University of Innsbruck

Master Thesis

Social Weaver

A Platform for Weaving Web 2.0 Features into Web-based Applications

Author:
Viktor Pekar

Supervisor:
Dr. Michael Felderer

Co-Supervisor:
Dr. Dirk Draheim

August 15, 2013

Contents

List of Figures

List of Tables

Listings

Abstract

Communication through the internet has been made easy in the last few years. But discussing workflows and functionality of web applications or web pages is still a time consuming task, that requires a lot of explanation. Social Weaving introduces a new concept of communication. Injecting - or how we call it: weaving - social elements like chats, wiki pages and so on directly into the view of your application (without the need of modifying the underlying code). Information becomes directly attached to its relevant position.

This thesis will explain the theory behind Social Weaving and show the prototype Social Weaver.

1 Introduction

This thesis is about Social Weaving. A new technique that combines modern communication methods with existing web sites and web applications. The goal is to create a layer above the existing environment without directly modifying it. When we talk about "modern communication methods" we have social media in mind, like wiki pages, chats, comment boxes, etc. but also support for file upload, appointment invitations to shared calendars and so on. After all it doesn't matter what exactly is woven into the environment. Since it might be some HTML code, the user has the free choice. What is such an environment that we mentioned above? Informally we define an environment something that is visible through a browser. Now we have large variety of software we see in a browser. We have static and dynamic web pages, web applications using Flash or Java, and a lot of another technologies. The best case for Social Weaving would be to support seamlessly everything - but as we will see this is not possible.

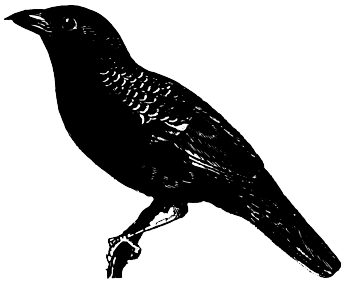

Figure 1: Social Weaver (Philetairus socius) is a species of bird in the Passeridae family endemic to Southern Africa

The great number of standards for the web doesn't prevent that every web page is constructed in a different way. There are no unique identifiers for elements, which would be necessary to guarantee a full Social Weaving support. Even though we cannot change the structures used in the web, we want to show what is possible with Social Weaving even now.

In Section Contribution - 2 we discuss in detail how the basic idea of Social Weaving works and what problems it solves. The rest of this thesis is about Social Weaver - a prototype for Firefox that shows a basic functionality for certain environments. So the second part (3.5) is an abstract requirement analysis that describes what a Social Weaving system needs. The third part (Section 4) shows the architecture of the prototype on a more concrete level.

2 Contribution

In the last couple of years the internet developed into a mass medium. It started with with the simple asynchronous one-to-one communication such as E-mail. Today we have all kinds of communication types: forums and bulletin boards support many-to-many information exchange, one-to-many is being provided by services like Twitter. Chats or instant messages give us the possibility of synchronous transmissions. With the launch of webcams the internet took regular voice calls to another level adding the opportunity to actually see each other. Success stories like Facebook and Twitter show us, that human communication is still in development. The problem is that we see communication as something we were practicing since we exist. Two persons standing in front of each other and using spoken language. But the internet offers us new possibilities therefore we need to take another perspective on communication.

The literal language, as we know it, is powerful. In fact so powerful, that teaching machines to speak and understand is still one of the greater challenges. It brings great advantages for communication. In case we cannot remember a specific word, it is easy for us to come up with an alternative or to somehow outline it. Even persons that are not speaking the same language, will be able to somehow communicate with each other using gestures or images. But on the other hand literal language has its shortcomings. To describe technical or scientific topics precisely we need a lot of words to bring it into understandable context. Everyone who sat in lecture that was a bit over his skills exactly knows this problem to well. The more concrete and complex something becomes, the more we feel the shortcomings of literal expressions.

Software makes no exception. Applications are built while keeping in mind, that a user will actually see the interface and interact with it. We are using a button because a user sees it and pushes it. Where the button is located or in which context it has which functionality is obvious to the user. At least it should be. But what if he wants to discuss something about this button with his colleagues? This could be a question or criticism. Nevertheless he will need to describe where the button

is; in which work flow it appears and so on. Assuming the colleague is not available on location, the usual way would be to create a screenshot, write an explanation, compose an E-mail and send it. From there on the E-mail thread would become the central discussion point related to our button. This is not efficient at all for several reasons:

- What if other colleagues might have something to add to the conversation, but are not included in the receivers list?
- If it is just a short question, the way with a screenshot etc. is not time efficient and exhausting for all participants.
- What if the information in the E-mail thread might be informative for other users in future? The would have to ask the same question again.
- ...

An alternative to using mails for problem solving would be a wiki or forum where all colleagues can collaborate. This partially overcomes the problems listed above. Since anyone who has access to such a platform will have a persistent overview about anything that has been discussed in past and will be in future. Topics can be bundled in threads or articles which allows structuring. Newcomers can use search functions and content tables to easily find content related to a specific issue. Still this approach has the disadvantage of being decoupled from the problem itself. Again lets imagine the problematic button which now will be discussed in a wiki article. First of all the user needs to know about the fact that there is a topic about this issue in the wiki. The experience shows that it is mostly not the first step to use the organizations platform to search for a solution but to ask your colleagues and Google instead - and maybe then to start with the search for alternatives. It would be the easiest way if the wiki-article is linked with the button. In this case the user would immediately see that there already is some discussion to that. And following the link would bring him without any detours directly to the related thread.

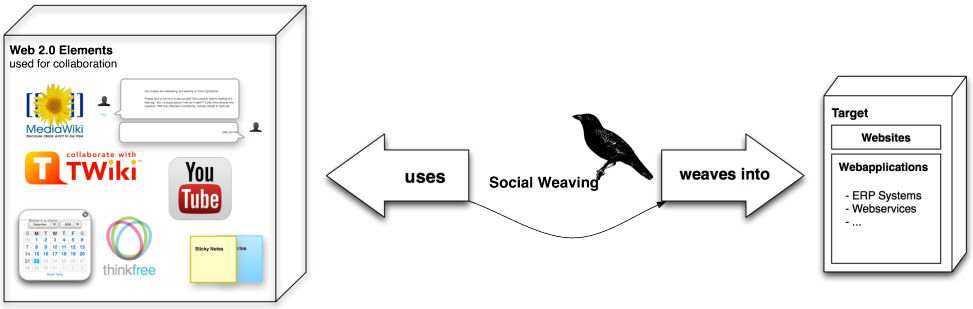

Figure 2: Basic idea of Social Weaving

So in combination we want the possibility to create some form of communication functionality - directly related to the button. And which is visible to a group of users for optionally unlimited amount of time.

Well if the web application would have an implemented comment box beneath the button, that would solve the problem. But that solution brings another bunch of disadvantages. It would require to modify the web application that includes this functionality. And we cannot add comment boxes, links, ... to any element just as precautionary measure. This attempt of solution is not an option.

What if we would have the opportunity to inject social elements directly into our web application without the need to modify it. Basically web applications run in a browser and what is displayed can be modified locally. And that is what we do. We weave social elements into the browser view and synchronize it for different user sessions. This way we reach exactly the functionality we need to solve our problem without touching the web application. We call this process Social Weaving.

Even though Social Weaving overcomes the above mentioned problems - it confronts us with new types of pretty unpleasant difficulties. Different browser types, unreliable web development styles, complex web technologies, etc. are examples for hurdles that Social Weaving has to deal with.

This thesis will show what is possible with Social Weaving and where its boundaries are.

In the following we first discuss the idea of Social Weaving based on

Figure 3: Impressive nest built by social weavers

an abstract requirements analysis of a prototype. What functionality does it need to achieve the goals we mentioned above and what are the difficulties? In the second part we are going down on a concrete level where we take the theory from the first part into action and actually explain the architecture and implementation of the prototype, Social Weaver, in detail.

3 Social Weaver - Abstract Domain Level

I know it when I see it

Potter Stewart

Even though Stewart had something completely different in mind when he used this famous phrase in front of the United States Supreme Court, it is still a quite good explanation why a prototype is useful.

We aim to create a vertical prototype. That means that our system should proof that the general idea is possible to implement. Wiegers writes that a vertical prototype should touch all technical layers to serve as a *proof of concept*[34].

First of all we start with a prototype driven requirements analysis. In Section 3.4 What is Social Weaver we discuss the requirements on an abstract domain level and put these into a relationship with the *WHY/WHAT/WHO-Model* [31].

Based on that in Section 4 we bring the requirements to a concrete domain level and furthermore explain some interesting details about the derived architecture and implementation.

3.1 The WHY-WHAT-WHO-Model

The WHY-WHAT-WHO model (see Figure 4) enables us to discuss on different layers of requirements abstraction. The purpose of the WHY-WHAT-WHO model should not be a strict separation into sections. It is much more to be seen as a context that we can refer to through this paper.

In the following all layers are described in detail and connected to parts of this thesis.

WHY Dimension

This dimension analyses the existing system or environment we build on, to define what is within a possible reach. Potential problems or limitations should be defined in context of this layer. Furthermore we should try to see what impact our system is going to have on the environment.

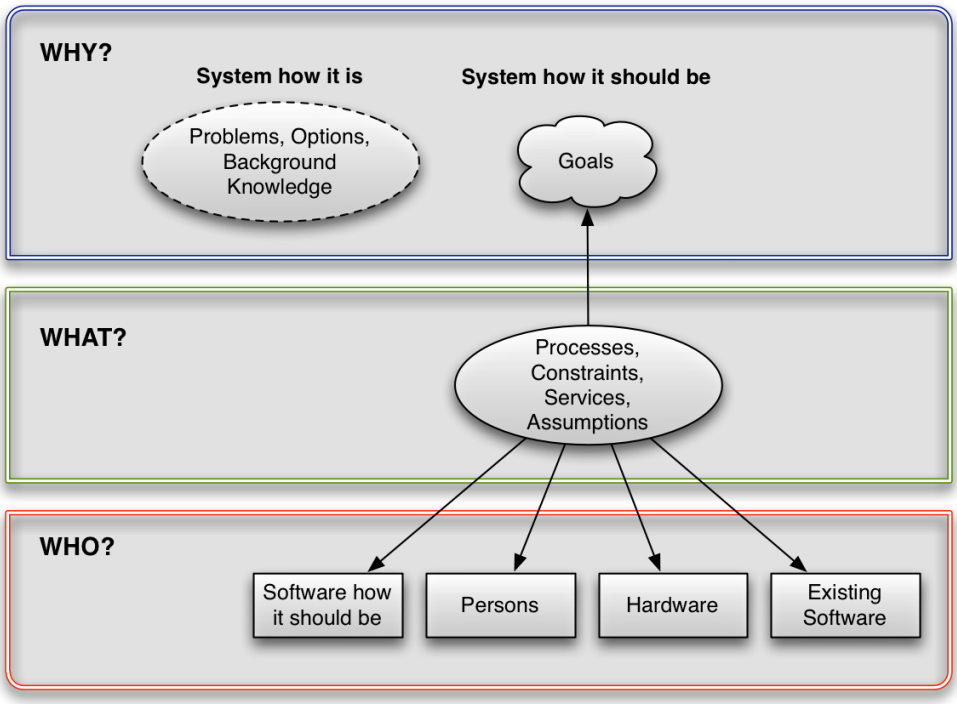

Figure 4: Three dimensions of the requirements types [31]

For this task we first of all need background knowledge which we gather in a domain analysis. Based on that we are be able to define the problem, that should be our motivation for building such a system.

Domain Analysis 3.2 is related to the WHY dimension.

WHAT Dimension

The WHAT dimension contains services, constraints and processes that are direct results from the WHY dimension and lead to the actual goal system. At this point the results from the WHY dimension should be verified if they are still applicable and possible to implement. Use Cases and concrete requirements belong to this section.

This thesis Sections 3.5 Requirements for Social Weaver and B Use Cases are related to the WHAT dimension.

WHO Dimension

The WHO dimension is about distinguishing what component has which responsibility. Components in this context don't need to be necessarily hardware components. For instance some user interaction might be classified as component. Responsibility in this context means that the component achieves the objectives ([31]).

This dimension is not applicable to certain sections. Throughout the thesis it becomes obvious what components have what functions to achieve objectives.

As example imagine the process of weaving a social element to a web view. It includes several components:

- User
- Plugin
- Web Service
- Browser
- Web view

The users interaction is needed to determine the correct web view within the browser. The browser initiates the plugin, whereupon the plugin uses information about the web view, that is provided by the browser, to create content, that is transmitted to the server.

3.2 Domain Analysis

A. Introduction

The domain analysis contains gathered knowledge about using web applications or web pages and that is related to Social Weaving. This already established environment provides more knowledge that we need for the analysis. We need to generalize it and additionally create new knowledge about Social Weaving, including meanings of processes and definition of terms. With this as root position we are be able to create requirements and define use cases.

B. Glossary

Web view describes the environment that is visible withing a browser. It may be a web application, a web page or any other displayed HTML code processed by the browser. In the context of social weaving the back end code is is not an issue at all.

Social element is some user generated content that is weaved into the web view. It is called social element because it provides some kind of collaboration possibilities. This can be a chat, comment box, file upload or just a link to external content.

Social Weaving we call the whole process of weaving some social content into a web view.

C. General knowledge about the domain

- Recognizing web elements in web views across sessions is a problem
- Web page and application architectures differ a lot
- Web views evolve with time
- User is able to uniquely identify elements in a web view
- Some web technologies cannot be analyzed (Flash, Shockwave, ...)

D. Clients

First of all we distinguish between the types of clients: user and administrators. Even though this as technical domain analysis to support the prototype development process, additionally some categories of possible users types are mentioned. This should give the reader an idea about possible real world use cases.

- User
  Basically any browser users is a potential client. All that is required besides the browser itself is a compatible plugin that supports Social Weaving.
  - Client & Consultant
    The client is using a platform (like ERP or banking system) and might ask his consultant directly posting his question in the view and location where the problem seems to be. The consultant answers directly in place. Both users have different roles. The client for instance can only post questions and not see the question of other users. Consultants can only answer and see all questions of all clients and so on.
  - Collaboration in Teams
    Several users have a communally sessions. All users can add, modify and see the social elements. This use case is useful for example when a consultant team works with a system. Questions and other kind of communication is persisted directly to the relevant items in the view.
- Administrator
  Every Social Weaving session needs an administrator who configures and keeps the web service running. Theoretically this can be also a regular user. Nevertheless this role has to be maintained to make a synchronization between user session possible.

E. Environment

For the prototype environment all users need to have a working computer with the Firefox browser and an installed plugin. Additionally the web service needs to be running on a Tomcat server.

A future goal is to support more browsers but is not covered in this thesis.

F. Similarity to other software
There is no known software that performs Social Weaving. Anyway the basic idea of annotating something is not new. Many systems provide the possibility of commenting. For instance task managers like Asana[1] or agile platforms like [2] provide flexible possibilities to comment a lot of things and upload files in any text field. Nevertheless it is not possible to annotate something that is not meant to be accessible.

Another analogical case of Social Weaving is annotation documents. There are many tools out there that you can use to annotate PDF files with comments, icons, links and so on. Compared to task mangers, we discussed above, here it is possible to annotate basically anything within the document. But still this is pretty much different from annotating any web view.

G. Similarity to other software domains
The idea to analyze web pages or web applications by its source code (meaning both server as well as client side generated HTML code) is not new. Some of these approaches were used as inspiration for the Social Weaving approach. First of all we discuss the rudiments to some comparable domains; afterwards the differences are explained and how Social Weaving differs from already existing work.

- **Web Site Re-engineering Using RMM**
  ... ([1])
- **Source Code Independent Reverse Engineering of Dynamic Web Sites**
  ...[9]
- **Flexible reverse engineering of web pages with VAQUISTA**
  ... [32]

[1] `http://asana.com/`
[2] `http://www.jetbrains.com/youtrack/`

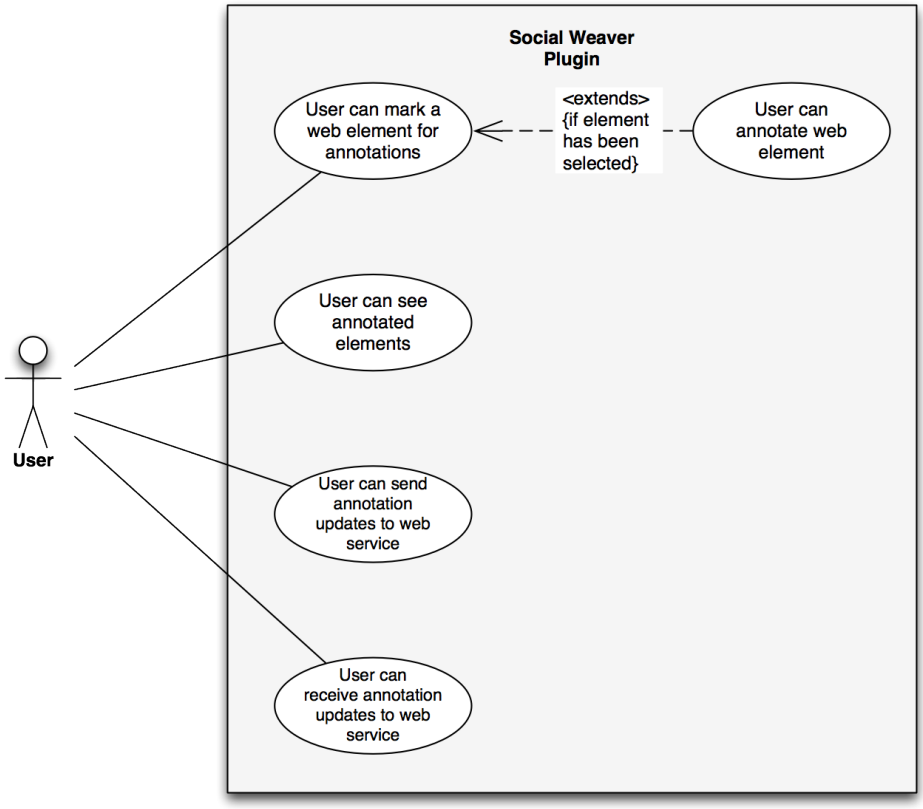

Figure 5: Use cases for the plugin

3.3 Use Cases

The purpose of this section is to give the reader a clear idea about what functions our system should provide. Use cases are categorized for the plugin (or the client) and the server side.

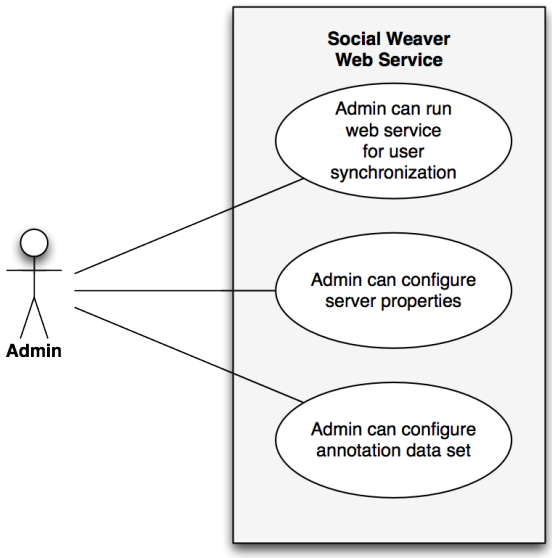

Figure 6: Use cases for the web service

3.4 What is Social Weaver

Social Weaver (SoWe) is the name of a prototype system that weaves social web features into web applications. The system consists of a Firefox plugin and the server side.

The plugin takes control of one or multiple user sessions and draws the additional content into the browser view. The server application synchronizes with each plugin and distributes updates between several clients.

For a better understanding lets step through a generic use case where a user just opens a web application and modifies some content. The use case enumeration is related to the Figure 8.

1. The user opens a web application
2. The SoWe-Plugin sends a notification to the server with all necessary information like user identifier, time stamp, ...
3. After the server receives the plugin message it synchronizes it with its current content in the database

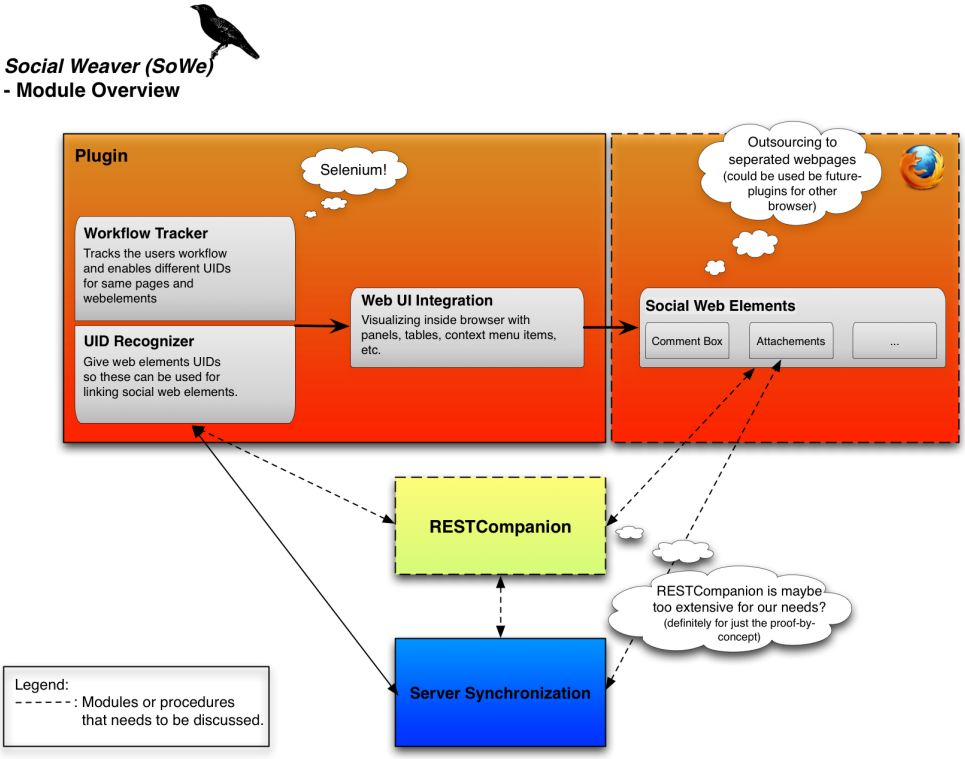

Figure 7: Social Weaver Module Overview

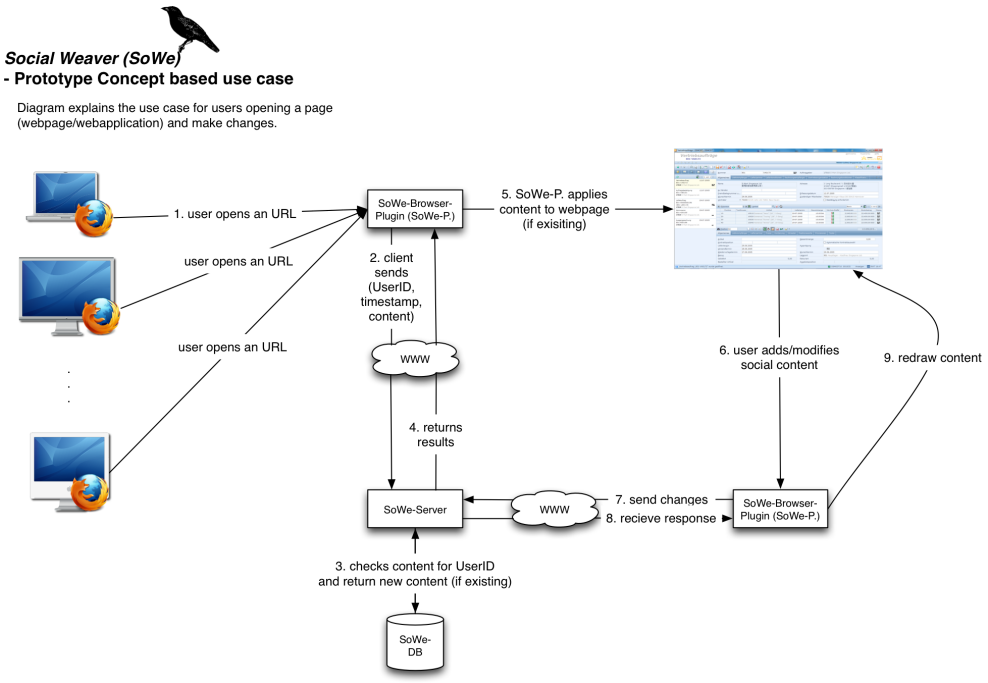

Figure 8: Social Weaver Prototype Use Case

4. The server application responses to the plugin client with content data if some exists

5. The plugin uses the content information from the server to insert all social web elements

6. The user decides to make some changes to the social web content (e.g. adds a comment or creates a new comment box)

7. Again a notification is being sent to the server with containing the changes

8. Server synchronizes the updates and responses

9. Plugin redraws the synchronized content

3.5 Requirements for Social Weaver

The general goal for Social Weaver is to weave social web 2.0 features into web-based applications. Since this is a broad requirement and impossible to be applied to any web application right from the beginning; it is necessary to break it down for the prototype.

More specifically the primary goal should be to get a system, that weaves one social web feature into a specific web application. Social Weaver has to be designed in a modular way, so that it is possible to add more social media features, support multiple platforms and more web applications. Now that we have a rough idea what SoWe is going to be, lets list some concrete high-level requirements:

1. Browser plugin that supports a comment box
2. Server application that stores and synchronizes data that it receives from different client-plugins
3. Data format for storing and processing data for social web content
4. Communication protocol between plugin and server

With these requirements we can start to specify our enlisting in detail:

3.5.1 Browser Plugin

In the following we define requirements on an abstract domain level according to [31]. A specification to a concrete domain level follows in Section 4.2, where we have specified what technologies to use.

As already mentioned the main requirement is that the plugin supports a comment box. That means that the browser has to display a comment box that is related to specific web element. For example in an online calendar an user adds a comment box related to an appointment that he wants to discuss in detail. Because it should be possible to add multiple comment boxes to any web element, we cannot just drop a box

inside the user view, overlapping other interesting parts of the web application. Hence we have the requirement to make additional content visible to the user without interfering with the view on the original content. Possibilities would be fold/unfold-windows or just using small icons as references in the original view and outsource additional social content in external windows.

Of course the plugin needs to be able to communicate to the server application as well. (The server application is explained in the next Section 3.5.2). First of all the plugin needs to receive data that it print to the screen. Secondly changes made by the user has to be reported to the server. Because we are distributing the information between several users, there is also a need for synchronization. User updates may not overwrite updates made by other users etc.

The parser framework contains application programming interfaces that create and parse the content of our tuples. This way it is easier to add plugins for other browsers for instance. The data in the content-part of our tuple should have a uniform format no matter what web application or browser is in use. The server application doesn't need to be aware about the environment the plugin runs in - it manages the social web content independently.

Another tricky and important point is the interaction with the web application. Most such sites are dynamic and there exists no static URLs we can refer to. And it is not certain that the same element, that two users refer to in their independent sessions, has a comparable identifier. This issue definitely needs to be handled specifically for any web application. The good news is that this only affects the plugin. The server application just needs clearly defined identifiers. As a solution for the plugin we need the possibility to use scripts for identifying elements. For example a script that supports the Google calendar is injected to make the plugin identify same appointments in different user sessions. This requirement is probably the vastly problematic one because it prevents a general usage of Social Weaver.

Lets summarize all the requirements we gained in this section:

1. Displaying and managing social elements related to a specific web

element

2. Managing several social elements without disturbing the view of the original content
3. Communication with web service
4. Creating Anchors
5. Creating content in uniform sending format
6. Parsing content from incoming messages
7. Identifying web elements across different user sessions

3.5.2 Server Application

The server applications primary requirement is to synchronize different user sessions on one or multiple web applications. A user session is defined within the plugin (which doesn't mean a plugin can manage only one session). The server basically receives messages from different sessions, synchronizes them and distributes the most current state to all sessions. To establish a loss less synchronization every message contains a time stamp.

We are assuming that every message contains an user identifier, a time stamp and an unique identifier for an element within the web application. This Anchor is the unique identifier for a single user action. For example if a user adds a comment to an already existing comment box that is related to an appointment in a calendar, the server receives the users identifier, the time stamp for the modification and an identifier for the appointment in the calendar. With this information the server can check its database for the comment box and add the new comment.

It is important to remember that the server only uses the received data as identifier. All actions are completely independent to the web application.

Also we may assume that the received message have the same Anchor form as discussed in the previous section.

```
(user identifier, time stamp, content)
```

The content part from the Anchor needs to be in an uniform format that has been generated by the plugin. So even the browser type doesn't matter to the server. The server has to be able to parse the content package and to create a new one that can be parsed by our plugins.

So the requirements for the server application are:

1. Offer service that receives messages from plugin-clients
2. Synchronization for requests from different user-session
3. Persist updates into a database
4. Keep the server application independent to weaved-into web application
5. Keep the server application independent from the plugin
6. Parse incoming messages
7. Create outgoing messages

3.5.3 Social Weaver - Script Support

The support for external scripts is essential for a generic usage of Social Weaver. The reason why script support is extracted into its own section, is that it should be decoupled from the server and plugin that were discussed before.

The underlying problem is the problematic identification of elements of a web view. There is simply no generic way of identifying elements in the users view across all web sites and applications. For that reason we need an extendable method to support more websites and applications. This could even mean that third-parties could support their own systems by just adding the script without the need to modify Social Weaver directly. In this section we briefly discuss what the purpose of such scripts is in detail and what requirements we have to fulfill.

The term *script* in our context should contain only information that is needed by the plugin to identify an element. Let us consider the Google calendar example once again. The case where we want to match the same appointment field across different user sessions brings the problem that there is no identifier for the element itself. To the user it is obvious to identify it because of the appointment name, date and time. And those parameters could be just the information we need to extract into our script. How this looks like in detail is discussed in the concrete domain level section.

The usage of scripts should be related to one or a set of URLs. This affects mostly the root URL of a server. But might be used for sub parts of a web page or application. As example a script related to *http://www.opensource.org/* is applied to all sub pages like *http://opensource.org/docs/board-annotated*.

But it might be of use to have a special matching procedure for sub pages. In that case a script for *http://opensource.org/faq* would overwrite the more general script.

A set of URLs could be used for scripts that are applicable for many websites.

The work flow when a script is used and when the default matching procedure that comes with the plugin is quite straightforward (see Figure 9). When opening a new URL then the plugin should check whether there is a script for that case and depending on the search results proceed with the script or default matching procedure.

On the abstract level we have the following requirements:

1. Container of all necessary information for element matching
2. Decoupled from browser plugin and server backend
3. Syntax that is easy to read and write
4. Extension of the plugin with parsing methods
5. Default matching procedure should be provided (so the overall functionality is not limited when no scripts exist)
6. Scripts should be related to a single or a set of URLs

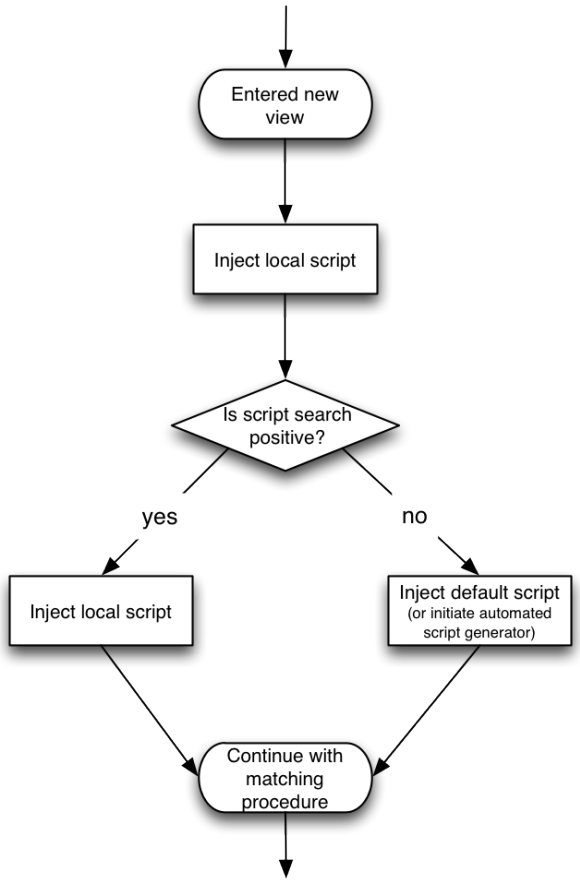

Figure 9: Work flow for Script Using

4 Social Weaver - Concrete Domain Level

4.1 Social Weaver - Firefox Plugin

This section briefly explains what technologies we use for Firefox plugin development and describe in detail how the Social Weaver plugin is implemented.

Firefox

Firefox is a free web browser that has been released 2004 by the Mozilla Foundation[3]. It is being distributed under multiple licenses under Mozilla Public License (MPL)[4], GNU Lesser General Public License and GNU (LGPL)[5] General Public License (GPL) [6].

The reasons why we chose Firefox as prototype environment are the high distribution of the browser and an easy extend ability with plugins, extensions and so on.

Firefox Plugin Development

To improve readability of the coming Section 4.2 Requirements for the Plugin, we discuss some aspects from the Mozilla Add-on SDK (Version 1.13) [7]. Readers who are not interested to much into technical detail or are familiar with the technologies can skip this section.

The used methods aren't just explained independently but brought into context to our prototype planning so it becomes clear what purpose they have.

The Add-on SDK allows to create add-ons for the browser using the most common web technologies (like HTML, CSS, JavaScript, ...). Furthermore it provides a Low-Level-API and a High-Level-API set. The most

[3] `www.mozilla.org`
[4] `http://www.mozilla.org/MPL/1.1/`
[5] `http://www.gnu.org/licenses/lgpl-3.0.de.html`
[6] `http://www.gnu.org/licenses/gpl-3.0.html`
[7] `https://addons.mozilla.org/en-US/developers/docs/sdk/latest`

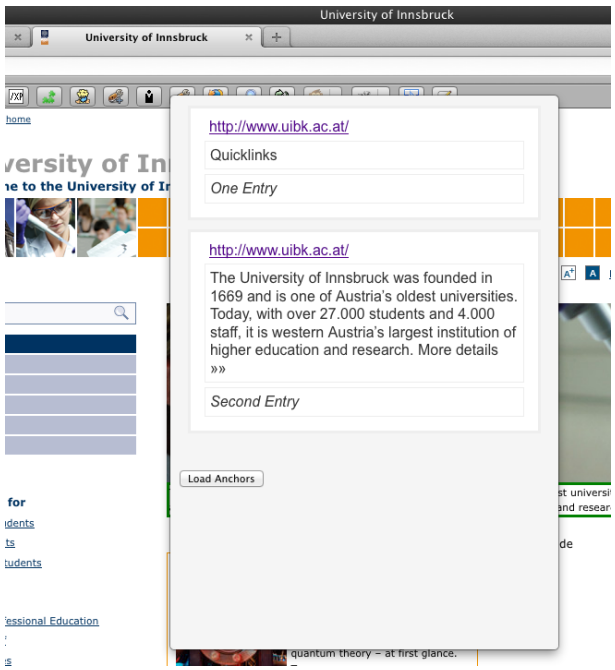

Figure 10: An example for a *panel* that shows a list of annotations

important interfaces that are being used for our prototype are High-Level-Interfaces and are explained in the following.

Panel

A *panel*[8] is very flexible dialog window. Its appearance and behavior is specified by a combination of a HTML and a JavaScript file. Additionally a CSS file might be used to change the look even further. The limitations of a panel are the limitations of the mentioned technologies. A panel is meant to be visible temporary and they are easy to dismiss because any user interaction outside the

We use the *panel* for getting user input, displaying information (like in screen shot 10) and to integrate our social media web elements.

Actually the flexibility of *panel* is the reason why our prototype is able to support social weaving for basically any web element that

[8] `https://addons.mozilla.org/en-US/developers/docs/sdk/latest/modules/sdk/panel.html`

can be represent in HTML code. Still it is necessary to embed the external HTML code which may leads to boundaries and difficulties.

Simple-Storage

This module[9] is an easy to use method to store basic properties (booleans, numbers, strings, arrays, ...) across browser restarts.

With an operation like

```
1 var ss = require("sdk/simple-storage");
2 ss.storage.myNumber = 41.99;
```

we store a number like an object and can it access just as easy like that. The price for such a simple usage is paid with high limitations. For instance searching is basically not possible. Nevertheless we can store an array and search the array.

That is exactly the way how we store our annotations for our prototype. More details are provided in the next section.

Page-Mod

The *page-mod*[10] module enables us to act in a specific context related to a web page. Then it becomes possible to attach Java Scripts to it and to parse or modify certain web page parts.

In our context we are going to use *page-mod* to parse the HTML code to find elements that can server as anchors for annotations. And of course to find elements that are already annotated.

Widget

The module that is called *widget* [11] is simply an interface to the Firefox add-on bar[12]. It is possible to attach *panels* and trigger operations by clicking the *widget*.

[9] https://addons.mozilla.org/en-US/developers/docs/sdk/latest/modules/sdk/simple-storage.html

[10] https://addons.mozilla.org/en-US/developers/docs/sdk/latest/modules/sdk/page-mod.html

[11] https://addons.mozilla.org/en-US/developers/docs/sdk/latest/modules/

Figure 11: Example for a widget serving as activation button

We use a widget to switch between different modes (see coming Section 4.2.1 for more details about how the mode-system works).

Self

Self[13] provides access to add-on specific information like the Program ID[14], which is important for an official distribution of the add-on. More meta information like the name or the version are accessible via the *self* module. Also bundled external files are integrated by *self*.

Even though it is an important module and part of the plugin - there is no specific usage relation to the system we use.

Notifications

---

sdk/widget.html

[12] https://developer.mozilla.org/en-US/docs/The_add-on_bar

[13] https://addons.mozilla.org/en-US/developers/docs/sdk/latest/modules/sdk/self.html

[14] https://addons.mozilla.org/en-US/developers/docs/sdk/latest/dev-guide/guides/program-id.html

This module[15] displays toaster[16]-messages[17] that disappear after a short time.

We use these to keep the user informed without bothering him to much by forcing him to dismiss trivial notifications.

Request

This simple to use but yet powerful module *request*[18], lets us perform network requests. Once we create a *Request* object we can specify whether it is a GET, PUT or POST request. These request types are specified by the REST standard so any web service that supports REST is able to interact with this module[13]. The response from a server is directly accessible like any other JavaScript object.

We are going to use *request* for our communication with our synchronization web service. This includes sending updates, made with the plugin instance, to the server and receiving updates, that were made in other sessions or with different plugin instances, from the server.

JQuery

jQuery[19] is a free JavaScript library under the MIT License[20] that offers many functions for modifying DOM trees. It has been released 2006 in context of a BarCamp[21] in New York.

Even though this library is not a part of the Mozilla Add-on SDK it is being heavily used by it. Basically any operation that changes or traverses

[15] `https://addons.mozilla.org/en-US/developers/docs/sdk/latest/modules/sdk/notifications.html`

[16] Toasters are commonly called notifications that just appear, or slides in the users view - like a toast hops up.

[17] `http://en.wikipedia.org/wiki/Toast_(computing)`

[18] `https://addons.mozilla.org/en-US/developers/docs/sdk/latest/modules/sdk/request.html`

[19] `http://jquery.com/`

[20] `http://www.mit.edu/`

[21] `http://en.wikipedia.org/wiki/BarCamp`

the HTML code (like changing the background color of web elements) is being reached with jQuery.

The reason why jQuery makes it so easy to handle operations on HTML code is based on the selector that searches the DOM tree. Most jQuery operations are based on elements in the DOM tree. With the selector `jQuery()` (which can be equally written as `$()`) any element in the DOM tree can be used to create a jQuery object. This object can be used to perform modifying operations on it. For instance lets step through the following short operation:

```
1 $('.user-name').click(function(){
2    this.empty();
3 });
```

In line one we use the `$` selector to find any element in the DOM tree that has the id `user-name`. Since we use the jQuery selector, any found element can be handled as a jQuery object. This way `click(function(){...})` appends a click observer to all elements. Any time one of those elements is clicked by the user, this triggers the function. This function includes `this.empty()`, which only removes all content inside the element.

Another and more related example of jQuery usage is used in the selection part of the plugin:

```
1 $(":visible").filter(function(index) {
2   if(this.content()){
3       return true;
4   }else{
5       return false;
6   }
7 }).mouseenter(function() {...}
```

First of all we request all elements that are visible to the user by using `$(":visible")`. Because it is unlikely that a user tries to select an element without any content - we filter all elements from the result set that are empty. (Such elements can be placeholders, flexible empty space and so on.) Now that our results contains mostly relevant elements to the user, we append `.mouseenter()` to recognize when the courser is placed above the element. The skipped function itself contains operations to recognize

more user operations and element analysis. To discuss more code in detail would be beyond the frame.

We only have seen a brief overview about jQuery operations. To fully understand how the prototype works and to operate with the script support it is advised to gather a better overview using for instance the official W3 introductions: `http://www.w3schools.com/jquery/`.

4.2 Requirements for the Plugin

Let us recap what requirements we gathered in Section 3.5.1 on the abstract domain level [31]:

1. Displaying and managing social elements related to a specific web element
2. Managing several social elements without disturbing the view of the original content
3. Communication with web service
4. Creating Anchors
5. Creating content in uniform sending format
6. Parsing content from incoming messages
7. Identifying web elements across different user sessions

In the following concretion we apply the abstract requirements to our environment which is the *Mozilla Plugin Development SDK* [22]. In the following we handle each and every requirement, mentioned above, separately:

4.2.1 Displaying and managing social elements related to a specific web element

Obviously "Displaying and managing a comment box related to specific web elements" consists of multiple sub requirement that we need to distinguish.

Before we are able to annotate something, we first of all need a function to select or recognize a web element the users cursor points to (check

[22] `https://addons.mozilla.org`

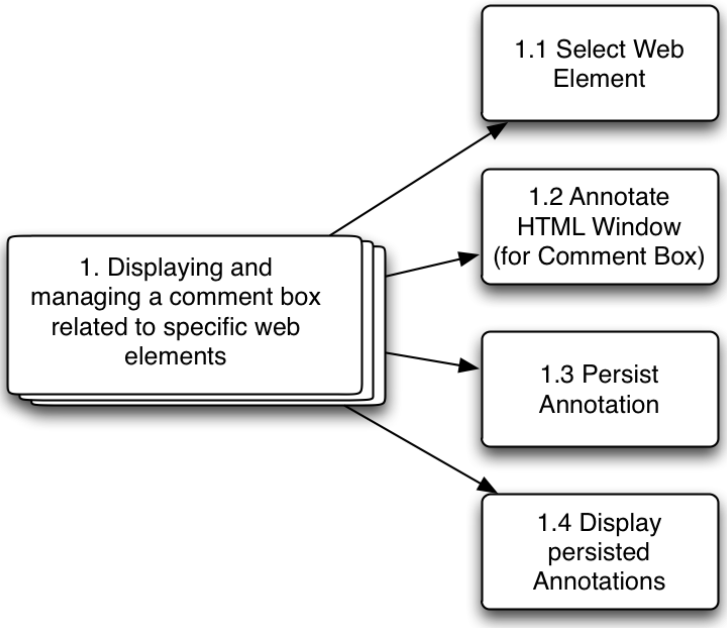

Figure 12: Partition of the first plugin requirement to sub requirements

1.1 in Figure 12). *Selecting* in this context means that we analyze the Document Object Model (DOM)[23] tree. The selection itself is easy to implement using the function *mouseenter* and *on('click')* from jQuery library[24]. It becomes problematic to find this element again without any user interaction. Therefore we need to set well chosen parameters that are used to determine the element. Next time we need to find the element - only the parameters can be used to find the element in the DOM tree. This issue is topic in the Section 4.7 Social Weaver - Script Support.

Theoretically it could be possible for user to select any element in the web view - but practically this would make the selection procedure confusing for development as well as for the user. Therefore we apply some filters. Elements like empty boxes, placeholders and so on are not selectable. But still it should be clear to the user what he might select.

To achieve this functionality we create thin rectangles around every

[23] `http://www.w3.org/DOM/`
[24] `http://jquery.com/`

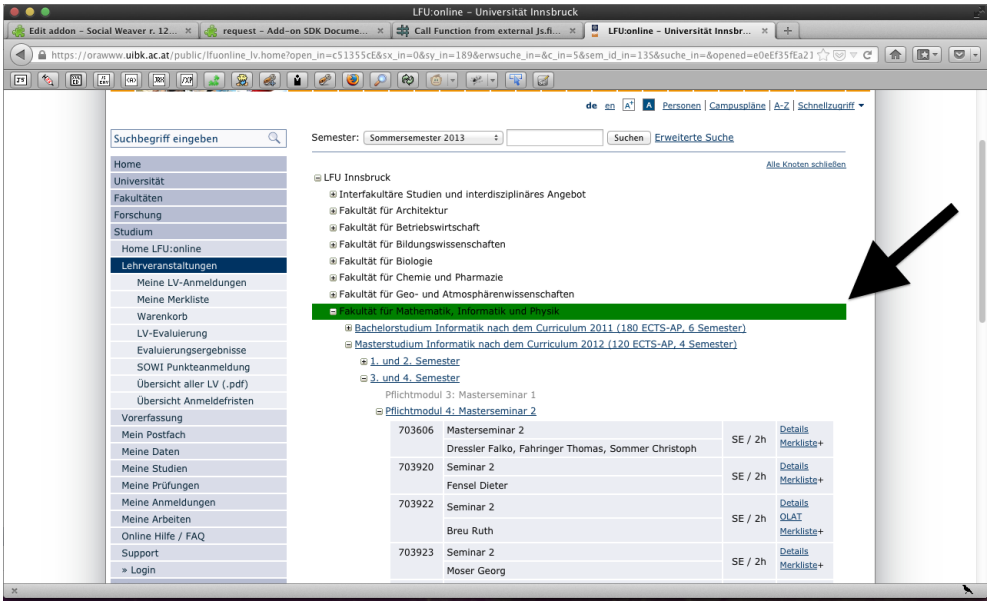

Figure 13: Rectangle shows that the underlying element is possible for annotation

element that is an selection option. These rectangles only appear for time the plugin is in selection-mode and uglify the web view just for a certain time. More on the different mode types in 4.2.2 User View Management .

Now that we can locate a specific web element we may annotate some social web element. For reasons of flexibility and simplicity we just annotate a HTML window (check 1.2 in Figure 12), where we can inject any external HTML code. The Mozilla SDK high-level APIs [25] offers all necessary tools to insert a HTML box as a *Panel*[26].

The annotation anchors are visible to the user in form of a colored background that we create by modifying the DOM tree. If the user clicks on such an element the already existing panel is opened.

To decouple our annotated data (like anchors, annotations, ...) from the actual synchronization, which is covered later, we want to use a stor-

[25] `https://addons.mozilla.org/en-US/developers/docs/sdk/latest/modules/high-level-modules.html`

[26] `https://addons.mozilla.org/en-US/developers/docs/sdk/latest/modules/sdk/panel.html`

ing system that is also provided by the Mozilla SDK (see 1.3 in Figure 12). The high level API *simple-storage* [27] enables us to store all information we need and recall them. Just the synchronization mechanism should modify this data set. All displaying procedures should be outside of server communication reach. This means that if an annotation is created, the changes are written locally into the plugin local data storage. And only after the persistence is complete, the synchronization procedure is initiated. While synchronizing, the changes are transmitted to the server.

The last sub requirement is to re display existing annotations (check 1.4 in Figure 12) from our *simple-storage*. Besides using the same techniques for drawing content and retrieving it from the storage we need to match the web page content to our saved annotations. For that we use a matcher instance that checks the DOM tree for IDs that we are already using.

This is actually only trivial on a very simple basis. Let us assume that we have more than one element attached to the same web element. Or we have different user sessions and/or include a work flow so that we need to distinguish the same element for different instances of the web page. Than it becomes quite complicated to generate IDs that we can rely on. Nevertheless these issues just affect the way we assign IDs to elements and how we retrieve them. The requirement 1.4 is just about matching existing IDs to a web page.

As already mentioned we use a matcher that checks the DOM tree for IDs. In case we have an anchor in our *simple storage* then we modify the web page HTML code similar as we did for requirement 1.1. Visual differences are that we don't modify the background of an element but generate an rectangle around it instead (see screen shot 14).

This way we are able to show the user which elements are annotated. Of course without further information it is not obvious what is annotated exactly. What we need is a easy to access functionality so that the user can find out what the annotation is.

For that reason we modify the above mentioned matcher class to gen-

[27] `https://addons.mozilla.org/en-US/developers/docs/sdk/latest/modules/sdk/simple-storage.html`

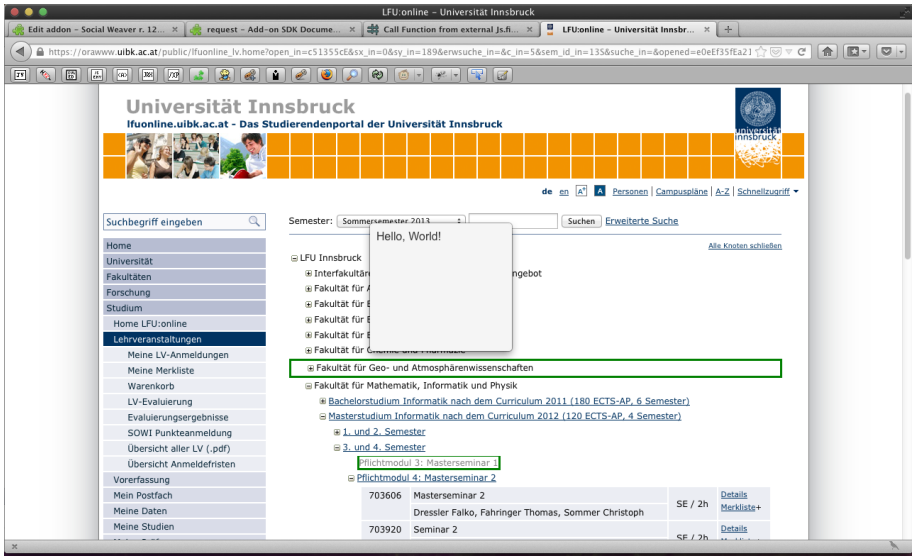

Figure 14: Rectangle shows that the underlying element is possible for annotation

erate a panel in case the user performs a *mouseenter* operation. This panel should show a brief version of the attached social element. In our case it could be the name of the context the comment box is related or the names of the attendees (our example screen shot just print outs "Hello, World!" 14).

4.2.2 Managing several social elements without disturbing the view of the original content

This requirement is not directly about functionality but should ensure a positive user experience. It is possible to attach annotations to nearly any element in a web application. This is a lot of potential additional footage. Nevertheless the user needs to be in the position to navigate like usually within the application.

Basically there are two options to guarantee such a requisition. Either we minimize the overhead that we display into the web view or we display additional information only at the appropriate time. To achieve the best results we combine both possibilities by introducing the mode

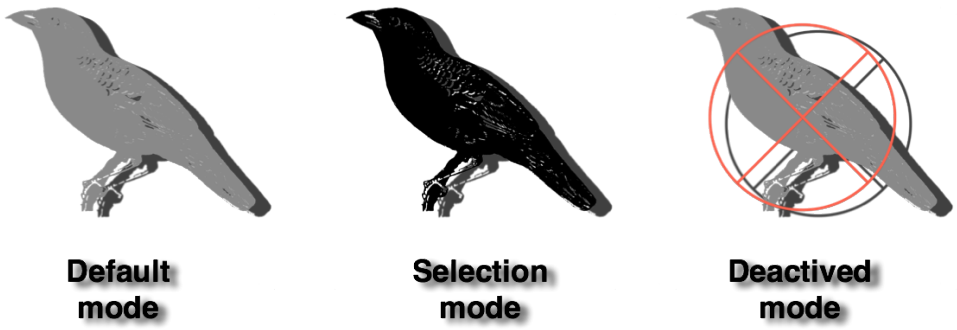

Figure 15: Widget representation for different modes

system:

The plugin has several modes that support different functionality and show different content. Those modes can be switched by clicking the widget in the add on bar. Every mode has its own logo, see Figure 15.

We distinguish the following modes:

1. Default mode

   In the default mode the plugin runs passively. It doesn't disturbs the user but runs synchronization and matching procedures in background. When the user opens a new web view, the plugin checks its database for annotations that belong to the view. If a positive match is found, it is into the users web view.

   The navigation of the browser is as usual, except for the annotations that include a click handler that triggers the function for opening the social element.

2. Selection mode

   The selection mode interferes eminently with the navigation and the representation of the regular browser view. Activating the selection mode marks all selectable elements with rectangles around it. Moving the cursor around the web view additionally marks the element beneath the cursor to visualize what element is marked when

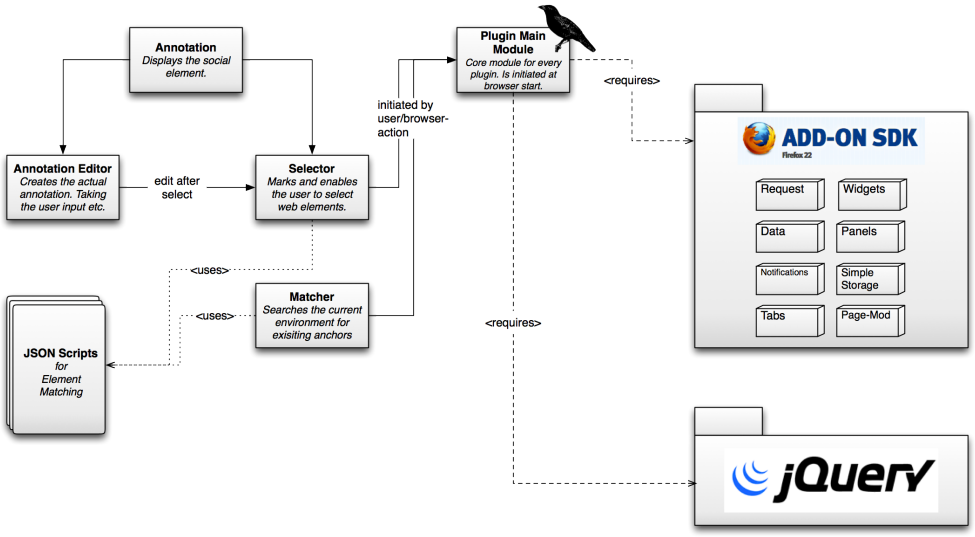

Figure 16: Architecture of the Firefox Social Weaver Plugin

clicked. In this mode clicking links, buttons and so on, doesn't trigger the functionality of those elements but instead select them so an annotation might be created.

Disabling the selection mode clears the view from the previously mentioned marks. Only successfully created annotations remains persistent.

3. Deactivated mode

Deactivating the plugin entirely disabled any functionality. This can be useful in worst cases like that the plugin prohibits regular navigation or the annotation marks interfere with the users view.

The mode system could be a starting point for extending Social Weaver for multiple user sessions in one plugin context. Or providing securely authenticated sessions that can only be activated when correct credentials are provided.

```
1  var sync = Request({
2        url: 'http://localhost:9998/anchor',
3        onComplete: function(response) {
4          for(var i = 0; i<response.json.length; i++){
5              var r = response.json[i];
6
7              newAnchor = new Array(r.anchorURL,
8      r.ancestorId, r.anchorText);
9              var newAnnotationText = r.annotationText;
10             handleNewAnnotation(newAnnotationText,
11     newAnchor);
12      };
13    }
14 });
15 sync.get();
```

Figure 17: Sample JavaScript code for retrieving JSON objects from a web service with a GET REST request

4.2.3 Communication with web service

To share our comments or annotations with other users we need a server side synchronization procedure. This section is only about the requirements that are related to the plugin side. (Details to the server side are discussed in **??** **??**.)

The first step to achieve this goal is to establish a communication between the plugin and a web service. For this purpose we are going to make use of the *request* module from the Mozilla Add-on SDK. It provides an easy to use JSON[28] and REST([13]) assistance.

We split this into the following sub requirements:

Plugin receives updates from server
: What the plugin needs to know from the server is a set of Anchors. Those Anchors contain information like the author identification, a time stamp and of course the content. (see **??**) So at this point we assume that our server provides a set formatted in JSON. The plugin generates a request to retrieve this data.

[28] `http://www.json.org/`

This goal is surprisingly simple to achieve. In the sample code 4.2.3 we just need to specify the URL of the web service and we are able to access the JSON objects right away exactly like JavaScript objects. Then we use the JSON objects to create an anchor entity and use the existing `handleNewAnnotion(newAnnotationText, newAnchor)` method to store it in our *simple-storage* list.

Our prototype is a proof-by-concept system, therefore we keep the synchronization really simple. Instead of checking for new annotations and match them with the already existing data, we just rewrite our local plugin data set with a copy from the server. This technique could easily lead to corrupt and inconsistent data sets. But since the prototype is not meant to be used for confidential data or in any real world scenario at all - we just take the risk.

Plugin sends updates to server

When a user creates a new annotation or modifies it - the plugin should send an update to the web service immediately. Again we set up a method using the *request* module.

How the data format looks like and how we parse incoming messages, is reviewed later (in the Sections **??**, **??** and **??**.)

4.2.4 Creating Anchors

We already mentioned anchors in the context of transmission. This section covers how those anchors are created at the plugin before sent to the web service. The first step in the creation of an anchor happens in the *selector* after the user clicked an element. After this happens, every rule from the current script is loaded and executed using the clicked element as input. The results are stored in an array we call payload. Once this process is finished we pass the payload with the current URL back to the *main module*:

```
1 event.preventDefault();
2         self.port.emit('show',
3           [
```

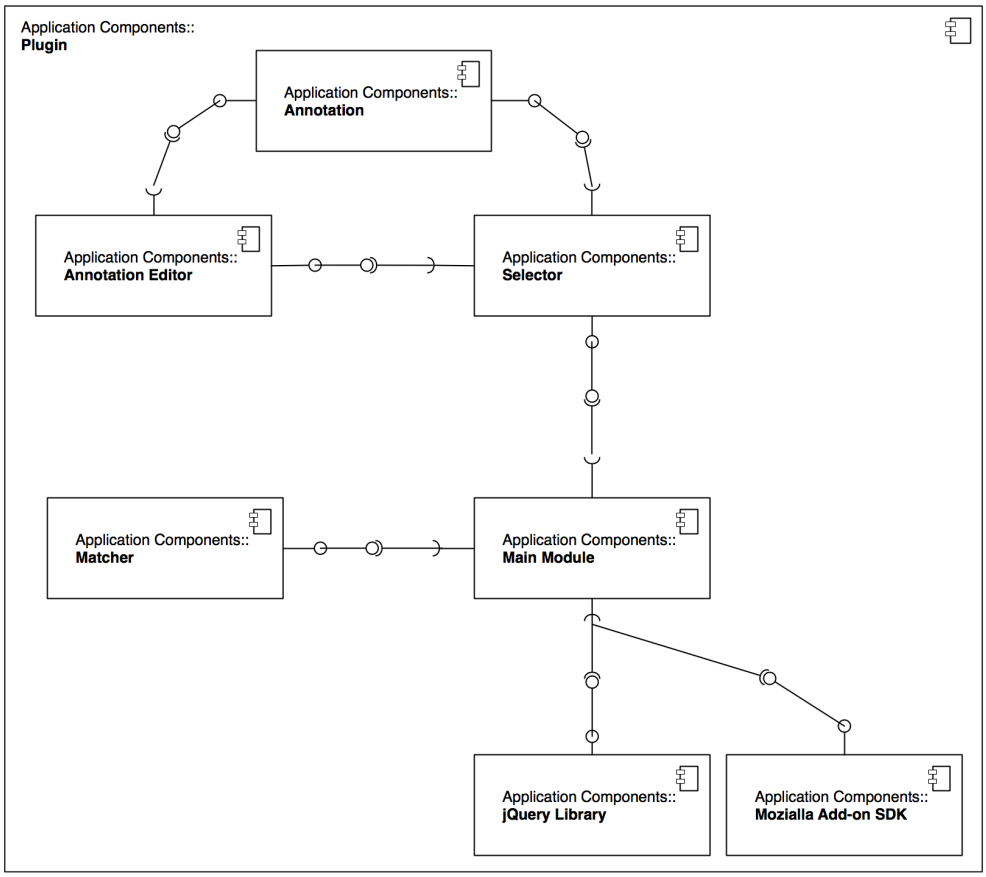

Figure 18: Component diagram for the plugin

```
4               url,
5               JSON.stringify(arr.payload)
6             ]
7     );
```

How the results contained by the payload look like, is discussed in the next Section 4.2.5. The *selector* is initiated in the *main module*, providing a worker that is listening for the emission called *show*.

```
1 worker.port.on('show', function(data) {
2           annotationEditor.annotationAnchor = data;
3           annotationEditor.show();
4         });
```

From here we pass the anchor information to the *annotation editor*. The editor is responsible for storing the user input, that defines the social element. At this point we just assume that, no matter what social element is annotated, it generates a string like value that we can use for processing. So the *editor* extends the anchor data with the social element content.

Now that the annotation and its anchor is complete we call:

```
1 function handleNewAnnotation(anchor) {
2   var newAnnotation = new Annotation(anchor);
3   simpleStorage.storage.annotations.push(newAnnotation);
4   synchronize();
5 }
```

Temporarily we persist the new annotation in *local storage*, and then start the synchronizing with the web service. The annotation is dismissed in case the synchronization fails.

4.2.5 Creating content in uniform sending format

The previous Section 4.2.4 showed how an annotation and its anchor is generated. Now the content of those entities is topic.

Again the first step in creating an anchor is to gather results on the clicked element according to the script. To illustrate that lets show this on an example in Figure 19. Next we use the following script:

```
1 {"rules": [
2         {"doc_location": "document.location.toString()"},
```

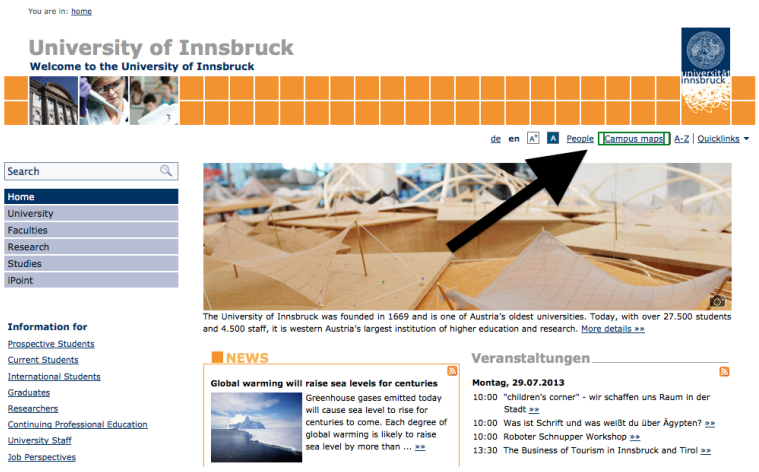

Figure 19: User selects the element pointed by the arrow

```
3          {"element_content": "matchedElement.text()"}
4 ]}
```

Executing this script on the web element results in the payload below:

```
1  [
2   {"doc_location":"http://www.uibk.ac.at/"},
3   {"element_content":"Campus maps"}
4  ]
```

Moreover we add the current URL to the annotation, which is redundant in that case with the rule *doc_location*, but not necessarily have to be in different cases. Thereafter we pass the coming information array to the *editor*:

```
1 {"annotation":[
2    {"url": "http://www.uibk.ac.at/"},
3    {"payload":[
4       {"doc_location":"http://www.uibk.ac.at/"},
5       {"element_content":"Campus maps"}
6   ]}
7  ]}
```

The *editor* processes the user input and transforms it to a string that is added to the annotation. To keep the previous example tangible, lets assume that the user creates a link to some wiki upon the element like:

`http://www.uibk.ac.at/internalwiki/faqs`. Consequently the updated annotation contains something like:

```
{"annotation":[
   {"url": "http://www.uibk.ac.at/"},
   {"payload":[
      {"doc_location":"http://www.uibk.ac.at/"},
      {"element_content":"Campus maps"}
  ]},
  {"social_element":"http://www.uibk.ac.at/internalwiki/faqs"
     }
 ]}
```

This JSON array has the form, ready for being transmitted to the web service. The server side uses the data to create anchor and social element entities, but doesn't modify the values itself, unless it is an update.

4.2.6 Parsing content from incoming messages

Basically parsing incoming messages is quite the analogous opposite to the creation and sending procedure in 4.2.5. After the web service responds to the plugin request for updates, the incoming message is an JSON array of annotations. Formerly we have seen an example for such an annotation like:

```
{"annotation":[
   {"url": "http://www.uibk.ac.at/"},
   {"payload":[
      {"doc_location":"http://www.uibk.ac.at/"},
      {"element_content":"Campus maps"}
  ]},
  {"social_element":"http://www.uibk.ac.at/internalwiki/faqs"
     }
 ]}
```

Once the annotation list is received by the array, it is split and the single annotations parsed into its pieces: *url*, *payload*, *social_element*. Those are formed to an *Annotation* object and persisted in local storage.

```
function updateAnnotation(anchor) {
  var newAnnotation = new Annotation(anchor);
```

```
3     simpleStorage.storage.annotations.push(newAnnotation);
4     updateMatchers();
5   }
```

At last the matchers need to be updated, so the new annotations are checked on coming web view.

4.2.7 Identifying web elements across different user sessions

Some of the previously discussed requirements were prerequisites, so finally the most challenging requirement can become subject. For matching web elements we need an annotation that has been created by a user selection. This process has been described in Section 4.2.4. This annotation has to be synchronized with the web service so other users are able to see the annotation as well. Formerly this was topic in Section 4.2.5.

In the first place the element identification is done by the matcher. Every time a new web view is opened the actual URL is passed to a matcher worker. A matcher worker is a thread-based instance of the matcher module. Using workers enables the plugin to run multiple matching procedures parallel (which is useful, for instance, when using tabs).

Once the matcher is triggered by opening a new web view the following algorithm take place:

```
1  boolean positiveMatch = true;
2  for every element e in currentUrl{
3    for every rule r in script.get(currentUrl){
4      currentResult = r.execute(e);
5      payloadResult = localStorage.getScript(currentUrl).
           getRuleResult(r);
6      if(currentResult != payloadResult){
7        positiveMatch = false;
8        return;
9      }
10   }
11 }
```

Informally the algorithm iterates over every element in a new web view. This web view belongs to a certain URL that we use to determine what rule script to use. In case that no specific script is provided, a set of

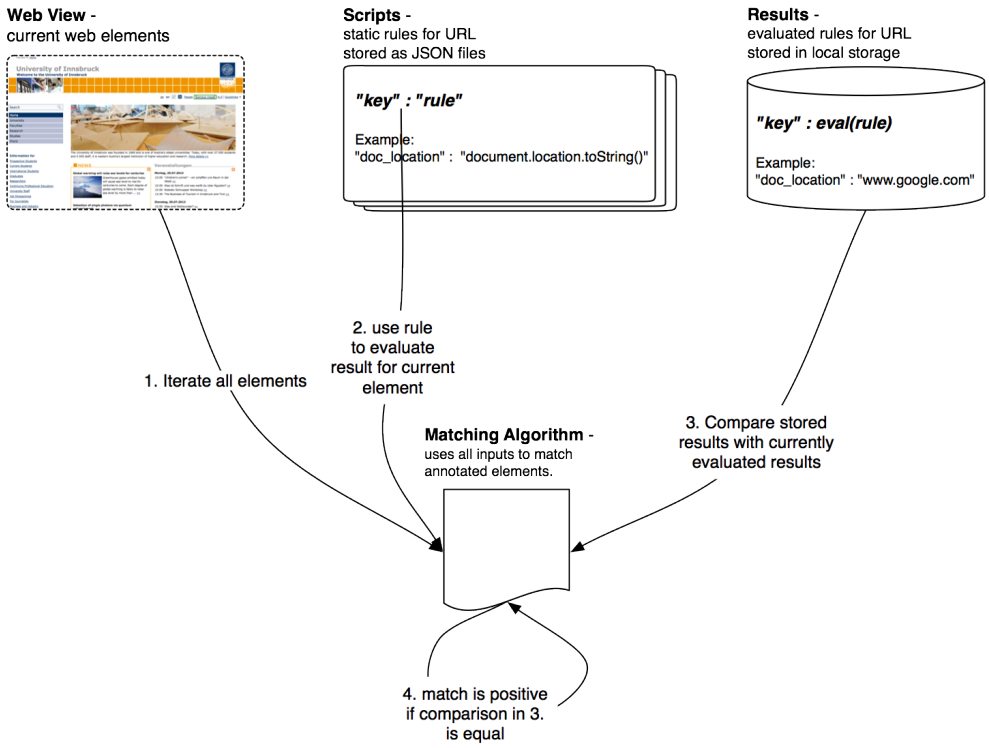

Figure 20: Visualization for the different inputs for the matching algorithm

default rules is established. For every and each element we execute the rules from the script and compare the results to possibly existing annotations. From the existing annotations we store the evaluated rule results. Therefore if an web element has the same results for all rule evaluations as an stored annotation, we assume that the current web element is an anchor that is related to the stored element.

Since this issue is a little bit confusing, lets explain what sources are used for decision making. See Figure 20. As sources for element matching we have the current web view and all its displayed elements, the set of static rule contained by the scripts and the local storage that contains existing annotations. In the following every numbered step on Figure 20 is explained in more detail:

1. We simply start with the mentioned iteration over all current web view elements.

2. For each element all rules from the static script are executed. The results are stored temporarily just for the period of comparison in the next few steps.

3. Now need to check the temporary results from 2. according to the stored annotation data. If local storage contains a evaluated rule set that equals the temporary evaluation on the current web element, ...

4. ... then it is a match.

Algorithm Performance The algorithm explained above is not performant. The reason is the obvious $O(n^3)$ complexity due to the triple encapsulated for-loop. Since most web views have a quite limited amount of non empty elements it is now drawback for the prototype testing. However it is easy to understand and to visualize. For reasons of completion lets take a look on a more performant idea: The basic problem is that we don't search for keys but values instead. In each step we evaluate the value for the current web element and compare it to the values of all stored annotations. To overcome this issue, it would be necessary to generate hash values over all values of an element. For instance an element

that is identified with three rules, gets a hash value generated from the composite of all rule results. Then this hash is used as the key of this annotation. This way we reduce the algorithm to a complexity of $O(n^2)$. The removed for-loop is the iterative search through stored annotations. This has become a hash search procedure that has the complexity of an average $O(1)$ and $O(n)$ in the worst case.

Figure 21: Some of the used technologies for the web service

4.3 Social Weaver - Web Service

The coming part is about the implementation of the web service that provides interfaces for our previously built plugin.

4.4 Used Technologies

Before we discuss our web service architecture, we first of all list the used technologies and give a brief explanation. Readers who are familiar with the following terms may skip this section. After pointing out the architecture, we map our defined requirements to our implementation and explain how those are achieved.

Model View Controller (MVC)

The model view controller is the probably most common used user interface architecture pattern. Martin Fowler even writes in [16] "I've lost count of the times I've seen something described as MVC which turned out to be nothing like it.". Since the reason, why we are applying MVC for our web service, is more a result of technology choosing, than an architectural one, there is need for explanation.

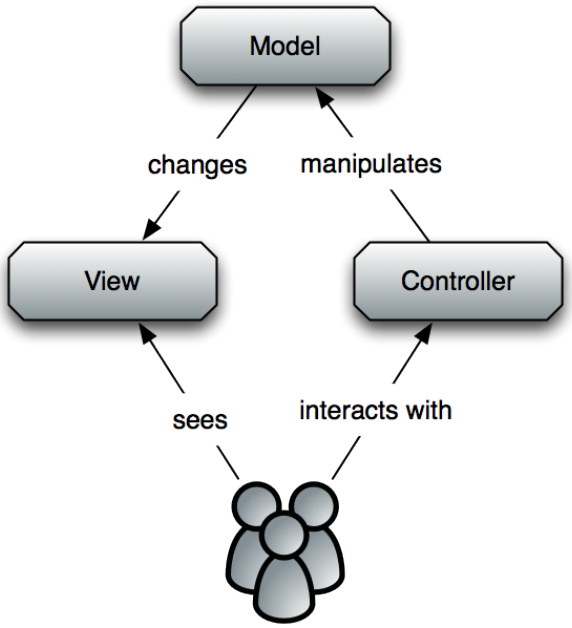

Figure 22: Basic structure of MVC

The basic idea of MVC is to separate displayed content and user interaction. As the name already reveals there are three modules that interact with each other and the user (also check Figure 22):

Model includes the logic of the system. That doesn't mean just business processes and functionality but even the persistence layer belongs to the model from the MVC perspective. The model updates the view and is being manipulated by the controller.

View is the representation of some data. A view doesn't have to be a whole view in a browser (like it is when we use *web view* outside this MVC context). It might represent a button, sub screen or any other user interface element. The view is seen by the user and being updated by the model.

Controller takes input from user actions and use these to trigger operations included by the model. The controller manipulates the model and is being used by the user.

User is able to see the content displayed by view and to use the controller parts to interact with the system. This is the essential part of MVC that seeing and interacting is separated.

Even though our web service has a proper MVC architecture we don't use it in the conventional way. Later we see that we use only RESTful requests to interact with the web service. It is easy and clean to extend a MVC pattern with additional interfaces for RESTful interactions. And with certain technologies, like Spring Roo, it is faster to generate a classic web service.

OpenJPA

OpenJPA is a object-relational mapping for the Java Persistence API (`javax.persistence`). The implementation makes it easier to interact with the persistence layer. Defining `EntityManager` for entities with some rules like

```
1  public interface EntityManager {
2
3      public void persist(Object entity);
4      public <T> T merge(T entity);
5      public void remove(Object entity);
6      public <T> T find
7      (Class<T> entityClass,
8        Object primaryKey);
9      public <T> T find
10     (Class<T> entityClass,
11       Object primaryKey,
12                   Map<String, Object> properties);
13     public <T> T find
14     (Class<T> entityClass,
15       Object primaryKey,
16                         LockModeType lockMode);
17   ...
18 }
```

enables us to user persisted data nearly like objects. In the web service any operation that writes to database is using OpenJPA.

PostgreSQL

PostgreSQL is a cross-platform open source object-relational database system that the web service uses for the persistence layer. Since our database usage doesn't exceed basic operations there is no need to explain extended features of PostgreSQL.

RESTful Web Services

RESTful web service or often called RESTful web API as well use HTTP and REST principles. REST was defined the first time in Roy Fieldings doctoral dissertation [14]. It stands for *representational state transfer*. REST uses the same types of operations as, for example SOAP. These are:

1. GET
2. POST
3. PUT
4. PATCH
5. DELETE

The central principle is that every accessable resource has its own and unique identifier (URI) in HTTP context. Related to this URI the different operations, mentioned above, can be executed. The RESTful provider handles each operation accordingly. We have two different cases where these operations slightly differ. It depends whether the URI belongs to a single resource entity (like for instance: `http://localhost:8080/anchors/54`) or a set of entities (like `http://localhost:8080/anchors`). Performing requests on a set URI operates logically on sets. This means GET retrieves the whole list, POST adds a list of entities, DELETE deletes everything and so on. On the contrary requests on single resources only apply to the single element.

Spring

Spring is an application development framework for Java. Since its first release in 2002 it grew rapidly under the Apache 2.0 license [29] into several directions like modern web, data access, integration, mobile, social, security and cloud ready. All those sub frameworks provide different functionality. Besides that there exists several side-projects. The web service makes use of one of those side-project that is called Spring Roo[30]. Roo is a development tool that makes it easy to create basic structures for web services with database connectivity. This tool makes extensive use of some Spring framework modules like Spring Integration, Spring Framework, Spring Security and Spring Web Flow.

AspectJ

AspectJ is an extension to the Java programming language to provide aspect oriented programming. Even though AspectJ is the probably best known representative for aspect orientation, the basic idea of this programming paradigm is applicable to any object oriented language. Basically an aspect enables us to break down program logic to another layer. In object oriented environment most of the time the logic, that is related to an object, is included in the same class. With aspects it becomes possible to distinguish different behaviors, related to an object, into different classes or even reuse them. For instance we have an entity controller for the anchor, the *AnchorController*.This object is responsible for handling requests that arrive at the web service. Since there are several request types and different possibilities to answer them, we can distinguish between JSON and HTML requests. JSON requests can be sent by any client, whereas HTML is requested from the web client. We split those cases up into two separate aspect classes. By default these are called *AnchorController_Roo_Controller* and *AnchorController_Roo_Json*. Using aspects becomes more valuable with increasing complexity. The *Anchor* entity is split into aspects for bean-, JPA entity-, JSON-functionality and

[29] `http://www.apache.org/licenses/`

[30] `http://www.springsource.org/spring-roo`

so on.

4.5 Web Service Architecture

The web service is a common MVC architecture that uses JSON/RESTful interfaces. The persistence layer is connected to a PostgreSQL database using OpenJPA.

Our main entities are the anchor and social element. The anchor has been already discussed from the concept point of view. The entity contains all the parameters that are necessary for matching web elements. This set of parameters can differ from one web application to another. Additionally the anchor entity contains an unique object identifier (OID) that defines the anchor even across different user sessions. This way updates can be performed more easily without having to check for all parameters every time. Which would be hard especially in those cases where an unusual set of matching parameters is being used. To avoid this difficulty but to still keep all parameters related we generate a hash from a combination of all relevant parameters. This hash is used as OID. Besides that the Anchor holds a time stamp with information about the last modified date. We use this data for the synchronization procedure.

The social element entity is handled as a separate entity but from the concept perspective it is a part of the Anchor. Basically it works as a container for any kind of social content. Because our prototype just provides a simple HTML inject, the social element contains an URL and a reference to the Anchor. But it is extendable to hold data for native and more complex social element types.

The entities are implemented as beans and therefore being directly persisted in the PostgreSQL database.

According to the model view controller pattern there are also controllers for the entities that provides several interfaces that are accessible through the standard REST requests.

The View from our Model View Controller architecture is a web view that allows the user to check the content manually (see Figure 23). This is just a pleasant side feature and not related to our Social Weaving use cases and for that reason this part should be discussed no further.

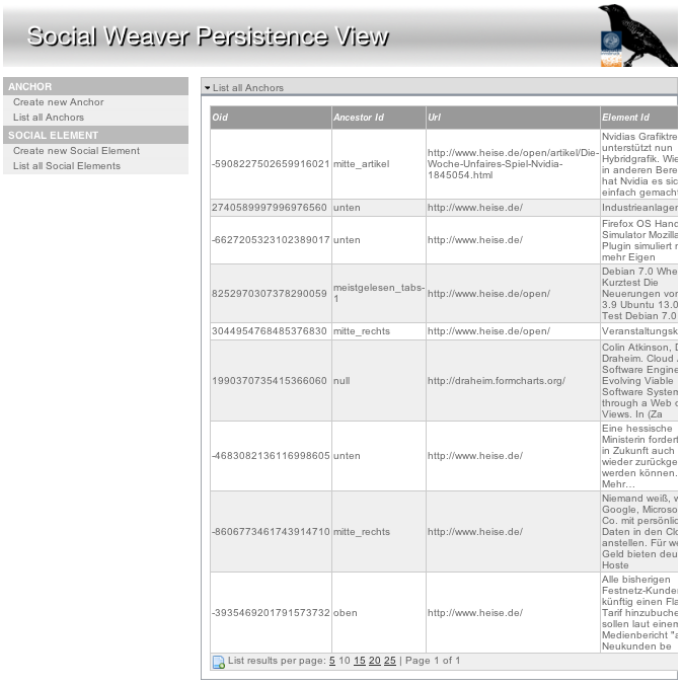

Figure 23: Screen shot from Social Weaver Persistence Web View

4.6 Requirements for the Web Service

In Section 3.5.2 we defined the following requirements for the web service:

1. Offer service that receives messages from plugin-clients
2. Synchronization for requests from different user-session
3. Persist updates into a database
4. Keep the server application independent to weaved-into web application
5. Keep the server application independent from the plugin
6. Parse incoming messages
7. Create outgoing messages

In the following we explain requirement by requirement how the prototype web service implements the features. The introduction of 4.5 included a brief overview of the web service. Technical details only are shown and discussed when relevant to the fulfillment of the corresponding requirement.

4.6.1 Offer service that receives messages from plugin-clients

For the needs of the prototype we use a standard RESTful PUT interface on the server side for the anchor entity:

```
1 {
2     @RequestMapping(method = RequestMethod.PUT, headers = "
          Accept=application/json")
3     public ResponseEntity<String> AnchorController.
          updateFromJson(@RequestBody String json) {
4         HttpHeaders headers = new HttpHeaders();
5         headers.add("Content-Type", "application/json");
6         Anchor anchor = Anchor.fromJsonToAnchor(json);
7         if (anchor.merge() == null) {
```

```
8               return new ResponseEntity<String>(headers,
                    HttpStatus.NOT_FOUND);
9           }
10          return new ResponseEntity<String>(headers, HttpStatus
                .OK);
11      }
```

With no security restrictions any client can send a request and since the anchor format is valid, the web service persists the data into the database. The request body has to have JSON format. The JSON body is deserialized (line five) using the default JSONDeserializer provided by the Flexjson framework[31]. The fact that we use a PUT request enables us to either update or create new entities. Messages that has the same payload, create the same OID hash. Every anchor on the same element is recognized as an update. New anchors insert a new entity.

A possible and valid PUT request using cURL[32] could look like:

```
curl -i -H "Accept: application/json" -X PUT -d
"url='http://draheim.formcharts.org/',
payload='[
{"doc_location":"http://draheim.formcharts.org/"},
{"element_content":"PD Dr. Dirk Draheim"}
]'"
http://localhost:8080/anchors
```

This way of receiving requests is very simple and provides no handling for user sessions, secure authentication or validation of anchors.

4.6.2 Synchronization for requests from different user-session

The prototype doesn't provide specific user sessions. This would be desirable as an extended feature. The web service handles a synchronization between different users exactly as for one user. Any message with an anchor that is received, injects into the data set of anchors. We use a time

[31] http://flexjson.sourceforge.net/
[32] http://curl.haxx.se/

stamp to signalize that updates were made. When users synchronize with the server the simply receive all anchors that are newer than the newest anchor located in the users plugin.

Lets clarify this on a small use case scenario with Alice and Bob:

1. Alice starts the plugin and synchronizes at time n. It is a new session so no anchors are sent back to Alice. Both parties, Alice and the web service, have n set as update time stamp.

2. Alice adds two anchors to the data set. These anchors are transmitted to the server and get the time stamp n+1.

3. Alice automatically synchronizes with the server after sending the update. Again both parties have a synchronized time stamp (n+1).

4. Bob joins this session and start his plugin. This initiates a synchronization and Bob gets all updates until n+1.

5. Bob sends a new anchor to the server.

6. The web services receives the anchor and sets its time stamp to n+2.

7. Bob automatically synchronizes his local data content. Bob and the web service are now at time stamp n+2.

8. Alice has to refresh the plugin since the server is silent. Until Alice or the plugin initiate a synchronization, Alice keeps time stamp n+1

Besides the drawbacks that users cannot determine who made what changes (and when), the web service has the issue that no push notifications are supported. The plugin is regularly update automatically or triggered by specific actions. Such actions can be update requests or opening new web views.

This minimalistic user synchronization support is sufficient for showing how Social Weaving works across different hosts though.

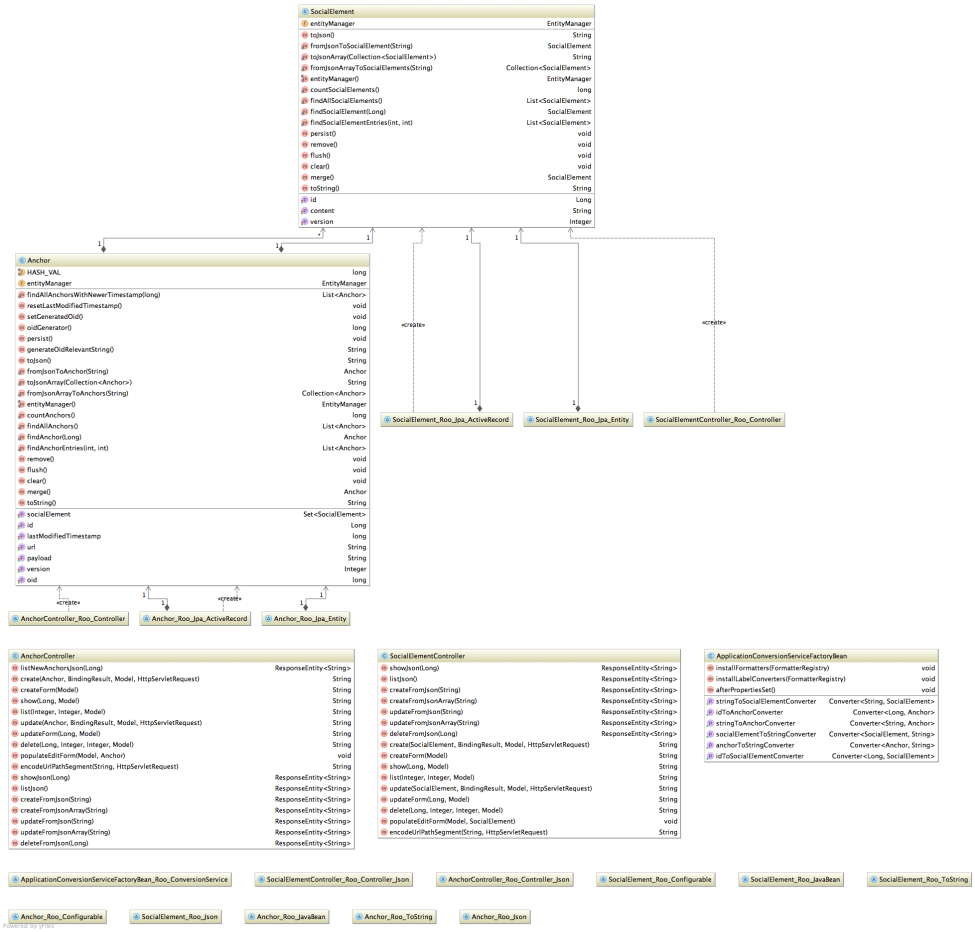

Figure 24: UML package diagram from web service

4.6.3 Persist updates into a database

Once the REST service receives and deserializes the JSON file, we proceed with populating the new data into the database. Since we use PUT functionality we need to merge it eventually into already existing database entries. This is realized using the following implementation:

```
1 {
2     @Transactional
3     public Anchor Anchor.merge() {
4         if (this.entityManager == null) this.entityManager =
              entityManager();
5         Anchor merged = this.entityManager.merge(this);
6         this.entityManager.flush();
7         return merged;
8     }
9 }
```

As entity manager we use the standard from the `javax.persistence` library[33]. The entity manager is configured with OpenJPA and connected to a PostgreSQL database.

4.6.4 Keep the server application independent to weaved-into web application

In the decoupling requirement it is about the content of the annotation messages that are transmitted between plugin and web service. The requirement should ensure that the message form is independent from the target web view. The target web view is the web page or application the annotation belongs to. The web service needs to be able to synchronize all annotations simultaneously no matter where they are annotated. For instance one annotation made on `http://www.uibk.ac.at/` has the same form and is handled equally as an annotation from `internal-sap.examplecoorp:4567`.

To affirm that this constraint is not disturbed, messages are handled as information containers. Once they arrive at the web service, it pro-

[33] `http://docs.oracle.com/javaee/6/api/javax/persistence/package-summary.html`

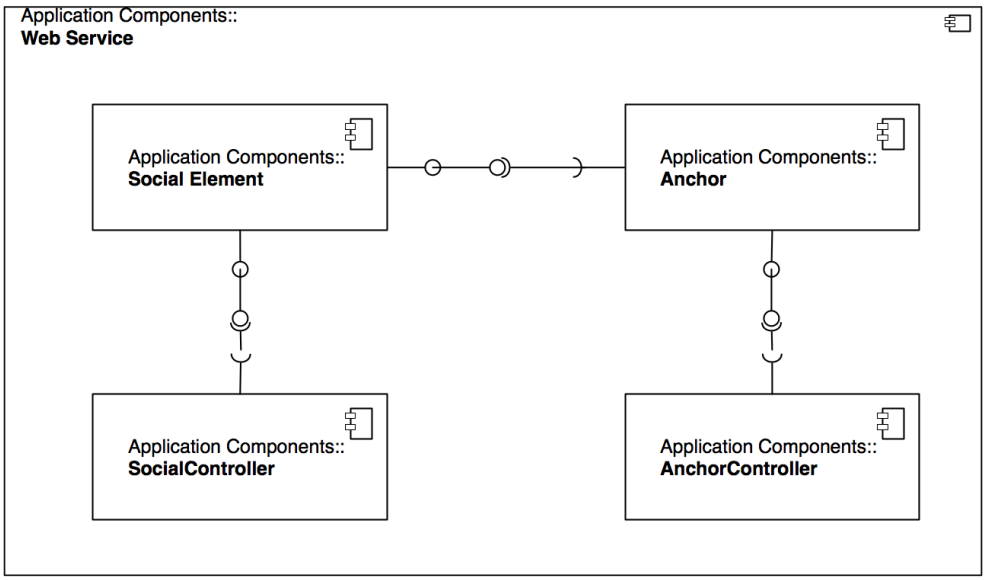

Figure 25: Component diagram for the web service

vides those without any knowledge about the inside of the message. All operations on the data is read-only.

4.6.5 Keep the server application independent from the plugin

The previous decoupling requirement 4.6.4 was about the content of messages. This requirement is about the message format and the processing rules. Since the Social Weaver prototype provides a plugin for Firefox only, this requirement is more a heads-up for potential upcoming development of plugins for other browsers. The exact format of the messages has been discussed already in the context of plugin requirements (4.2.5 and 4.2.6) and is issue in the coming Sections (4.6.6 and 4.6.7). As long this format is kept up, the requirement fulfillment remains valid.

4.6.6 Parse incoming messages

Formerly message parsing was discussed in Section 4.2.6. Hence this section doesn't cover the way of creating the message. We just assume

the defined format, which was:

```
{"annotation":[
   {"url": "http://www.uibk.ac.at/"},
   {"payload":[
      {"doc_location":"http://www.uibk.ac.at/"},
      {"element_content":"Campus maps"}
  ]},
  {"social_element":"http://www.uibk.ac.at/internalwiki/faqs"
     }
 ]}
```

In Section **??** we stated that the web service doesn't modify the content of the message. The only operations are read only. The following steps take place when a new annotation array arrives:

1. The array is split and the annotations handled separately
2. Each annotation is split into its elements: `url`, `payload` and `social_element`
3. `url` and `payload` are stored as values in an anchor entity
4. `social_element` is stored as value in a social element entity
5. Each time an anchor is persisted it updates its time stamp to the most recent update time
6. The primary key for an anchor becomes a has value over the payload and url

It is possible that an incoming annotation already exists and therefore is just an update. This case only differs slightly from the procedure mentioned above. The `social_element` is overwritten with the new content. The anchor entity receives an up to date time stamp. But the remaining fields remain untouched.

4.6.7 Create outgoing messages

Generating outgoing messages in this context means providing them as JSON representation through a RESTful interface. There are several interface for retrieving annotations from the web service like:

```
1 @RequestMapping(headers = "Accept=application/json")
2     @ResponseBody
3     public ResponseEntity<String> AnchorController.listJson()
          {
4         HttpHeaders headers = new HttpHeaders();
5         headers.add("Content-Type", "application/json;
              charset=utf-8");
6         List<Anchor> result = Anchor.findAllAnchors();
7         return new ResponseEntity<String>(Anchor.toJsonArray(
              result), headers, HttpStatus.OK);
8     }
```

It is possible as well to retrieve specific annotations by its value or key. Most of the time we need the special query for retrieving only the newest anchors. This task is accomplished by:

```
1 @RequestMapping(value="/updates/{lastModifiedTimestamp}")
2     public ResponseEntity<String> listNewAnchorsJson(
          @PathVariable Long lastModifiedTimestamp) {
3         HttpHeaders headers = new HttpHeaders();
4         headers.add("Content-Type", "application/json;
              charset=utf-8");
5         List<Anchor> result = Anchor.
              findAllAnchorsWithNewerTimestamp(
              lastModifiedTimestamp);
6         return new ResponseEntity<String>(Anchor.toJsonArray(
              result), headers, HttpStatus.OK);
7     }
```

Since these methods are formed according to the spring standard, the only part that requires explanation is the actual `finder` method in line five. Behind the scenes we use a `TypedQuery` to create a modified SQL query that returns all entities that has a newer time stamp than the parameter, that has been passed from the plugin.

```
1 TypedQuery<Anchor> query = entityManager().createQuery("
    SELECT o FROM Anchor o WHERE o.lastModifiedTimestamp > :ts
```

```
", Anchor.class);
```

4.7 Social Weaver - Script Support

The following part explains how the script support looks like in detail for our Firefox plugin. In Section 3.5.3 we discussed the purpose of the script support and what its goals are. The coming part applies this knowledge practically to the prototype development. Before we start stepping through the gathered requirements from 3.5.3, a brief example of a script use case is explained to provide a better overview to the reader.

A script contains the information about how an element in a web view might be found. We use JSON as format for the scripts because of the support that JavaScript provides for that standard. The script defines a set of rules that are used to perform operations on the selected element. The results from these operations are generated to a payload. This payload might be the information for anchors, URL and content related to social elements. The server handles this payload as one value and doesn't parse it. All meta information that are needed by the server are transmitted separately from it. To understand how the script is used to generate an anchor format take a look at the example:

```
1  {
2      rules:
3      [
4
5           {"doc_location" : "document.location.toString()"},
6           {"element_content" : "$(matchedElement).text()"},
7           {"element_children" : "$(matchedElement).children().
                text()"},
8           {"element_content": "matchedElement.innerHTML"},
9           {"dom_path": "true"}
10     ]
11 }
```

The purpose of this script example is to show the different possibilities and not a real scenario use case. It becomes obvious why this set of parameters would be no good choice.

Basically a script is a set of rules. A rule consists of a key name and the actual operation that is being used to perform an action. The key name can be any string withing quotes chosen by the script author. The

key names should be unique in one script though. The operation part offers different opportunities:

- jQuery Operation
  jQuery is a great possibility to traverse the DOM tree and it is possible to inject jQuery commands directly in the script. It is necessary that the command returns a string that is used for identification. The first line

```
1 {"doc_location" : "document.location.
      toString()"}
```

  would tell the plugin to save the document location - which is the plain URL in most cases.

  To trigger an operation related to the element that has been clicked by the user we might use the keyword `matchedElement`. In case a jQuery method is used it is still necessary to transform the matched element to jQuery format (`matchedElement` → `$(matchedElement)`).

- JavaScript Operation
  Even though most functionality related to DOM tree traversing should be covered with jQuery it might be of use sometimes to use JavaScript commands directly.

```
1 {"element_content": "matchedElement.
      innerHTML"}
```

  This line is an example for how to retrieve the HTML content of an element. This information might be used for matching elements for instance.

- Predefined Operation Some functions we need are not directly provided using JavaScript. The best example for such a case is the DOM tree path. We can use the DOM tree path for distinguishing similar elements. This functionality is provided by the plugin and can be enabled or disabled with the rule:

```
1 {"dom_path": "true"}
```

4.8 Requirements for the Script Support

In the conceptual Section 3.5.3 we defined a couple of abstract requirements that we need to specify for a proper implementation. Those requirements were:

1. Container of all necessary information for element matching
2. Decoupled from browser plugin and server backend
3. Syntax that is easy to read and write
4. Extension of the plugin with parsing methods
5. Default matching procedure should be provided (so the overall functionality is not limited when no scripts exist)
6. Scripts should be related to a single or a set of URLs

4.8.1 Container of all necessary information for element matching

For matching different elements we need the flexibility to use a variable set of rules. It would be a drawback to have a static rule system. The problem is that in some web views we need different operations to match elements than on others. Depending on the environment we need a specific container with the rule set.

We solve this issue by introducing the payload container that contains a JSON array with all information that is set by the script. This way we are able to extend information for element matching and modify it just by changing the script. The plugin and backbone of our system doesn't necessarily have to be changed because it always is handled as a single JSON string.

4.8.2 Decoupled from browser plugin and server backend

The script itself doesn't contain any information of the browser plugin type or the server module. The defined rules need to be supported by the plugin though.

A plugin that doesn't support jQuery commands would not work with our script example from above. Or the *predefined operation* (mentioned in 4.7) is another case that needs to be supported by the plugin. Using unknown or unsupported operations leads to unexpected results.

The server on the other hand is completely decoupled from the script support. The payload is just one column and the information about element anchors is of no use for the server. This is way the payload is not be parsed on the server.

Even corrupt payloads that cannot be used by the plugin are perfectly synchronized at the back end. This might seem not much of use at first appearance. But there is the case when a the payload for matching an element becomes defect not by using wrong rules, but because of some changes in the web view. Then we still want to keep the information about the element and just re-create the payload to it. Hence it makes sense to keep even those elements in the server database.

4.8.3 Syntax that is easy to read and write

It is a desirable goal for the future to have a system, like a script generator, that can be used be persons without any technical knowledge. But since we conduct prototype development we at least assume knowledge about web development. To choose the right rules and operations for element matching, requires this kind of knowledge anyway.

JSON is commonly used format and even possible to read by persons without technical knowledge. When providing a template with some example rules it is obvious to see how these rules might be extended. The most challenging part is to find and define the correct set of rules - but this is beyond the scope.

Beyond the criterion that scripts are human readable it would be easy to create a form or generator with a graphical interface to generate such scripts. Error handling could be supported already at this point. Even more desirable is an automatic script generator that generates scripts from user behavior without his active interaction (or at least minimal).

4.8.4 Extension of the plugin with parsing methods

When we were talking about rules and operations for element matching, this included short Java Script or jQuery commands. Technically we are not limited to short commands and it is possible to insert a whole command block as a rule. But for debugging reasons this is not a good solution. And some operations may require more than just a chain of jQuery commands.

For example lets take the procedure of determining the DOM path of an element in the tree. The idea is to start with the class name of the element itself. Then check the class name of its parents so long until the root of the DOM tree is reached. All names chained together describe the path of the element in the tree. The fact that this result is not necessarily unique underlies the problem that DOM trees are often ambiguous. We take a closer look on this issue in 4.9 Ambiguity Problem. But for some web views the DOM path still might be the right choice.

To understand why such a procedure is not possible to be fitted in a script we need to take a look at the code first:

```
1     var rightArrowParents = [];
2
3     $(this).parents().not('html').each(function() {
4         var entry = this.tagName.toLowerCase();
5
6         if (this.className) {
7             entry += "." + this.className.replace(/ /g, '.');
8         }
9
10        rightArrowParents.push(entry);
11    });
12    rightArrowParents.reverse();
13    var domPath = rightArrowParents.join(";");
```

In line three we initiate a loop for all parent elements of the selected element `this`. We watch out for every parent that has a `className` in line six. In a positive case we add this information to our path result.

The problem why we cannot insert such an operation into a script is, that we need an outside variable for keeping track of the results through

the loop iterations. Line three until eleven is the actual jQuery command. But it would be a mess to pass the whole code with the script. Theoretically it would be possible to support such scripts but it would make the script idea even more error prone.

Instead we use *predefined functions*. Those functions are implemented directly in the plugin and can be triggered using defined rules in the script.

To enable the DOM path parameter in the rule set we would just need to insert this rule:

```
1           {"dom_path": "true"}
```

The plugin would recognize this rule and run the code that determines the DOM path. In the Section 4.8.2 we already mentioned that this way of supporting more complex operations leads to minor coupling between plugin and script.

4.8.5 Default matching procedure should be provided (so the overall functionality is not limited when no scripts exist)

Even though it is not possible to provide a default matching algorithm that applies to any web view, it is desirable to at least have some default matching in case to specific script for the visited URL exists. If the user visits an new environment there is at least the chance that some elements might be matched.

To fulfill this requirement, the plugin provides a default script. When a new web view is loaded - all scripts are checked whether one of them applies to the new page. If it is not the case the default script is chosen and the user notified.

4.8.6 Scripts should be related to a single or a set of URLs

Every script, except for the default script from 4.8.5, mostly needs to be related to a specific environment. When we talk about environment in this case, it means views that inherit from a root URL. As example a script can be related to `http://www.gnu.org`. But any page that has this URL

as root, like for instance `http://www.gnu.org/philosophy/philosophy.html`, still applies to the same script.

The assumption is that views in the same environment apply to the same web architecture and hence its elements can be identified similarly.

Obviously this must not hold for every case. Theoretically a web view can has a totally different architecture than its root. But for prototype development this drawback is considered as tolerable.

4.9 Ambiguity Problem

Throughout the conceptual (3) and implementation (4) section about the prototype development we often noticed issues that make it tricky to achieve the proposed goals. The greatest challenge by far is the unique recognition of elements in a web view. More precisely: the finding of an element in the DOM tree.

This procedure consists of the two steps:

1. User chooses element from view
2. Plugin searches for this element from the DOM tree

The first step is easy because we have the clear information from the user input. The hard work is done by the user. Still our problematic situation takes place already at this point.

Hence the plugin needs to know in what way it is going to identify the element in the second step, it is necessary to gather all the information in first step. This is were the rules from a script (4.7) come in. Those rules are commands that are executed and retrieve results. Those results identify the element.

In the second step, when a view is opened, all appearing elements are checked with the script rules and the results compared to our results from step one. If the results appears to be the same - we assume it is the searched element.

The fact that we can only assume, finally leads us to the actual problem of this section. The DOM tree can be ambiguous and as a matter of fact this is no exception.

4.9.1 Ambiguous Grammar

We know the origin of ambiguity in computer science from formal grammars. A grammar is ambiguous for which there exists a string that has more than one leftmost derivation. Check Figure 26 for a common example of a ambiguous grammar.

```
1 S → SS | (S) | ϵ
```

S → SS | (S) | ε

Two leftmost derivations for: ()()()

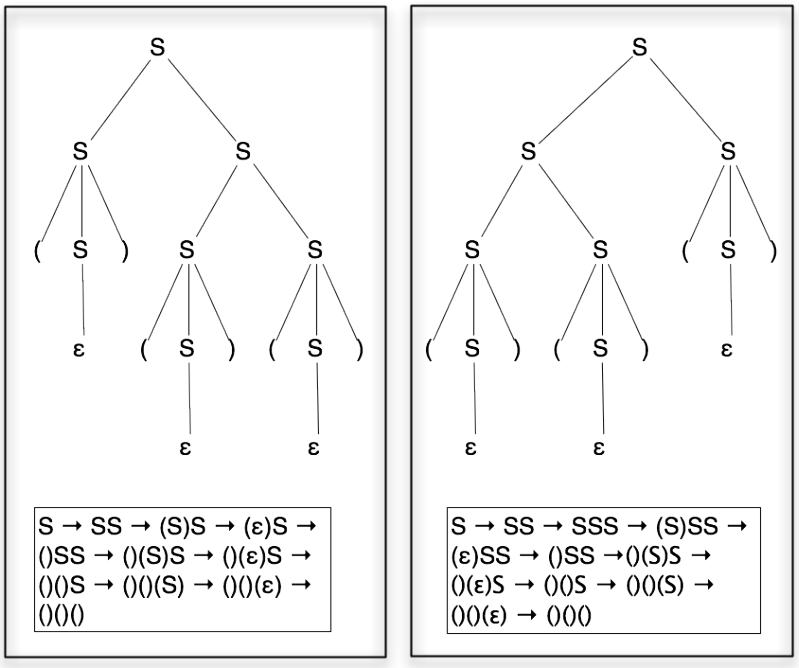

Figure 26: Example for an ambiguous grammar

```
1  <html>
2    <head>
3        <title>Software Development</title>
4    </head>
5    <body>
6      <b>Software Development Activities</b>
7      <ul>
8        <li> Requirements
9        <li> Construction
10       <li> Deploying
11     </ul>
12     <b>Software Development Methodologies</b>
13     <ul>
14       <li> Spiral
15       <li> V-Model
16       <li> Scrum
17     </ul>
18   </body>
19 </html>
```

Figure 27: HTML example for showing ambiguity

This generates a chain of correct opening and closing brackets. The grammar is ambiguous because we have to ways of generating the same string `()()()`. For the parse tree and step by step creation see again the Figure 26.

4.9.2 Ambiguity for Element Matching

What is the concern of ambiguous grammars when we talk about web architectures? We use the basic idea of ambiguity to describe the problem giving elements unique parameters. Imagine we would try to identify elements just on their location in the DOM tree. At a first glance this may appear as an effective way since we are used to storing data in tree based data structures.

We explain the problem with the HTML example in Figure 4.9.2.

Determining the DOM path for `Software Development` would return the value: `<html>,<head>,<title>`. In this case the path would be a unique identifier in this environment. The DOM path for `Construction`

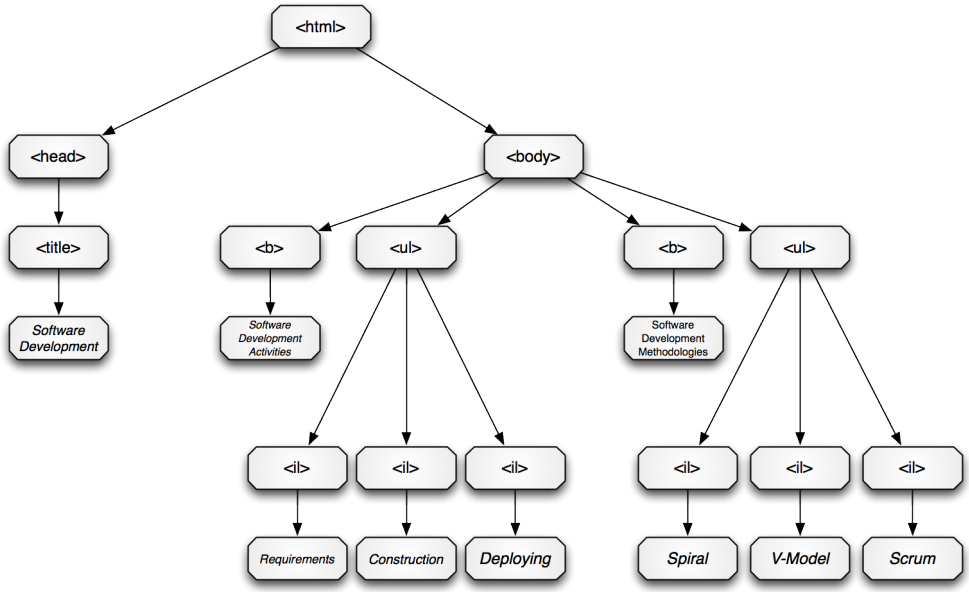

Figure 28: DOM tree representation for HTML code in Figure 4.9.2

would be `<html>,<body>,<ul>,<li>`. This path would not only apply to all `<li>` elements in the same `<ul>` context, but to all `<ul>` lists in the same subpath (which is `<html>,<body>` so far). Using this path would apply to all six list elements.

4.9.3 Parameter Data Object Model Tree

For the moment we forget that we can use other parameters (like the `<li>` content or surrounding parameters like the `<b>` blocks to identify elements. When we represent the HTML code as DOM tree, we receive a structure where it seems that every element can be located uniquely (see Figure 28). The parameter DOM path for `Construction`, which has the regular DOM path `<html>,<body>,<ul>,<li>`, would become `#0<html>,#0<body>,#0<ul>,#1<li>`. To keep that feature available in-code, we need to create a copy of the actual DOM tree and add parameters that uniquely identify the class types. The parameters are set by following the algorithm in Figure 29. Informally this algorithm starts at

1 Select the root as current node and mark it with #0

2 Retrieve class types of all children

3 Mark every class type set with an incremental counter starting with #0 to #n

4 Repeat Steps two and three for every child until no child has class type

Figure 29: Algorithm for parameter DOM tree

the root and marks it. From there on every children is classified for its class type. Children with the same class type are marked with an incremental counter. This operation is recursively be repeated until no child has a class type. Applying this algorithm to the DOM tree in Figure 28 we receive the parameter DOM tree in Figure 30. As you can see the class type `<b>` is marked with `#0` as well as `<ul>`. Even though both children have the same parent, there is no need to use the same counter for marking.

The advantage of using increment for same class types only is a better robustness. If the HTML code change it might happen that the parameter DOM tree becomes corrupt. With independent counters impacts from changes are attenuated. The idea of the parameter DOM tree is just mentioned for reasons of completeness. The algorithm is not implemented in the prototype because of practical reasons.

The parameter DOM path is more effective than the regular path, because it overcomes the ambiguity problem. But there still are problem that apply to both methods. Web applications and pages changes frequently and though the underlying HTML code. When we use the a tree representation for the whole code - basically any changes becomes critical and might affect the element matching.

For that reason the prototype focuses on the script support approach. It provides more flexibility and is more independent of changes that appear not near its location.

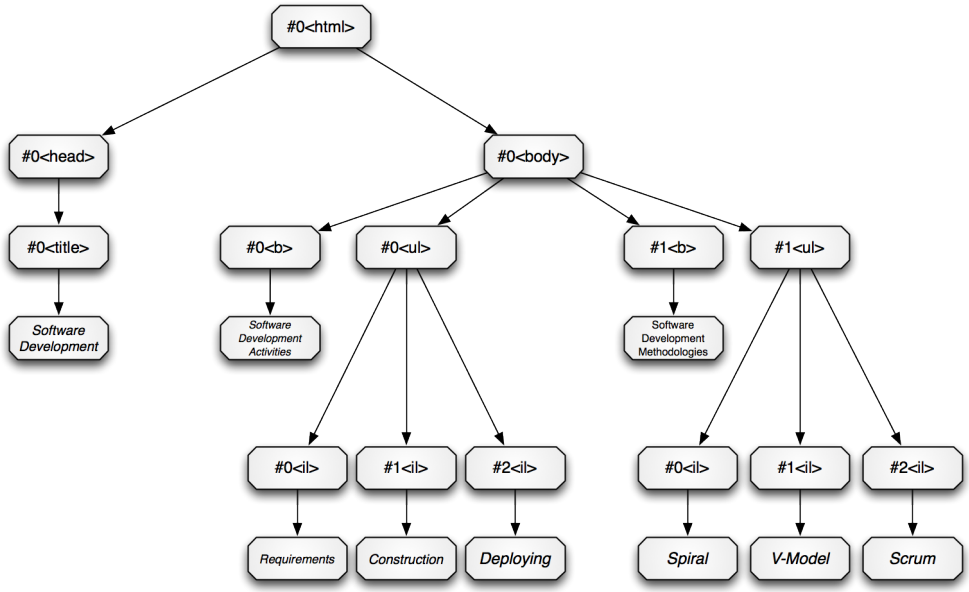

Figure 30: Example for parameter DOM tree

4.9.4 Effects of Web Evolution Element Matching

This section compares the script support and parameter DOM path approach in regard to web evolution. Web evolution means any changes that happen to the HTML code we use for element matching. We don't care whether those changes are updates made by user or that happen automatically. Changes are a risk for both approaches so that elements can not be matched anymore. But the script support is more robust to changes that appear somewhere not near the elements location. The parameter DOM path might affect on any element anchors. To show this, we use the HTML example from Figure 4.9.2. The element `Construction` should be matched with the parameter DOM path and a simple script. Then we change the structure of the HTML code and observe the effects on both procedures.

In the previous section we already determined the parameter DOM path for the element `Construction` which is `\#0<html>,\#0<body>,\#0<ul>,\#1<li>`. This parameter is enough

to clearly find the right element.

As script we use the text from the content of the element itself and the class type of its immediate parent. The get those information we use the following script:

```
{
    rules:
    [
        {"element_content" : "$(matchedElement).text()"},
        {"element_parent" : "$(matchedElement).parents().
            first()"}
    ]
}
```

Normally we would check on the documents URL to only apply the script to the right environment. But in this case we keep it simple to keep focus on the actual issue. If the script is executed on `Construction` we retrieve the following results:

```
{
        {"element_content" : "Construction"},
        {"element_parent" : "li"}
}
```

The update that happens to the HTML code is a newly inserted element into the *Software Development Activities* list. The new parameter tree is shown in Figure 31. Even though the matched element `Construction` is not directly changed, the updated environment results in a new parameter DOM path, which is: `\#0<html>,\#0<body>,\#0<ul>,\#2<li>`. The changes are slightly different, but still it crashes our matching procedures. `Construction` cannot be found anymore using the DOM path.

But the script still finds the element correctly since its independent from most of the environment. The results for the script execution we received earlier, doesn't change.

Keep in mind that there still are changes that can happen in the HTML code and that would affect scripts. There simply is no guarantee for a reliable anchor without exception.

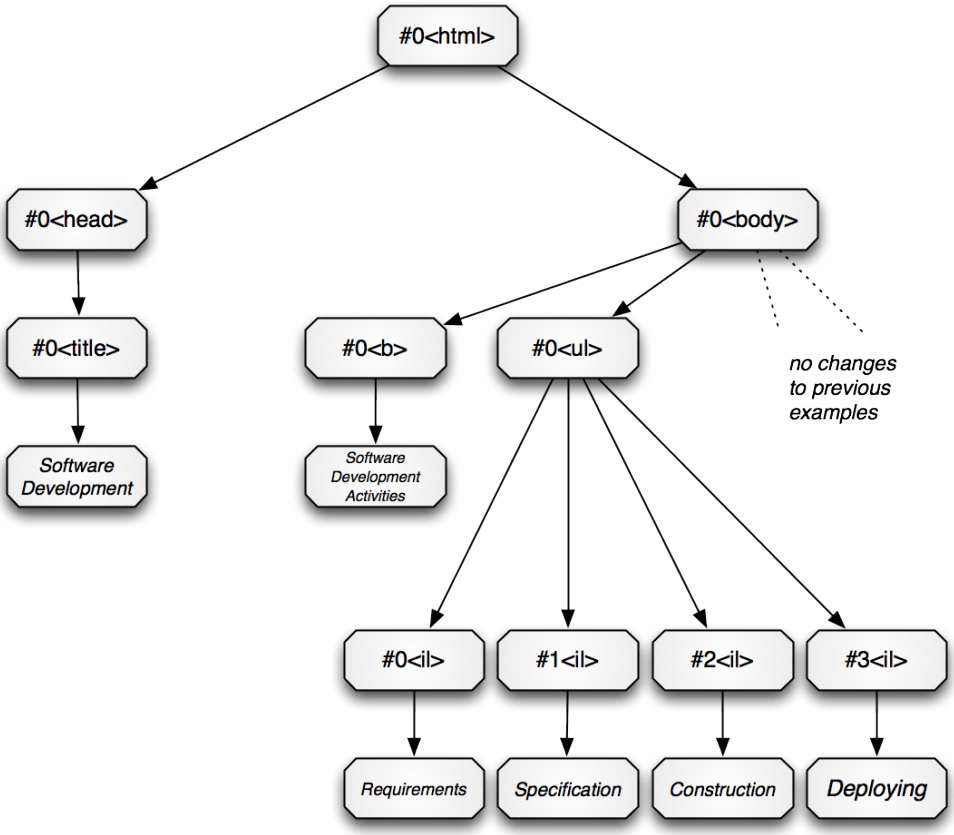

Figure 31: Modified parameter DOM tree

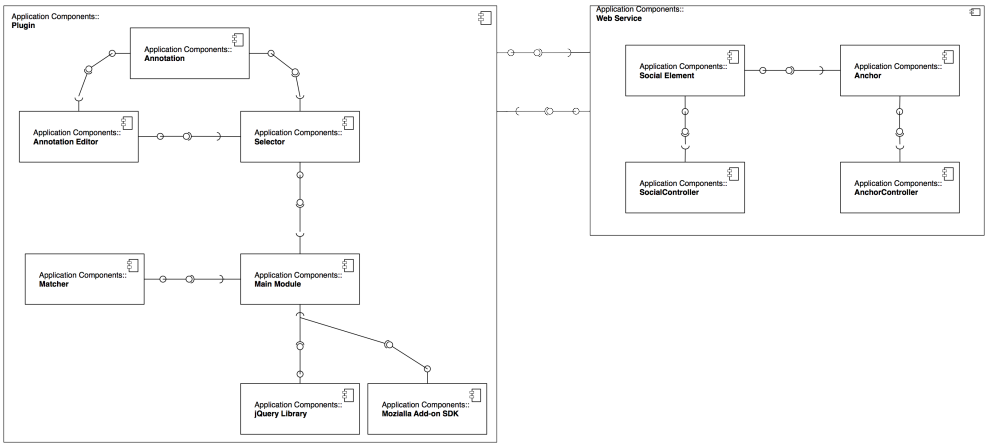

Figure 32: Component diagram for whole system

4.10 Dependencies accross Modules

5 Social Weaver Analysis

5.1 Social Weaver in Action

This chapter leads us through an real example where Social Weaver is being used. It is explained which components are used in what situations and how they interact with each other. Again we use Google Calendar as basis for the scenario.

Initial Scenario

The following explanations are based on a scenario with two users (Alice and Bob) who both are running the Social Weaver plugin in Firefox and are connected to the same Web Service (which means they share the same Social Weaver session).

Additionally they obviously need a shared Google calendar. For accessing the calendar they use the default web service provided by Google and no alternative client.

The scenario consists of the following steps:

1. Alice weaves a comment box to an appointment in the shared calendar

2. Alice uses the comment box to leaves a question

3. Bob logs in and anwers Alice's question

4. We manually destroy the anchor directly in the database

5. Alice recognizes this failure and relinks the comment box

In the following two sub-procedures, update and matching, are explained. The reason why we handle those separately is, that we use them more frequently in the whole process. That way we can just refer to them and keep the focus on the actual workflow.

Update Procedure

The synchronization for Social Weaver is quite simple. Basically the plugin sends at start up (or when asked) two parameters in a JSON array to the web service using a authenticated POST request. Those parameters are the current URL and the time stamp of the last update. If Alice starts up her first browser the first time, the plugin would send the following JSON file:

```
1 {
2   last_update=12341234,
3   current_url='www.cal.google.com'
4 }
```

The web service uses those information to determine whether there are new and relevant anchors or not. In the positive case (see Figure 33) the anchors are returned. In Alice's case nothing is returned since we have no marked elements.

What happens at the server with those data in detail? We use the time stamp of the last update and the current URL to create a query that receives only the corresponding anchors.

Through a simple HTTP header authentication we know which user is getting access to the anchor data. Even though it is not really relevant in our simple case. It would be more an issue when having multiple users in different session in one Social Weaver context. But such scenarios aren't covered in this thesis.

Matching Procedure

When we use the term matching procedure it means that existing anchors are visualized to the user. Before every matching procedure we assume that an update is triggered to ensure that the newest data is being used.

Beyond the update procedure there is no need communicate with the server. When the user opens a new URL it triggers the matching procedure. The plugin searches its local content whether there are some anchors for this URL. In a positive case (see Figure 34) the content is visualized to the browser view.

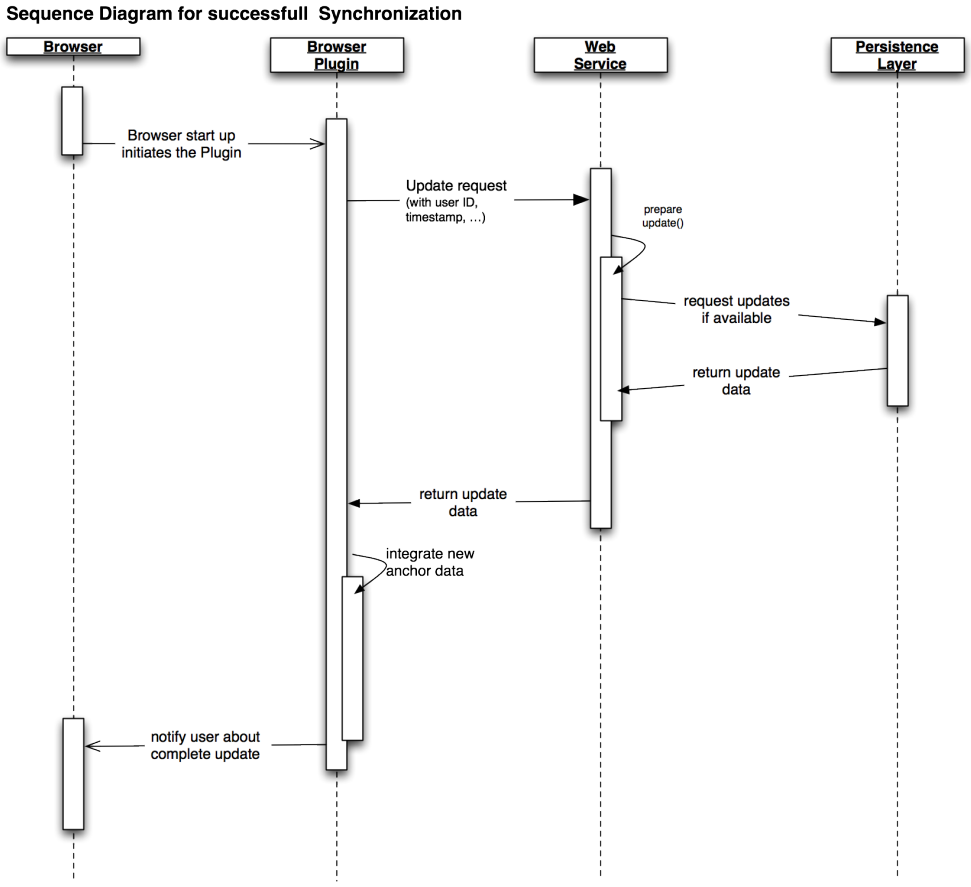

Figure 33: Sequence diagram for a successful plugin update

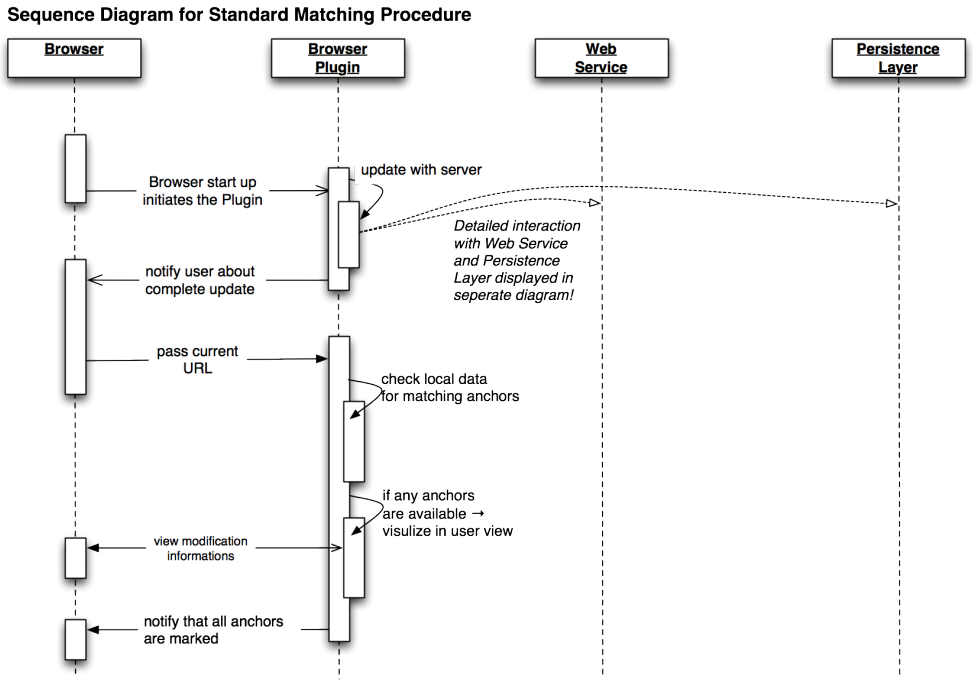

Figure 34: Sequence diagram for a standard matching procedure

At this point Alice would receive nothing from the plugin since no anchors exist for `www.cal.google.com`.

The way how anchors are retrieved from the plugin is quite similar how it is done at the back end. The major difference is that we don't use any time stamps at this time. There is no need for that since we assume everything is up to date after the update procedure.

Scenario Execution

Now that we learned about the two sub-procedures we are able to start with the actual scenario. First step is going to be that Alice weaves a comment box to an appointment:

Alice enters `www.cal.google.com` which first of all triggers an update. Afterwards potentially new anchors would be displayed in the browser - which is not the case right now. Now Alice is able to mark an appointment. By clicking an element she appends a comment box. In the

background the plugin runs the script (or scripts) that are related to the URL to define an identifier for the element. Using this identifier, the content-data for the comment box and the current URL the plugin creates a message in JSON format and passes it to the server.

The content for the comment box in this case is just a link. We use an external comment system that is injected as HTML code. Where or how exactly this comment box is defined, is not relevant to our matters.

For our example the JSON file might look like:

```
1 {
2   content-data='www.chatsystem.com/alice-apointment-17349',
3   element-id='72348082340880234480',
4   current-url='www.cal.google.com'
5 }
```

The message is passed as an authenticated POST Rest request. The web service performs some checks before the anchor is persisted. For instance it could be the case that there is already an anchor for exactly this element (because an other user created one in the meantime or the element identifier is not unique in this context).

In our scene everything works out fine and the web service persists the new anchor in the PostgreSQL database. The web service returns a positive status code to the plugin. This again triggers the previously discussed update and matching procedure. Alice sees her comment box attached to the appointment after it is guaranteed to be persisted on the server. It is not possible that the plugin creates locally new anchors that don't exist on the server.

Finally it is possible for Alice to use the comment box. This step is very simple. As we already mentioned the comment box is an external service that is only injected by a link into our system. Therefore Alice can add a comment without any consequences to our system at all.

Now it is Bobs turn. This process is quite similar and partially redundant to what happened when Alice created the anchor. For that reason we don't go into detail like we did for the first step.

Bob opens the Google Calendar website, which triggers the update and matching procedure for this URL. Since there is an existing anchor

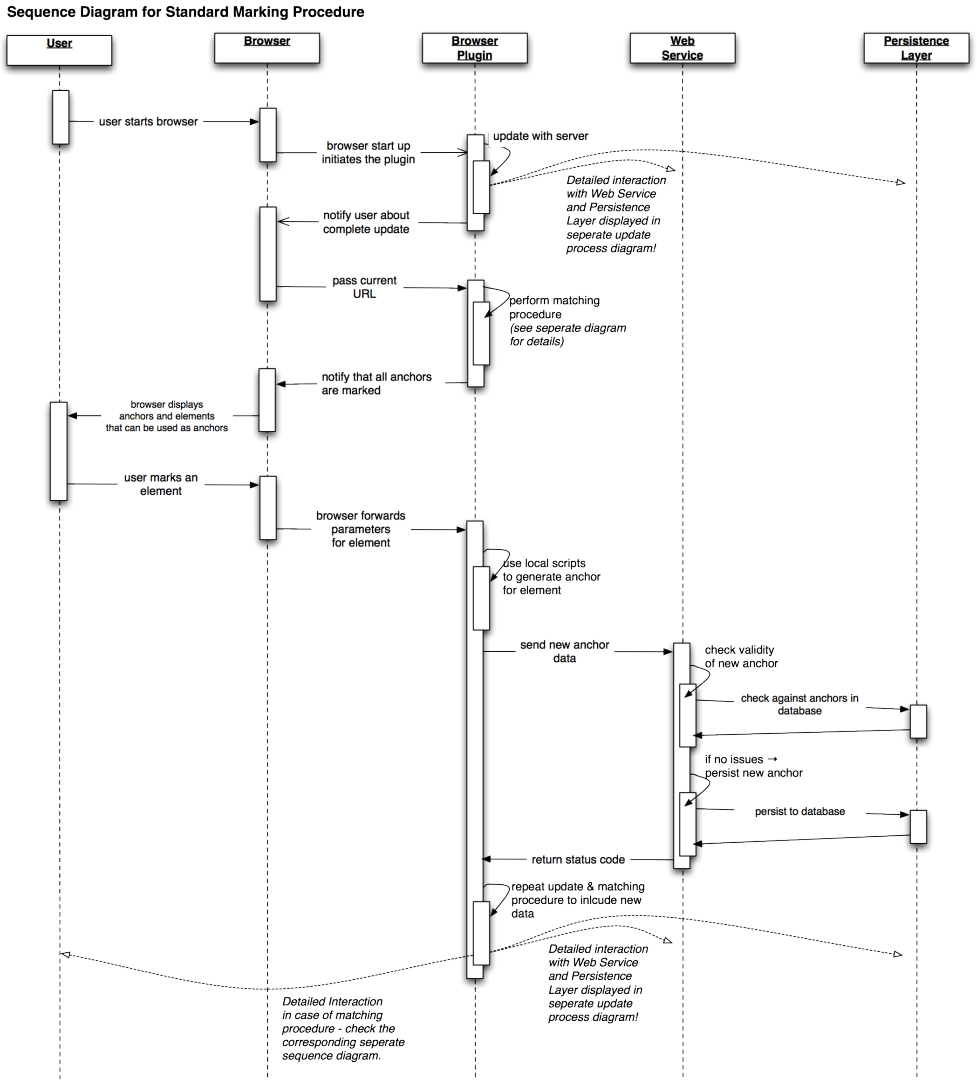

Figure 35: Sequence diagram for standard marking procedure

now - Bob's plugin receives the data for displaying the comment box entered by Alice.

The last two steps are getting more interesting again. We basically simulate a evolution of a website that destroys our anchor mechanism. That can happen very easily depending on the type of script we are using or how fast the webpage evolves, but this issue is discussed more deeply in the coming Section Referencessowe-assessment 5.3. What we do is to modify the element identifier directly in the database. This way it becomes impossible to match this anchor for the given URL.

So Alice visits her calendar to check whether Bob has answered her question. Again an update and matching procedure is started. The update works seamlessly, but an error occurs while the matching progresses. The plugin runs the script to determine the element that belongs to the element id - but with no success.

For that case the plugin enables Alice to re-link the content to the correct element or as in this case - appointment. The plugin performs the two following steps:

1. Create a new anchor element, that is basically a copy to the old one but with a correct element-id. This step is identical to the above described procedure when Alice weaved the comment box into an appointment the first time.

2. Additionally to the first step, we need to remove the broken anchor from the server. This is done by sending a remove Rest request to the server including the old element identifier.

After those steps are finished, it is necessary to run the update and matching procedures again. Now Alice and Bob are able once again to use the comment box.

Re-linking an anchor doesn't necessarily has to be due to an error or web page evolution. For instance if Bob changed the time of the appointment - the anchor would not work either. In this case a re-link would solve the problem as well.

5.2 Social Weaver Scripts in Action

... in work ...

5.3 Social Weaver Assessment

In the following we analyze how good Social Weaver works in several real scenario cases. Since it is developed as a proof-by-concept prototype, a general support for all web sites or web application was out of reach. Anyway the script support allows us to reach at least some flexibility. The testing range should cover static and more dynamic websites. Furthermore some freely available web applications are tested. We distinguish some criteria:

- Level of Marking Support
  This criterion is the ability of the plugin to recognize elements in a web view. This means first of all that all relevant elements should be recognized. The best case would that elements like advertisements or scrolling bars would be left out. Still all buttons, form elements and similar elements would be spotted. This criterion is not purely objective since relevant elements may differ for each user.

- Level of Matching Support
  Matching Support describes the ability of the plugin to find element that were previously marked. Even though this is at least as important as the Marking Support, there is no guarantee that matching is handled equally well as marking. For instance if we match an element only by its path in the DOM tree; This path might be ambigious to another element. In this case our social element would be weaved into the wrong place. We consider this as the worst case even worse as if no element could have been matched.

- Level of Anchor Reliability
  Anchor Reliability can be seen as part of the matching criterion. But with reliability we refer to the time relevant aspect. With the evolution of a web page, our anchor information might become obsolete. The chance for this to happen is increasing with time. News pages are the best example for a very fast evolution. An anchor attached to an article on the frontpage would not last more than a couple of days. But even on such a dynamic web page there mostly are ele-

ments that are more reliable (e.g. the search column or navigation bars).

This criterion should evaluate how probably it is that anchors outlast time.

- Expense
  Expense in this context means how much effort has been used to give support for the tested environment. The extent of the script itself and an appraisal how tricky the construction of the script is, whether just standard procedure has been used or if it was necessary to insert some hacks.

6 Conclusion

6.1 Results

6.1.1 Limitations

6.1.2 Market Potential

6.2 Summary

6.3 Future Work

Appendices

Use Cases This part will list the properly formulated use cases that can be derived from the gathered requirements.

A Akteure

- User
  Common user who uses the plugin to use Social Weaver.

- Plugin
  In context of the thesis the firefox plugin mentioned in section 4.1 Social Weaver - Firefox Plugin.

- Server Service

B Use Cases

UC. A. User can mark a web element for annotation

Use Case User can mark web element for annotation	
Use Case Description	User should be able to see what elements in the web view are annotable. In case his cursor moves above a annotatable element it should be visually marked.
Initiator	User who is performing an interface action.
Pre Condition	User needs to see what web elements are ready to be marked.
Process	Starting position is that the users sees some kind of web view with the plugin activated. Then the user enables the mode in which web elements are highlighted, that might be marked. From here it becomes possible to mark an element by simply clicking it. This brings the plugin into a new state where the type of an annotation can be specified.
After Condition	A successful marking is the precondition for an annotation action.
Miscellaneous	

UC. B. User can annotate a web element

Use Case User can annotate a web element	
Use Case Description	User should be able to annotate a specific web element so that we can use it as anchor in the future.
Initiator	User who is performing an interface action.
Pre Condition	UseC A. is the precondition for this Use Case.
Process	After the user marked an web element, the next step is to define the type of the annotation. What types are available depends on the state of the plugin or modifications. At this point we just assume an annotation were created and attached to the web element. From here the next condition is to make this annotation visible to the user who is author - or other users who just have a reader role.
After Condition	A successful annotation is the precondition for a visualization of an annotation object.
Miscellaneous	

UC. C. Plugin can display annotated elements

Use Case Plugin can display annotated elements	
Use Case Description	Already annotated web elements in a view should be recognized by the plugin and signals shown to the user where to find which annotations.
Initiator	Indirectly by a user who opens a view, which triggers the matching process of the plugin.
Pre Condition	Already existing annotated elements that might be displayed.
Process	At this stage we assume an annotation was created. It is not relevant whether the current user is the author of the annotation or a just random user. She opens a web view were an annotation is available. The web element that serves as anchor that is visually highlighted. Further interaction with it lead to an extended view that allows the current user to observe or interact with the annotation.
After Condition	
Miscellaneous	

UC. D. Plugin can send Annotations to Server

Use Case Plugin can send Annotations to Server	
Use Case Description	The plugin sending data about annotations to the server is one part that is necessary to provide synchronization between user sessions.
Initiator	Social Weaver Plugin Instance
Pre Condition	An annotation has been created
Process	After a user created a new annotation, this issue needs to be updated at the web service. It is necessary that the plugin is able to pack up the information about the way how the annotation is anchored to the web element and about the annotation itself. This package is to be transmitted to the web service where it is processed and stored.
After Condition	
Miscellaneous	

UC. E. Plugin can retrieve Annotations from Server Service

Use Case Plugin can retrieve Annotations from Server Service	
Use Case Description	Annotations created by other users or in previous sessions are stored at the server. This information needs to be synchronized with the plugin.
Initiator	Plugin and partially Web Service. This means the update might be initiated by the plugin by simply requesting it. On the other hand the update procedure itself is performed by the web service.
Pre Condition	Existing and synchronized annotations at the server.
Process	The process starts when a plugin requests an update from the web service. The origin for that can either be a default update at plugin start up or after an annotation is created and sent to the server. Either way, after the request is sent, the plugin keeps listening for the update messages. The messages needs to be in a format (analogous to the sending use case) that can be parsed to relevant information about annotation location and the type.
After Condition	
Miscellaneous	

UC. F. Server Service can retrieve Annotations from Plugin

Use Case âĂć	
Use Case Description	âĂć
Initiator	âĂć
Pre Condition	âĂć
Process	âĂć
After Condition	âĂć
Miscellaneous	âĂć

UC. G. Server Service can send Annotations Updates to Plugin

Use Case âĂć	
Use Case Description	âĂć
Initiator	âĂć
Pre Condition	âĂć
Process	âĂć
After Condition	âĂć
Miscellaneous	âĂć

C Defintions

Anchor is an information container that includes data to determine an element in a web view. It also includes data for the social web feature that will be weaved to the element.

Annotation see Anchor

Marks are the visual representation for an anchor. An element that has an anchor will show a mark in form of a rectangle, or colored background.

Social Weaving

Social Weaver

References

[1] G Antoniol, G Canfora, G Casazza, and A De Lucia. Web site reengineering using rmm. In *Proc. of the International Workshop on Web Site Evolution*, pages 9–16, 2000.

[2] Claus Brabrand, Robert Giegerich, and Anders Møller. *Analyzing ambiguity of context-free grammars*. Springer, 2007.

[3] Adrian Colyer, Andy Clement, George Harley, and Matthew Webster. *Eclipse aspectj: aspect-oriented programming with aspectj and the eclipse aspectj development tools*. Addison-Wesley Professional, 2004.

[4] IEEE Computer Society. Software Engineering Standards Committee and IEEE-SA Standards Board. Ieee recommended practice for software requirements specifications. Institute of Electrical and Electronics Engineers, 1998.

[5] World Wide Web Consortium et al. Document object model (dom), 2001.

[6] TH Corman, CE Leiserson, and RL Rivest. Algorithms, 1991.

[7] Dirk Draheim, Christof Lutteroth, and Gerald Weber. Generator code opaque recovery of form-oriented web site models. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 302–303. IEEE, 2004.

[8] Dirk Draheim, Christof Lutteroth, and Gerald Weber. *Source Code Independent Reverse Engineering of Dynamic Web Sites*. Freie Univ., Fachbereich Mathematik und Informatik, 2004.

[9] Dirk Draheim, Christof Lutteroth, and Gerald Weber. A source code independent reverse engineering tool for dynamic web sites. In *Software Maintenance and Reengineering, 2005. CSMR 2005. Ninth European Conference on*, pages 168–177. IEEE, 2005.

[10] Thomas Erl. *Service-Oriented Architecture A Field Guide to Integrating XML and Web Services*. Prentice Hall, 2004.

[11] Thomas Erl. *SOA Principles of Service Design*. Prentice Hall, 2008.

[12] Michael Felderer, Joanna Chimiak-Opoka, Philipp Zech, Cornelia Haisjackl, Frank Fiedler, and Ruth Breu. Model validation in a tool-based methodology for system testing of service-oriented systems. *International Journal On Advances in Software*, 4(1 and 2):129–143, 2011.

[13] Roy T Fielding and Richard N Taylor. Principled design of the modern web architecture. In *Proceedings of the 22nd international conference on Software engineering*, pages 407–416. ACM, 2000.

[14] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, 2000.

[15] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[16] Martin Fowler. Gui architectures, 2006.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns. 1995. *Reading, Massachusetts: Addison-W esley. ISBN 0-201-63361-2*.

[18] Ulric J Gelinas, Steve G Sutton, and Jane Fedorowicz. *Business processes and information technology*. Thomson/South-Western, 2004.

[19] Michael A Harrison. *Introduction to formal language theory*. Addison-Wesley Longman Publishing Co., Inc., 1978.

[20] Mario Jeckle, Chris Rupp, J§rgen Hans, Barbara Zengler, and Stefan Queins. *UML 2 Glasklar*. Hanser, 2004.

[21] Yingtao Jiang and Eleni Stroulia. E.: Towards reengineering web sites to web-services providers. In *Software Maintenance and Reengineering,*

2004. CSMR 2004. Proceedings. Eighth European Conference on, pages 24–26, 2004.

[22] Donald Ervin Knuth, Donald Ervin Knuth, and Donald Ervin Knuth. *Sorting and Searching*. Addison-Wesley, 2003.

[23] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, August 1988.

[24] Ramnivas Laddad. *Aspectj in action: enterprise AOP with spring applications*. Manning Publications Co., 2009.

[25] Avraham Leff and James T Rayfield. Web-application development using the model/view/controller design pattern. In *Enterprise Distributed Object Computing Conference, 2001. EDOC'01. Proceedings. Fifth IEEE International*, pages 118–127. IEEE, 2001.

[26] T. Lethbridge and R. Laganiere. *Object-oriented software engineering*. McGraw-Hill Higher Education, 2001.

[27] Peter Liggesmeyer. *Software-QualitŁt*. Spektrum, 2002.

[28] Joe Marini. *Document Object Model*. McGraw-Hill, Inc., 2002.

[29] James McGovern, Oliver Sims, Ashish Jain, and Mark Little. *Enterprise Service Oriented Architectures*. Springer, 2006.

[30] Ian Summerville. *Software Engineering*. Pearson Education Limited, 2001.

[31] A. Van Lamsweerde. Requirements engineering: from system goals to uml models to software specifications. 2009.

[32] Jean Vanderdonckt, Laurent Bouillon, and Nathalie Souchon. Flexible reverse engineering of web pages with vaquista. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 241–248. IEEE, 2001.

[33] S Webster. Object-oriented system development: D de champeaux, d lea and p faure addison-wesley (1993)£ 37.95 532 pp hardback isbn 0 201 56355 x. *Information and Software Technology*, 35(11):702, 1993.

[34] K.E. Wiegers. Software requirements. 2003.

[35] Pree Wolfgang. *Design patterns for object-oriented software development*. Reading, Mass.: Addison-Wesley, 1994.