://wave-protocol.googlecode.com/hg/whitepapers/google-wave-architecture/google-wave-architecture.html`, accessed on 24 June 2010.

[7] D. Wang and A. Mah, "Google Wave Operational Transformation." Whitepaper. Available online at `http://wave-protocol.googlecode.com/hg/whitepapers/operational-transform/operational-transform.html`, accessed on 24 June 2010.

[8] A. Baxter, J. Bekmann, D. Berlin, J. Gregorio, S. Lassen, and S. Thorogood, "Google Wave Federation Protocol Over XMPP." Draft protocol specification. Available online at `http://wave-protocol.googlecode.com/hg/spec/federation/wavespec.html`, accessed on 24 June 2010.

[9] "Google Wave Federation Protocol: Community Principles." Website. Available online at `http://www.waveprotocol.org/wave-community-principles`, accessed on 24 June 2010.

[10] J. Gregorio and N. A., "Google Wave Conversation Model." Draft protocol specification. Available online at `http://www.waveprotocol.org/`

draft-protocol-specs/wave-conversation-model, accessed on 24 June 2010.

[11] “Google Wave API – Google Code.” Website. Available online at http://code.google.com/apis/wave/, accessed on 24 June 2010.

[12] “Wave Gadgets Tutorial – Google Wave API – Google Code.” Website. Available online at http://code.google.com/apis/wave/extensions/gadgets/guide.html, accessed on 24 June 2010.

[13] “Google Gadgets API – Google Code.” Website. Available online at http://code.google.com/apis/gadgets/, accessed on 24 June 2010.

[14] “Google Gadgets Specification – Google Gadgets API – Google Code.” Website. Available online at http://code.google.com/apis/gadgets/docs/spec.html, accessed on 24 June 2010.

[15] “Google Wave Robots API: Overview – Google Gadgets API – Google Code.” Website. Available online at http://code.google.com/apis/wave/extensions/robots/index.html, accessed on 24 June 2010.

[16] W. Tscheschner, “Oryx dokumentation,” tech. rep., Hasso-Plattner-Institut Potsdam, 2007.

[17] W. Tscheschner and N. Peters, “Oryx - Introduction.” Presentation slides, 2009. Available online at http://bpt.hpi.uni-potsdam.de/pub/Public/StudentsCorner/oryx-workshop.architecture-stencilsets-plugins.pdf, accessed on 24 June 2010.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[19] M. Goderbauer, “Synchronization of diagrams in the processWave.org Editor using Syncro,” tech. rep., Hasso-Plattner-Institut Potsdam, 2010.

[20] T. Zimmermann, “Collaborative Undo/Redo and Import/Export of Models in the processWave.org Editor,” tech. rep., Hasso-Plattner-Institut Potsdam, 2010.

[21] C. Reß, “Usability and User Interface Design of the processWave.org Editor,” tech. rep., Hasso-Plattner-Institut Potsdam, 2010.

[22] "9 Communication – HTML5." Draft standard, 2010. Available online at http://www.whatwg.org/specs/web-apps/current-work/multipage/comms.html#crossDocumentMessages, accessed on 24 June 2010.

[23] M. Krüger, "processWave.org: Synchronous Communication in Collaborative Modelling," tech. rep., Hasso-Plattner-Institut Potsdam, 2010.

[24] "Cross-Origin Resource Sharing." W3C Working Draft, 2010. Available online at http://www.w3.org/TR/cors/, accessed on 24 June 2010.

[25] M. Götz, "processWave.org: Memory Management Strategies in Google Wave," tech. rep., Hasso-Plattner-Institut Potsdam, 2010.