Hasso Plattner Institute
University of Potsdam

Bachelor thesis

The design of an web-based event-driven client application architecture for Project Zoom

Bachelorarbeit

Entwicklung einer web-basierten Ereignis-gesteuerten Clientanwendungs-Architektur für Project Zoom

Norman Rzepka
norman.rzepka@student.hpi.uni-potsdam.de

Supervised by Prof. Dr. Holger Giese, Gregor Berg M.Sc. and Thomas Beyhl M.Sc.
System Analysis and Modeling Group

Potsdam, June 29, 2013

Acknowledgements

I would like to thank ...

Zusammenfassung

Die deutsche Zusammenfassung der Arbeit.

Abstract

The English abstract.

Contents

1. Motivation

This Bachelor Thesis documents a part of the Bachelor Project G1 2012 at the Hasso Plattner Institute in Potsdam. To support the documentation of innovative projects, a group of six students developed a software tool for the Hasso Plattner Institute School of Design Thinking (D-School) [1].

1.1. Design Thinking at Hasso Plattner Institute

The School of Design Thinking offers academic courses for students. Design Thinking is a method for creating new ideas and developing novel solutions [PMW09].

During the courses, students work on team projects. Starting in the Basic Track there are 1-week, 3-week and 6-week projects. In the Advanced Track students work 12 weeks in a row on one project. In addition to the student tracks the D-School also offers Executive Training where the projects usually do not exceed one week in length.

As one of its core principles the D-School actively encourages multidisciplinary teams. This mixture of different background leads to multiple viewpoints during the design phases and helps the team to filter obstacles early in the process.

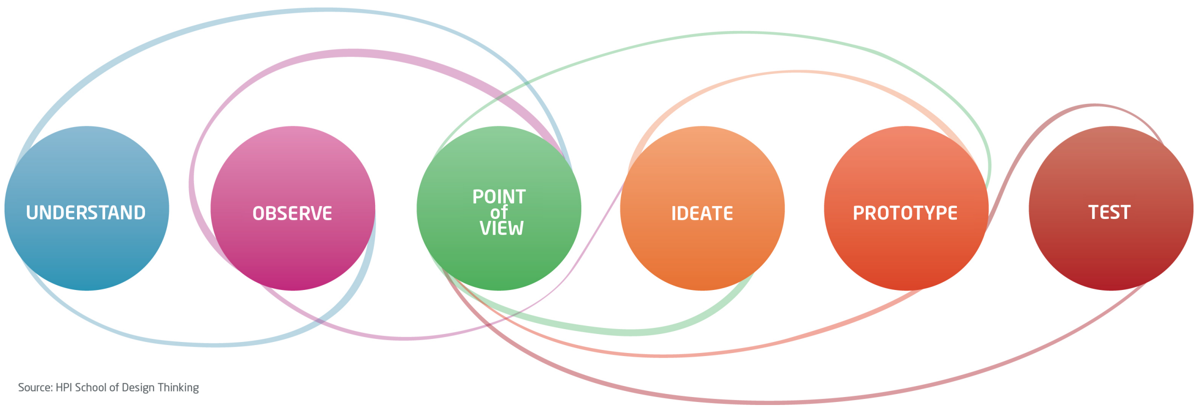
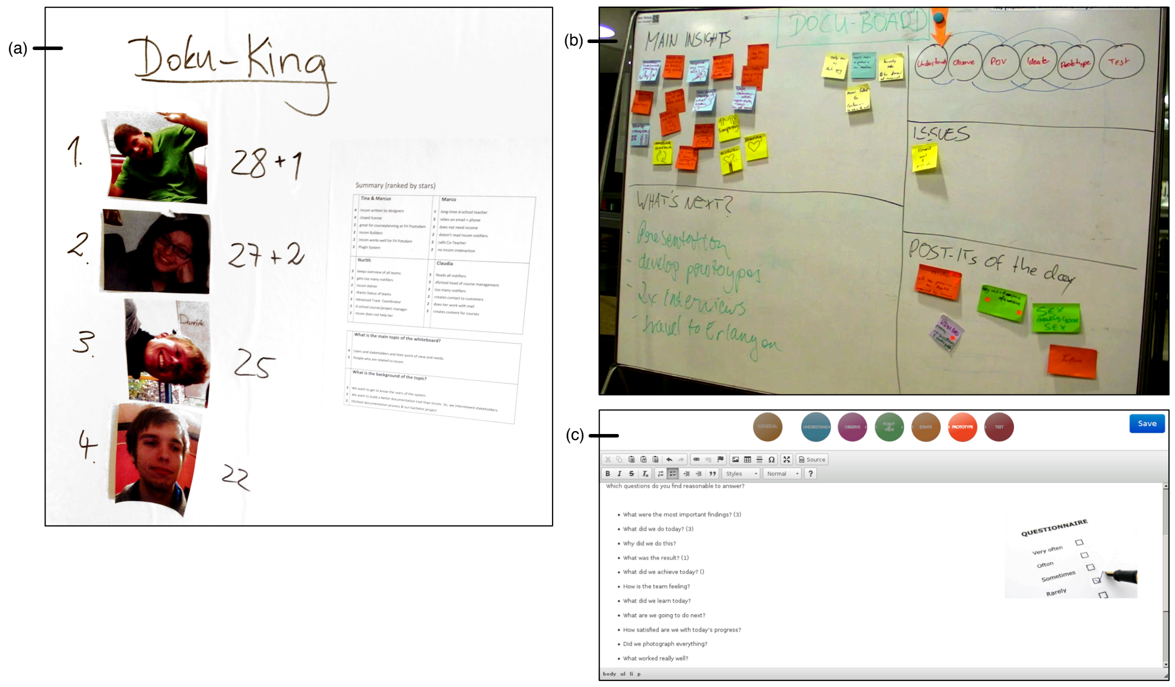


Figure 1.1. – An overview of the phases in the Design Thinking process. Source: [PMW09]

The projects usually deal with a problem posed by an industrial partner. The Design Thinking method includes an iterative process which consists of several well-defined phases, shown in figure 1.1. The phases guide the teams from basic understandings to the development of testable prototypes. The process is non-linear because the teams are encouraged to iterate. These cycles help to refine the

[1]`http://www.hpi.uni-potsdam.de/d_school/home.html?L=1`, accessed 06/16/13



Which questions do you find reasonable to answer?

- What were the most important findings? (3)
- What did we do today? (3)
- Why did we do this?
- What was the result? (1)
- What did we achieve today? ()
- How is the team feeling?
- What did we learn today?
- What are we going to do next?
- How satisfied are we with today's progress?
- Did we photograph everything?
- What worked really well?

Figure 1.2. – Prototypes created during the first design iteration: *(a)* Doku-King: a gamification[2] approach, where players earn points by documenting their work and validating the results of others. *(b)* Doku-Board: a preformatted white board, where students create daily summaries of the most important insights or results. *(c)* a rich-text-based editor, where students can summarize their results structured by the Design Thinking phases.

prototypes based on actual user feedback. Throughout the process teachers are advising the students.

Several staff members at the D-School are responsible for coordinating and acquiring the student's projects. Key stakeholders for this thesis are the Head of D-School, the Program Manager, the Knowledge Manager and the Track Managers.

1.2. Design process

The Bachelor Project also applied an iterative process, which is very similar to the one taught at the D-School. With this approach the project participants were able to benefit from the Design Thinking method while automatically learning about some of the needs of the D-School students.

During the project there have been several interviews with the relevant stakeholders. The group learned about the student's documentation efforts, especially commonly used methods and tools. Other stakeholders like teachers or D-School staff members have also been interviewed. Based on this information, the prototypes shown in figure 1.2 were developed to enhance the documenting experience of the students. These prototypes were evaluated during user testing sessions.

[2]Gamification is a trend that connects concepts of human-computer-interaction and game studies. [DDKN11]

In the second iteration a new prototype was designed which also addressed the need of the staff members to archive and categorize the projects for easy retrieval. This prototype is called “Project Zoom”. The following sections will highlight the most relevant use cases and requirements for this thesis.[3]

1.2.1. Use Cases

Project Zoom addresses two main use cases, which were distilled from the insights gathered in the observation phases.

U1: Student teams document their projects by organizing the digital documents they created, in a visual manner.
PREREQUISITE: The students have all documents they created available digitally.
POSTCONDITION: A visual knowledge graph is being created.

U2: D-School staff members get an overview of all projects. This overview enables access to the projects’ classifications, related people and documentations.
PREREQUISITE: The projects were entered in a database (e.g. FILEMAKER[4]) and the students have documented their projects.
POSTCONDITION: A visual representation of all projects is displayed.

Beyond these two major use cases there are other relevant use cases.

U3: The students add additional documents from their computers at home.
PREREQUISITE: The students have the documents available digitally. The computer needs to have an HTML5-capable web browser installed.
POSTCONDITION: The documents are included in the knowledge graph.

U4: Students, teachers and D-School staff members can retrieve versions of the knowledge graph at different points in time.
PREREQUISITE: The students have documented their projects using the proposed tool.
POSTCONDITION: A historical version of the knowledge graph is displayed.

U5: Students can access the documents they saved using different commonly-used storage providers, e.g. Box[5].
PREREQUISITE: The students stored documents using a supported service.
POSTCONDITION: Documents are offered for insertion in the knowledge graph.

U6: The Head of D-School and Program Managers show an overview of a filterable set of projects to potential industrial partners. *Optional*: A mobile tablet device is used for the presentation .
PREREQUISITE: The projects were entered in a database, e.g. FILEMAKER.
POSTCONDITION: A visual representation of the projects is displayed.

[3]The “Software Requirements Specification”[BBD+13] covers the use cases and requirements in detail.
[4]The D-School uses a FILEMAKER database to store projects and contacts. `http://www.filemaker.com/`, accessed 06/16/13
[5]The D-School uses Box as a shared storage service. `https://www.box.com/`, accessed 06/26/13

1.2.2. Requirements

To fulfill these use cases there are some technical requirements for designing and implementing Project Zoom. The following is a selection of relevant requirements.

R1: The system's interface has to support multiple platforms, including the popular desktop operating systems and modern tablet devices. (use cases U3, U4)

R2: The system's interface has to be accessible from outside the D-School. (use case U3)

R3: The system has to support concurrent users accessing and modifying contents. *Simplifying assumption*: A resource (e.g. a project's graph) can only be modified by one user at the same time[6]. However, multiple users may edit different resources and any user can read any resource.

R4: The system connects to different data sources, e.g. Box and FILEMAKER, and makes the stored data available through its GUI. (use cases U1, U5)

[6]Project teams at the D-School usually assign one member for documentation. Therefore, concurrent editing is not a high priority requirement.

2. Application Design

2.1. User interface

Project Zoom is designed to apply the concept of semantic zooming. Semantic Zoom is part of the computer interface model PAD proposed by Perlin and Fox [PF93]. This concept taps into the natural spatial thinking of users. Information is displayed on a large infinite two-dimensional canvas. Users browse around either by panning or zooming. There are different representations of the same pieces of information at different magnification levels. When zoomed out, documents are only represented by an icon or a title string. While zooming in, these abstract figures gradually resolve into the full representation of the respective documents.

In Project Zoom there are three main zoom levels: An overview of all projects (**Overview View**), a detailed view of a project and its metadata (**Details View**), and a view containing a project's knowledge graph (**Process View**). Users can navigate from one view to the next by zooming in and out. This approach allows building one cohesive system, which hosts interfaces that are targeted at the specific needs of different stakeholders (see use cases U1-U5).

There is a single zoom slider that is segmented into three parts for the main views to also support multiple zoom levels themselves. Especially the Process View takes advantage of the zooming capabilities and displays the documents in the graph in different levels of detail. The Process View is an interactive graph that allows users to add documents onto the canvas and connect them with edges. The document nodes can then be annotated using the commenting feature or by drawing clusters (i.e. free form shapes) around them. These actions are exposed through context-sensitive menus. The interactive graph also features automatic layout capabilities like collision prevention.

2.2. Architecture

In the design phase multiple architectures for Project Zoom have been evaluated.

A **monolithic application**[1] is a self-contained system that can run on a single computer. The benefits are that the data is consistent for all users. Additionally, the system is easy to set up because there is only one computer required for the system to work. This approach is well suited for applications that deal with independent datasets, which can be stored on one platform. Word processors are an example of monolithic applications. However, as use cases U3 and U4 as well as requirements

[1]Wikipedia, Monolithic application, `http://en.wikipedia.org/w/index.php?title=Monolithic_application&oldid=552899667`, accessed 06/17/13

R1 and R3 demand the data to be accessible on multiple platforms concurrently this architectural approach is not suitable for Project Zoom.

The **peer-to-peer**[2] architecture allows the distribution of data on multiple connected computers. Each node in the network is equally privileged and handles a subset of the data individually. Data consistency can be eventually achieved through replication from a neighbor node to another. This is a well-known database problem [GHOS96]. Access control can be integrated by using trusted computing technology [SZ05]. Peer-to-peer architectures are usually applied when a centralized controlling instance is to be avoided. Bitcoin[3] is a popular peer-to-peer application. The ability of a node to function, especially when joining the network, depends on the availability of neighbor nodes. Because of the low number of users the availability of nodes might pose as an issue in Project Zoom. Besides, implementing a peer-to-peer usually requires to build a native application which makes it harder to meet requirement R1.

The **client-server**[4] model is one of the most-used distributed architectures. One central computer acts as a server, which stores the dataset and manages access control. Client computers can access and modify the resources on the server through a network interface. An example of a client-server application is the World Wide Web. Fundamentally, the server is a single point of failure[5]. However, there are methods to maximizing the availability [GS91] [Col00]. This architecture is suitable for Project Zoom as it fulfills all the requirements, especially R1–R3, and enables all the use cases.

A web-based client-server architecture is a system that uses an HTTP-Server[6] and a browser application as client. Project Zoom incorporates this approach for several reasons:

- Using the client application of a web-based system only requires the installation of a web browser, which are wide-spread and usually preinstalled on the operating system. This makes it easy for students to modify the data from their computers at home (cf. use case U3).
- Developing native apps for mobile devices requires different technologies for each platform [CL11]. However, recent mobile devices are equipped with HTML5[7]-capable web browsers. Applications built using web technologies are likely to run on multiple mobile platforms (cf. requirement R1).
- Due to the standardization of the web technologies, emerging platforms will likely support them as well.

Alternatives to web-based client-server architectures are based on client applications that users have to install on their devices. Since the users of Project Zoom utilise multiple computing devices, this is not a favorable approach, as the application has to be installed on each device separately.

[2] Peer-to-peer architecture, [Sch01]
[3] Bitcoin, http://bitcoin.org/, accessed 06/25/13
[4] Client-server architecture, [Ber96]
[5] Wikipedia, Single point of failure, http://en.wikipedia.org/w/index.php?title=Single_point_of_failure&oldid=555725127, accessed 06/17/13
[6] Hypertext Transfer Protocol (HTTP), http://www.rfc.net/rfc2616.html, accessed 06/27/13
[7] Hypertext Markup Language (HTML) Version 5, http://www.w3.org/TR/html5/, accessed 06/27/13

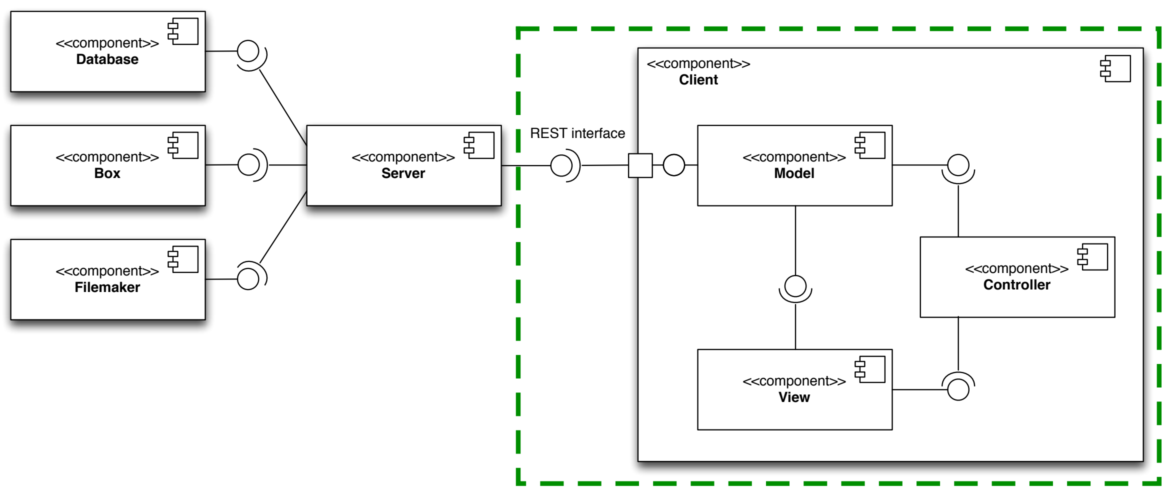


Figure 2.1. – Architecture overview of Project Zoom. The highlighted part is covered by this thesis.

Additionally, users may not even be able to install the application onto the device due to imposed access restrictions.

Figure 2.1 shows the system architecture of Project Zoom. The server implementation features an event-system that is fed by the data connectors and sets up the pipeline for thumbnail generation. The server also hosts the database model and handles user access control and authentication. Client and server are connected by a REST[8] interface. The client application is built using an MVC pattern, which will be detailed in later chapters.

2.3. Project context

There are six theses covering Project Zoom. Bocklisch [Boc13] describes the architecture of the server and the domain data model. Werkmeister [Wer13] details the connection of the data providers, e.g. Box and FILEMAKER, to the system. Bräunlein's thesis [Brä13] covers both the design and generation of document thumbnails for the Process View. Dieckhoff [Die13] details the automatic layouting capabilities of the interactive graphs. Herold [Her13] explains the context-sensitive actions that users can perform on the graph and its nodes and edges. This thesis is about the architecture of the client application, as highlighted in figure 2.1.

[8] Representational State Transfer, [Fie00]

3. Client Application Architecture

3.1. Overview

The client application of Project Zoom is a web-based application. It can be run using an HTML5-capable web browser. The client application accesses the server's resources through a REST interface. The code is partitioned into several modules. An architectural pattern similar to Model-View-Controller (MVC) is applied.

3.1.1. MV* pattern family

The Model-View-Controller (MVC) pattern was first introduced by Reenskaug in 1979 [Ree79] and later published by Krasner et al. in 1988 [KP+88]. It was one of the first approaches to separate business and presentation logic code within a software project.

A **Model** is an active representation of the data in the system. It usually encapsulates methods for fetching, persisting and transforming the data. It also emits events upon data changes that Controllers and Views can listen to. The Model itself is agnostic of any particular user interface. A **View** encapsulates all the code required to display a user interface with the data from the Model. A **Controller** is responsible for updating the Model when a user manipulates the View. [KP+88] [GHJV94]

Since its introduction in the 1980s the MVC has mutated to accommodate modern technologies. The architectural patterns MVC, Model-View-Presenter (MVP) and Model-View-ViewModel (MVVM) as shown in figure 3.1 a–c are commonly referred to as the MV* pattern family [Osm12]. As the separation of business and presentation logic MV* patterns are widely used in the development of web applications [Tak12].

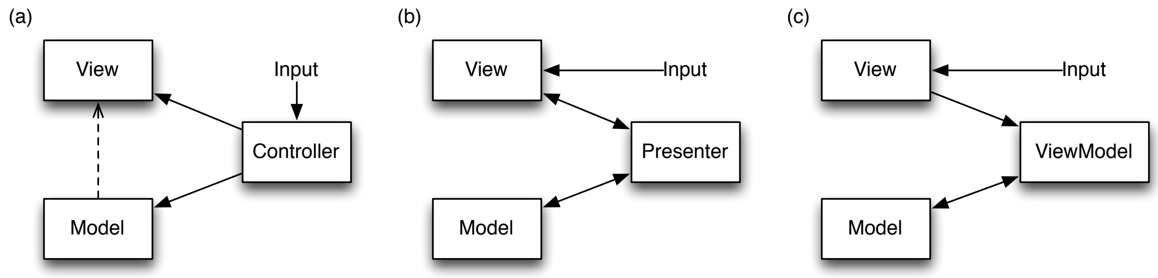


Figure 3.1. – *(a)* Model-View-Controller *(b)* Model-View-Presenter *(c)* Model-View-ViewModel

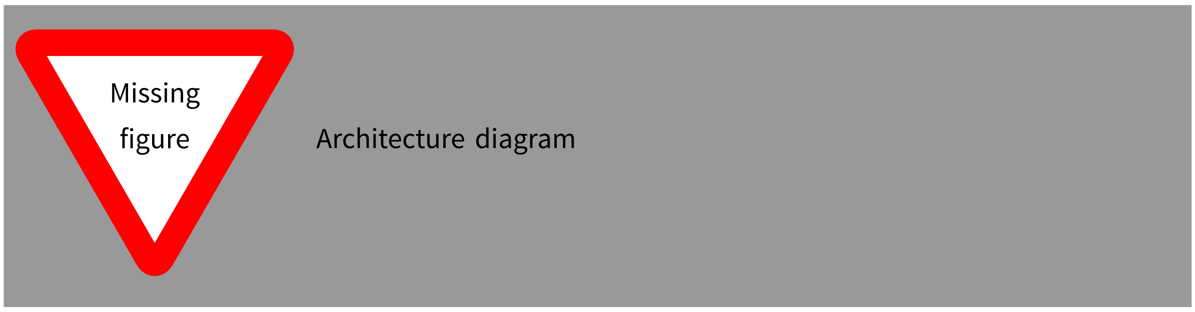


Figure 3.2. – Architecture diagram

Model-View-Presenter (MVP) is derived from MVC to achieve a complete separation between Model and View. Whereas in MVC the View depends on the Model, in MVP the View is agnostic to a specific Model and can be reused for different Models. The Presenter replaces the Controller and has the responsibility to connect the interfaces of Model and View.

Model-View-ViewModel (MVVM) is based on the concepts of MVP. However, the Presenter is replaced by a ViewModel, which contains a subset of the Model as well as additional state and methods. The ViewModel and the View communicate through data-binding[1] and events. Since Model and View are separated the ViewModel connects both and contains logic in state-change and event handlers.

MVP and MVVM are commonly used in scenarios in which it is important to have user interface components that are of general-purpose and have to be reused in several different locations. Such applications are usually enterprise or consumer applications with a large number of Views. MVC is a more lightweight approach, which is well suited for applications with a smaller number of Views and conceptual tight coupling between presentation and data. [Osm12]

The client application of Project Zoom only has a few Views, as described in section 2.1. The interactive graph (cf. use case U1 and U4) is a very close representation of the respective Model. Besides, it has to be custom developed, as there are no applicable standard components available. Based on these observations the MVC architecture has been selected for the client application of Project Zoom. Figure 3.2 shows the implemented architecture. The components shown are detailed in later sections.

3.1.2. Event-driven programming

Users expect computer systems to respond quickly. This is especially true for web applications [Sel99]. In any case users expect the interface to be non-blocking when the system is performing long running tasks [Nie94]. In web applications this is achieved by asynchronous APIs for tasks like

[1] Data-binding is a technique where properties of a Model can be assigned to user interface components in a way that changes from either one are reflected to the other. [BJG+04]

requesting data from the server [2] or waiting for user input [3]. An event-driven system is a popular solution for dealing with asynchronous code execution [Mic06].

Events are messages sent between components in a system. The receiver has to subscribe to a type of events, which senders then eventually publish (cf. Observer pattern [GHJV94]). There are two categories of events:

- Global events are only identified by their names and can be received by any object in the system.
- Object specific events are identified by both their names and sender objects. Receivers need to have a reference to the sending object when subscribing to this kind of events.

Event-driven programming is a concept where components of a system heavily communicate via events. Particularly, data changes and user actions are propagated using events. Taking an event-driven approach leads to looser coupling between components and consequently, to better testable and maintainable code [Fai11].

Requesting data from the server (cf. use case U3), waiting for user inputs and listening to data synchronization messages (cf. requirement R3) are asynchronous tasks that the Project Zoom client performs. Using an event-driven architecture allows the system to pass data streams efficiently through the system.

3.1.3. Synchronization

As a collaborative web application, Project Zoom allows multiple users to access and manipulate shared resources (cf. requirement R3). There are four popular approaches for dealing with synchronization issues in distributed systems: Locking, Operational Transformation, three-way merges and Differential Synchronization [Fra09].

Locking is a mechanism to enforce mutual exclusion and is a standard solution for synchronization [Dij65]. Figure 3.3 outlines how locking ensures consistency as only one user has access to the resource at a time. Due to its ensured consistency locking is implemented in a variety of applications, e.g. ACID[4]-compliant database systems, office applications[5] and document-collaboration software[6]. As real-time collaboration[7] systems require concurrent access to shared resources, locking is not suitable for this class of systems.

[2]XMLHttpRequest, `http://www.w3.org/TR/XMLHttpRequest/`, accessed 06/19/13

[3]Document Object Model (DOM) Events, `http://www.w3.org/TR/DOM-Level-3-Events/`, accessed 06/19/13

[4]Atomicity, Consistency, Isolation and Durability (ACID) are a set of properties that database transactions guarantee. [G+81]

[5]File locking in Microsoft Word, `http://support.microsoft.com/default.aspx?scid=kb;EN-US;176313`, accessed 06/25/13

[6]MediaWiki Documentation: LockManager, `https://doc.wikimedia.org/mediawiki-core/master/php/html/classLockManager.html`, accessed 06/25/13

[7]real-time collaboration is a mechanism where the changes to a document on a client are immediately propagated to the other clients. Because of network-latency, the real-time property is in this context not as narrow as in the fields of operating systems or computer graphics. [SC02]

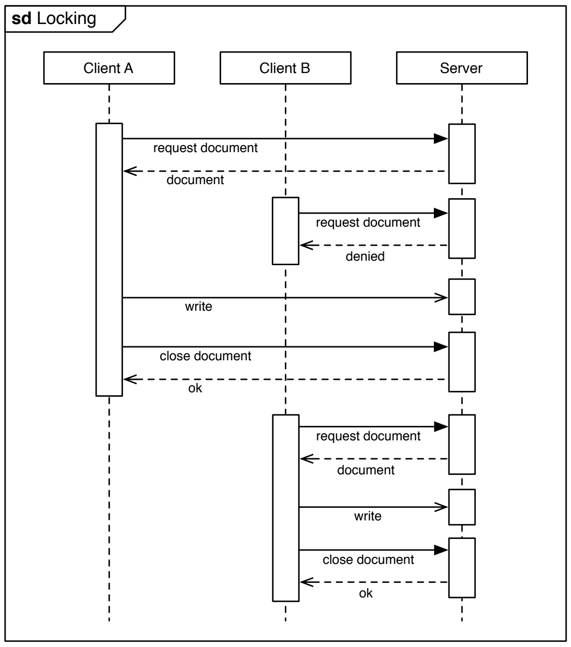


Figure 3.3. – Sequence diagram of synchronization with locking

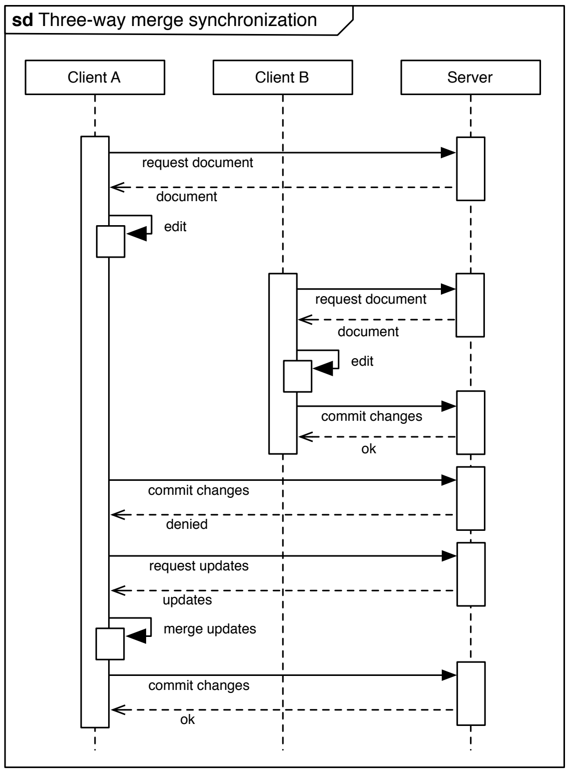


Figure 3.4. – Sequence diagram of three-way merge synchronization

The **three-way merge** is a synchronization approach that has been implemented in some revision control systems [PCSF08]. As shown in figure 3.4, the clients hold a local version of a document and make edits to it individually. When synchronizing these changes, the client first requests updates from the server and merges them locally. Then, the synchronized data is sent back to the server. A system using the three-way merge approach is not guaranteed to be consistent as synchronization is only triggered when a client publishes an edit. This edit is not automatically propagated to other collaborating clients. Therefore, this approach is not suitable for real-time collaboration systems either.

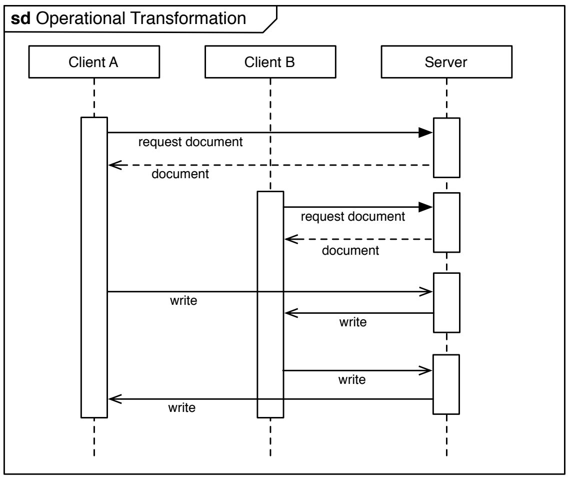


Figure 3.5. – Sequence diagram of Operational Transformation

Operational Transformation (OT) is a class of algorithms that use event-passing for synchronization [EG89]. Figure 3.5 shows the interaction scheme of OT. All user actions on the document are captured in form of change events and then sent to the collaborating clients. These change instructions are commonly known as patches. *Eventual consistency*[8] can be achieved through different models [SJZ+98] [LL04] [LL05]. Google Docs[9] is a popular example application that uses OT.

Differential Synchronization is very similar to OT [Fra09]. However, instead of capturing all changes as they happen, the change instructions are distilled by comparing two snapshots of the document. This approach is preferred when capturing all the user edits is a practical challenge.

Even though concurrent editing of the same document is not a requirement, concurrent read access is (cf. requirement R3). Using automatic updates instead of manual ones greatly enhances the user experience. Both Operational Transformation and Differential Synchronization support real-time collaboration. Therefore, they are the best-suited approaches for Project Zoom. Operational Transformation has been selected, because the system is already built using an event-based architecture in both the server [Boc13] and the client (cf. section 3.1.2).

ok, this is too less conceptual

[8]Eventual consistency describes that all accesses to a resource yield the same result when there have been no writes in a particular time span. [GA02]

[9]Google Docs, `https://docs.google.com/`, accessed 06/25/13

3.1.4. Technology

Since Project Zoom is a web application, the client code needs to be written in JavaScript[10]. The presentation layer is built using the standard HTML[11] and CSS[12] technologies. The interactive graph relies on SVG[13] because of its unique zooming and hit-test[14] capabilities. The d3[15] library is used to manipulate the SVG document.

3.2. Model

The Model is the component that is responsible for fetching the data from the server, listening to changes and passing changes back to the server. This section covers the techniques the client applies to handle the domain data on an abstract level. Bocklisch's thesis describes the actual domain data model in detail [Boc13].

3.2.1. Representing the data in the client application

The data is represented through two container classes that wrap around the native objects of JavaScript: `DataItem` and `DataCollection`. Both classes extend the native objects with getter and setter methods for accessing and manipulating the respective properties or items. Using getters and setters is a popular technique of tracking changes in the data and emitting corresponding events [Osm13]. An alternative approach to container classes would be the `Object.observe` API[16] which has yet to be standardized [Wal12].

The container classes can be used to build hierarchical tree structures. In such a structure, property changes of child objects are then propagated to their ancestor objects. In addition, nested properties can be accessed through the parent's `get` method using the JSON Pointer [17] syntax.

Both container classes provide methods for fetching data from the server. For that they use the XMLHttpRequest [ASSvK12] API of the browser. This technique is commonly referred to as Asynchronous JavaScript and XML (AJAX) [Gar05]. There is also support for *lazy loading*[18] of properties or subtrees.

[10] JavaScript is a scripting language that was designed for the use in web browsers and was standardized under the name ECMAScript `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`.
[11] Hypertext Markup Language, `http://www.w3.org/TR/html5/`, accessed 06/19/13
[12] Cascading Style Sheets, `http://www.w3.org/TR/css-2010/`, accessed 06/19/13
[13] Scalable Vector Graphics, `http://www.w3.org/TR/SVG/`, accessed 06/19/13
[14] Hit-testing is a technique to determine which user interface element intersects with the user's cursor [FvDFH95].
[15] Data-Driven Documents, `http://d3js.org/`, accessed 06/19/13
[16] Harmony Observe (proposed standard), `http://wiki.ecmascript.org/doku.php?id=harmony:observe`, accessed 06/28/13
[17] RFC 6901, JavaScript Object Notation (JSON) Pointer, `http://tools.ietf.org/html/rfc6901`, accessed 06/20/13
[18] Lazy loading is a technique where data is only loaded once it is required, instead of loading it upon initialization. [Fow02]

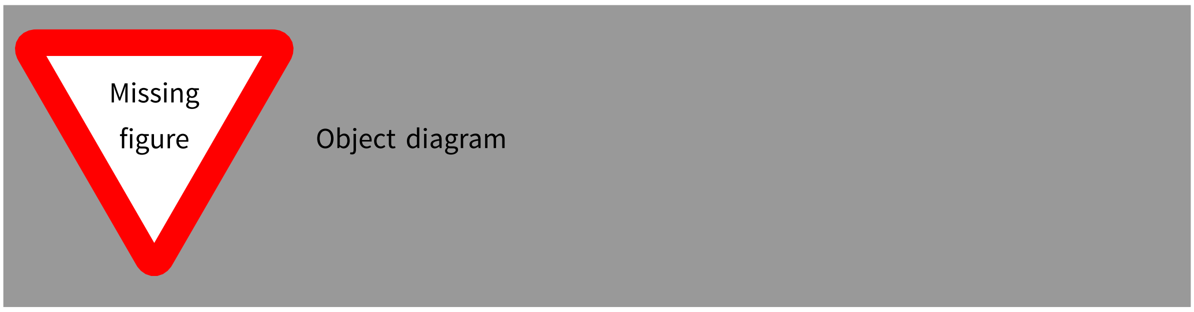


Figure 3.6. – Illustration of an example domain data structure.

3.2.2. Connecting to the server's REST interface

The server provides the data through a REST interface. Representational State Transfer (REST) is an architectural style on top of HTTP where resources are accessible through non-mutable URLs and are accessed and manipulated through the HTTP methods instead of custom URLs. [Fie00]

The REST interface for Project Zoom relies on the JSON[19] format for data exchange. As JSON is based on a subset of JavaScript, is very easy to parse and create JSON documents through native APIs. The full specification of the project's REST interface is attached in the Appendix A.

Upon initialization the Model requests the `projects` and `tags` collections from the server. This is useful because all business logic depends on these data collections. Lazy loading would increase page-loading time even further. As shown in figure 3.6 the projects collection is one of the root data structures in the system.

3.2.3. Synchronizing changes with the server

Changes to the server are transmitted using a patch-based format. Patches are documents that describe changes between two versions of a document. In many scenarios, patches have a substantially smaller footprint than their referenced document. Therefore, they are faster to send over a network. Furthermore, applying patches instead of replacing whole documents is less likely to cause consistency errors [EG89]. Patches are also a key concept in Operational Transformation, which has been employed to support real-time collaboration.

Project Zoom uses the recently introduced JSON Patch standard [BN13]. JSON Patch is a format for describing a list of mutations in an existing JSON document. A mutation entry contains an operation identifier, e.g. `add`, `remove` or `replace`, as well as a property address using the JSON Pointer syntax and a new value. Appendix B shows an example of a JSON patch applied to a JSON document. An accumulator object generates the JSON patches. It connects to a `DataItem` object and listens to the change events. As `DataItems` propagate change events from their child nodes,

[19]RFC 4627, The application/json Media Type for JavaScript Object Notation (JSON), `http://tools.ietf.org/html/rfc4627`, accessed 06/20/13

Listing 3.1 – Pseudo code for compacting a chronologically ordered list of JSON patches

```
foreach patch1, i in patches
  # patch1.item is the object that is referenced by patch1.path

  if patch1 is marked as overridden
    remove patch1

  foreach patch2, j in entries where i > j
    if patch2 removes patch1.item or any parent of patch1.item
      remove patch1
      mark patch2 as overridden

    else if patch2 replaces patch1.item or any parent of patch1.item
      remove patch2

    else if patch2 replaces or removes a child of patch1.item
      merge patch2 into patch1
      mark patch2 as overridden
```

the accumulator can be attached to any node and will receive the change events from the complete subtree. For each change the accumulator appends a new patch entry to a list buffer, which will eventually be sent to the server.

For some user actions, e.g. dragging an element across the canvas, the number of patch entries can grow very fast. To minimize the transportation footprint of the patches, the accumulator provides a method for compacting patches by reducing the amount of redundant entries in the patch. The proposed algorithm is shown in listing 3.1 and has a time complexity of $\mathcal{O}(n^2)$. Compacted patches are then sent to the server using the HTTP PATCH method.

The client is designed to listen to changes sent from the server. For that the client opens a WebSocket[20] connection. The server sends JSON patches with an accompanying resource identifier, which get applied to the data structure in the client. Because of the event-driven architecture of the client, these changes will be propagated to the View immediately. As concurrent editing of a single resource (e.g. an interactive graph) is not a requirement (cf. requirement R3), conflicting edits will be rejected by the server.

3.3. View

Longer intro

The View is responsible for rendering the data to the user interface.

[20]WebSocket, `http://tools.ietf.org/html/rfc6455`, `http://www.w3.org/TR/2009/WD-websockets-20091222/`, both accessed 06/23/13

3.3.1. Component interface

The View is assembled by a hierarchical structure of View component instances. These instances share a common interface as shown in figure 3.7. The View objects are responsible for controlling one element in the Document Object Model (DOM), which is referenced through the attribute `el`. At this stage the View component is not interactive yet. The method `activate` enables interactivity, by attaching event handlers to the DOM. `deactivate` then removes these event handlers, making the View component non-interactive again. This approach allows other components to control which View components are currently active. Consequently, parent components are responsible for activating and deactivating their children. The View components do not attach their element to the DOM themselves.

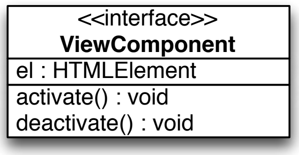


Figure 3.7. – Class diagram of the View component interface

3.3.2. Data integration

Upon initialization View components are assigned with a `DataItem` or a `DataCollection`. The contained data is then used to render the user interface. Views tap into the event system and listen to changes from the Model to update accordingly. For example, when a user moves the cursor across the canvas to drag a graph node, the `position` property of the node is being altered by a Controller (cf. section 3.4.2). This change is propagated to the View, which then renders the node at its new position. Server-sent changes are handled in the very same way. So, the View is agnostic to where the event originated.

3.3.3. Architecture

There are three main Views: Overview View, Details View and Process View (cf. section 2.1). As shown in figure 3.2 these are represented by three hierarchical View component structures. Figure 3.8 is an example of how the individual components have distributed responsibilities across the user interface.

3.4. Controller

The Controller components handle user interactions and manipulate the data in the Model as well as the Views.

Longer intro

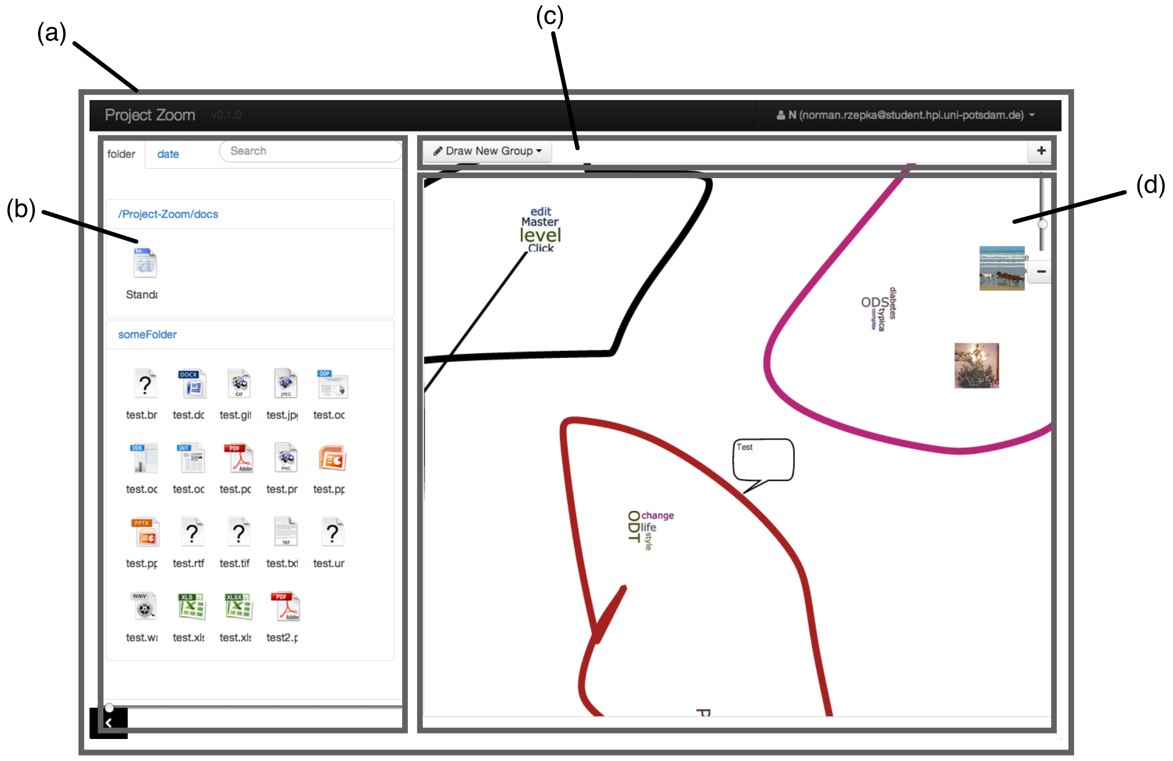


Figure 3.8. – Screenshot of the Process View with component annotations: *(a)* ProcessView, *(b)* ArtifactFinder, *(c)* Toolbar, *(d)* Graph

	Overview View	Details View	Process View
0-9	active	-	-
10-19	transition-out	transition-in	-
20-29	-	active	-
30-39	-	transition-out	transition-in
40-150	-	-	active

Figure 3.9. – Zoom level configurations

3.4.1. Main controller

In Project Zoom, zooming is the main method for navigating from one View to another (cf. section 2.1). The main Controller handles this zoom-based behavior. For that the domain of zoom levels is split into several subranges as shown in figure 3.9. The controller enforces states of the main Views in each subrange as described.

The Controller works with the global range of zoom levels. This global zoom level is decoupled from the zoom level that the individual Views work with. Because of this it is easier to alter the parameters in the controller while maintaining the assumptions in the View components and vice versa. The controller converts the global zoom level into the specific levels using pluggable functions.

3.4.2. Behavior controllers

In addition to the main controller there are several other controllers that encapsulate a particular behavior in one of the main views, e.g. dragging nodes, drawing clusters or commenting. These behaviors share an interface as shown in figure 3.10, which is similar to the interface of the View components . Again, the `activate` and `deactivate` methods enable or disable the interactivity provided by the respective behavior. Behaviors operate independently of each other and only communicate via events. For example, the deletion behavior emits an event whenever a node has been removed from the graph. The selection behavior recognizes this event and eliminates any references it had to that node. This loosely coupling makes it easy to extend the functionality of the application by adding custom behaviors. Behaviors alter the data in the Model. Because of the event-driven architecture these alterations are propagated to update the View and sent to the server for persistence.

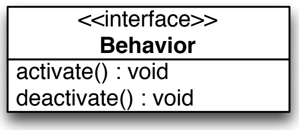


Figure 3.10. – Class diagram of the Behavior interface

4. Implementation considerations

This section documents some of the implementation problems that have been solved while developing the client application of Project Zoom. Some of the concepts introduced here are important to understand when extending the system.

better motivation

4.1. Module and file management

Modularization is a common pattern in software development. Modules are pieces of code that have a particular responsibility within the system. They provide a well-defined interface for other modules to consume and explicitly list their dependencies. This decoupling leads to better maintainability, as the code is encapsulated and modules may easily be replaced. [Osm11]

JavaScript in its current version does not support modules natively. A common approach is to separate the code into different files and have them loaded through `<script>` tags into the DOM. This technique is prone to naming conflicts. As the files share the same global scope, variables are shared as well. This can be avoided by wrapping a file's code in an immediate function [RB13]. However, to export their functionality the files usually append properties to the global object, e.g. `window.jQuery`. For larger systems this leads to *namespace pollution*. Another issue is that the files have no means of declaring their dependencies programmatically. Thus, the ordering of the `<script>` tags is significant.

The Asynchronous Module Definition (AMD) format[1] provides a module implementation for JavaScript. Modules are defined by specifying a function that creates the module and a list of dependencies. In conjunction with an AMD script loader, these modules can be loaded asynchronously. The script loader ensures that the respective dependencies are resolved in advance. In addition, there are tools for concatenating the module files.[2] The AMD format has been used to organize the code of the client application.

4.2. Testability

Automatic tests are a popular technique for improving code quality. For the tests to be effective, they have to be run in isolation. Therefore, there are some principles when writing code to improve testability [Tro13]. The code needs to be properly modularized to enforce decoupling. Additionally,

[1]Asynchronous Module Definition, `https://github.com/amdjs/amdjs-api/wiki/AMD`, accessed 06/27/13
[2]Concatenating scripts into one file is a common technique to reduce page-loading time by decreasing the required HTTP requests.

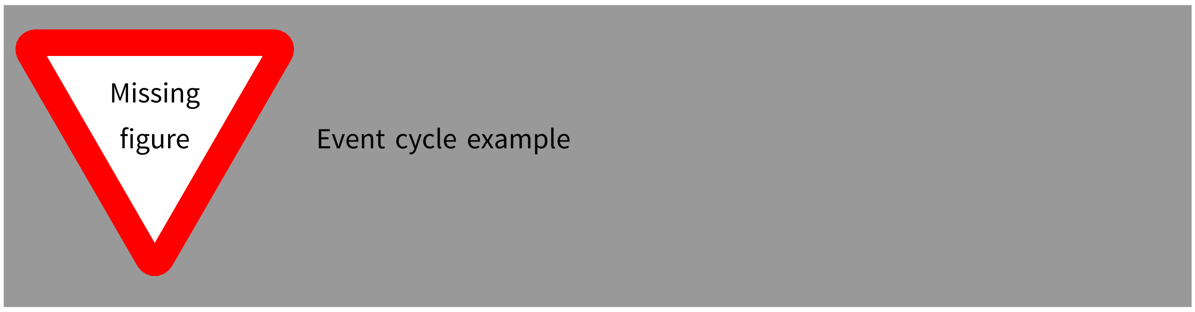


Figure 4.1. – An example of an infinite loop in the event-system.

side effects should be encapsulated in separate modules. When running tests on a module, all of its dependencies should be replaced by mock objects[3]. Popular script loaders for AMD modules allow replacement rules for resolving dependencies. There is a test suite that covers parts of the client's Model component by testing the behavior of the data wrapper and event system classes.

4.3. Memory leaks in JavaScript

JavaScript is a managed-memory language. Runtime implementations include a garbage collector (GC) that frees unnecessary objects. To determine the state of an object, its references in the system are examined. If an object has no references that are reachable from a particular set of root objects, it is considered unnecessary. Since there are higher order functions in JavaScript and scopes are realized through closures[4], removing all references of an object may become a tedious task in larger systems.

A popular solution to this problem is to employ an event dispatcher. An event dispatcher is a singleton[5] that keeps track of all callbacks (including event handlers) as well as their sender and receiver objects. Thus, removing all callbacks related to an object is reduced to a single method invocation. This solution is implemented in the client's Model component.

4.4. Event cycles

When using an event-based system it is possible to create infinite execution loops. An example is shown in figure 4.1. A common solution to that problem is to keep track of the object that initiated an event and omit event handlers of that object while propagating the event.

citation missing, maybe any data-binding implementation

[3]Mock objects mimic the behavior of real objects but execute in a controlled way. [Osh09]
[4]Closures are execution contexts that allow a function to access variables that are external to its definition. [RB13]
[5]A singleton is an instance of a class that is guaranteed to have only one instance. [GHJV94]

4.5. Event congestion

As shown in figure 4.1 some changes to data objects are triggered by native browser events, such as `mousemove`. Their frequency is determined by the sample rate of the input device (e.g. mouse or trackpad) and may be as high as 60 signals per second. This is a favorable effect, as it enables a lag-free user interface. However, as the event-driven architecture is designed to propagate change events not only within the client application but also to the server, this high event rate may lead to a congestion of the network connection.

A solution to this problem is to throttle the network requests based on a fixed time interval. With this technique, temporary states in which the client has not yet completed his action, e.g. not released the mouse while still dragging, are also propagated to the server. Addressing this issue, there is another approach that only sends requests after there have been no events for a fixed time span. This mechanism, which is called *debouncing*, is used for the client's synchronization with the server.

citation missing, maybe some functional programming book or underscore doc

5. Evaluation

This section explains how the proposed architecture of Project Zoom enables the use cases and meets the requirements that were previously established.

maybe longer intro

U1 Figure 5.1 shows how files are pulled from a storage provider by the server[1] and sent to the client. Because of the implemented dragging behavior, a user is able to add these documents as a node to the visual graph. Using context-sensitive actions these nodes may be connected to others or annotated in different ways [Her13].

U2 The D-School uses a FILEMAKER[2] database to store projects. The server polls that database. The client can then access them through a REST interface and display them in a user interface. This interaction is outlined in figure 5.2.

U3 Project Zoom is designed to use a web-based client-server architecture. The server software is capable of handling requests from clients over a network connection. If the software is deployed on a public server, the application will be reachable from any internet-connected computer. Use case U3 requires the users to have an installation of an HTML5-capable browser to access the application. Such web browsers are very popular and likely to be already preinstalled on the user's computer.

[1] Werkmeister covers how the data is pulled from different kinds of storage providers, e.g. Box and FILEMAKER. [Wer13]
[2] Filemaker, `http://www.filemaker.com/`, accessed 06/16/13

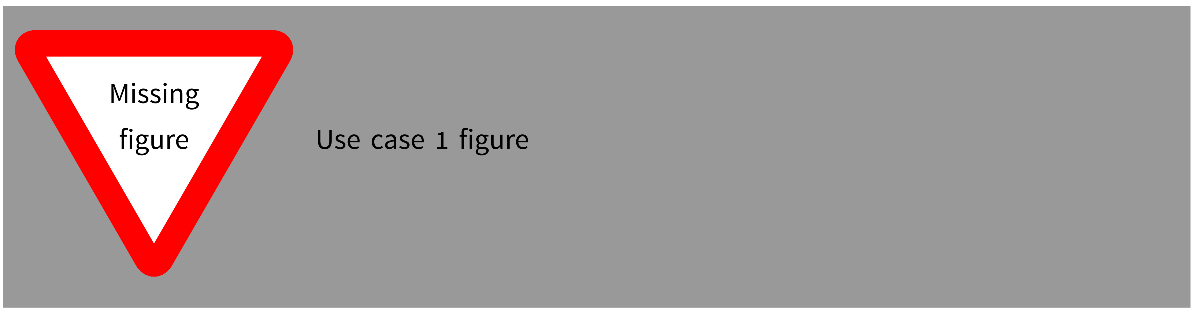


Figure 5.1. – Sequence diagram for use case 1.

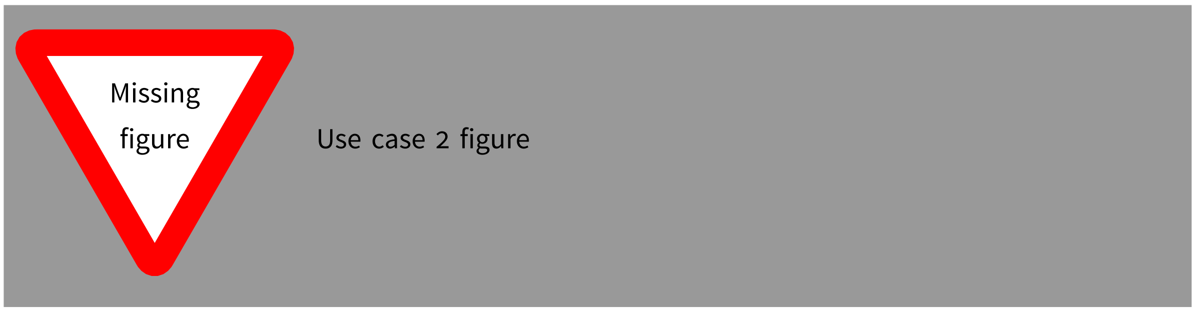


Figure 5.2. – Sequence diagram for use case 2.

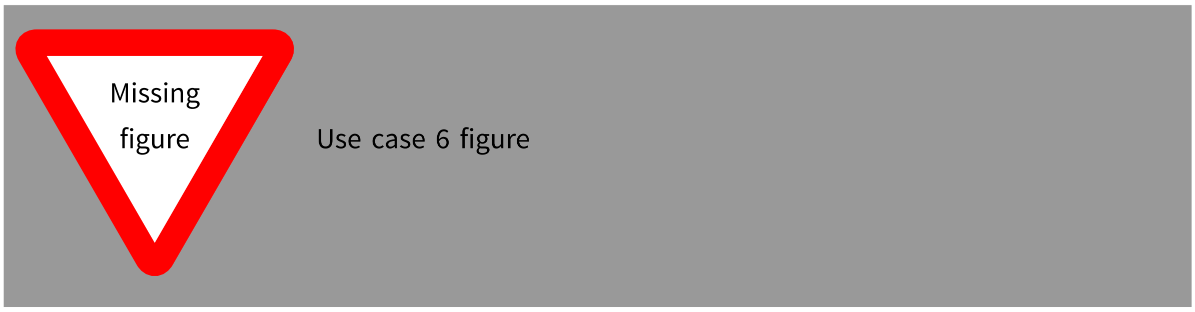


Figure 5.3. – Photo of Project Zoom running on an Apple iPad (3rd generation).

U4 When a graph in Project Zoom has been edited, it is automatically stored shortly afterwards in the server's database. Every version of a particular graph is kept in the database for later retrieval [Boc13]. The server's REST interface includes a method for retrieving a specific version of a graph (cf. Appendix A).

U5 D-School students store the files they create in their projects using cloud storage providers, e.g. Box. The server includes a connector to the Box API. The extensible architecture of the server also allows to integrate with other services [Wer13]. Because of the event-driven architecture of the client, added files are displayed in the user interface after a very short time (cf. figure 5.1).

U6 The D-School runs a variety of students projects. Using the FILEMAKER database, Project Zoom is able to display them in the user interface. Users are able to select a filter, which only shows the matching items [Die13]. Since the client of Project Zoom is implemented using HTML5 technologies, it also runs on popular mobile tablet devices, as demonstrated in figure 5.3.

R1 The requirement **R1** has already been covered by U6.

R2 The requirement **R2** has already been covered by U3.

R3 Due to the nature of a web application, multiple clients can connect to a server. Thus, multiple users can access the same application concurrently. The system does not impose restrictions on concurrent read access through the client's user interface for any resource. An Operational Transformation algorithm is applied to provide a real-time collaboration experience and ensure eventual consistency. Because of the very basic handling of consistency conflicts, concurrent editing of the same resource (e.g. a graph) is not supported in the current implementation and subject to future work.

R4 The requirement **R4** has already been covered by U5.

6. Conclusion

tbd

6.1. Summary

This thesis first covered the design of Project Zoom. Its user interface is offers three different main views that address the needs of different stakeholders. Users navigate from one view to the next by zooming in or out.

- design, multi-layered, semantic zoom for multiple stakeholders, semantic zoom
- web-based application for platform support, distributed
- rest interface
- mvc pattern
- event-driven
- operational transformation as synchronization

6.2. Future Work

- conflict resolution
- other applications

List of Figures

Bibliography

[ASSvK12] Julian Aubourg, Jungkee Song, Hallvord R. M. Steen, and Anne van Kesteren. Xmlhttprequest, w3c working draft, December 2012.

[BBD$^+$13] Tom Bocklisch, Dominic Bräunlein, Anita Dieckhoff, Tom Herold, Norman Rzepka, and Thomas Werkmeister. Software requirements specification. Unpublished. Hasso Plattner Institute, 2013.

[Ber96] Alex Berson. *Client/server architecture (2nd ed.)*. McGraw-Hill, Inc., New York, NY, USA, 1996.

[BJG$^+$04] Samuel W Bent, David J Jenni, Namita Gupta, Robert A Relyea, and Jeffrey L Bogdan. Data binding, September 2004. US Patent App. 10/939,881.

[BN13] Paul C. Bryan and Mark Nottingham. Rfc6902, javascript object notation (json) patch, April 2013.

[Boc13] Tom Bocklisch. Eine architektur für ein ereignisgesteuertes webbasiertes backend für project-zoom, 2013. Bachelor thesis. Hasso Plattner Institute.

[Brä13] Dominic Bräunlein. Generierung und bereitstellung von semantischen thumbnails aus heterogenen daten für project-zoom, 2013. Bachelor thesis. Hasso Plattner Institute.

[CL11] Andre Charland and Brian Leroux. Mobile application development: web vs. native. *Commun. ACM*, 54(5):49–53, May 2011.

[Col00] Adrian Mark Colyer. High-availability www computer server system with pull-based load balancing using a messaging and queuing unit in front of back-end servers, February 8 2000. US Patent 6,023,722.

[DDKN11] Sebastian Deterding, Dan Dixon, Rilla Khaled, and Lennart Nacke. From game design elements to gamefulness: defining "gamification". In *Proceedings of the 15th International Academic MindTrek Conference: Envisioning Future Media Environments*, MindTrek '11, pages 9–15, New York, NY, USA, 2011. ACM.

[Die13] Anita Dieckhoff. Layout-funktionalität für interaktive graphen für project zoom, 2013. Bachelor thesis. Hasso Plattner Institute.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569–, September 1965.

[EG89] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. *SIGMOD Rec.*, 18(2):399–407, June 1989.

[Fai11] Ted Faison. *Event-Based Programming: Taking Events to the Limit*. Apress, 1 edition, 12 2011.

[Fie00] Roy Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, 2000.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 1 edition, 11 2002.

[Fra09] Neil Fraser. Differential synchronization. In *Proceedings of the 9th ACM symposium on Document engineering*, DocEng '09, pages 13–20, New York, NY, USA, 2009. ACM.

[FvDFH95] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice in C (2nd Edition)*. Addison-Wesley Professional, 2nd edition, 8 1995.

[G$^+$81] Jim Gray et al. The transaction concept: Virtues and limitations. In *Proceedings of the Very Large Database Conference*, pages 144–154, 1981.

[GA02] Sanny Gustavsson and Sten F. Andler. Self-stabilization and eventual consistency in replicated real-time databases. In *Proceedings of the first workshop on Self-healing systems*, WOSS '02, pages 105–107, New York, NY, USA, 2002. ACM.

[Gar05] Jesse James Garrett. Ajax: A new approach to web applications, February 2005.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 11 1994.

[GHOS96] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, SIGMOD '96, pages 173–182, New York, NY, USA, 1996. ACM.

[GS91] J. Gray and D.P. Siewiorek. High-availability computer systems. *Computer*, 24(9):39–48, 1991.

[Her13] Tom Herold. Kontextsensitver assistent für interaktive graphen, 2013. Bachelor thesis. Hasso Plattner Institute.

[KP$^+$88] Glenn E Krasner, Stephen T Pope, et al. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, 1(3):26–49, 1988.

[LL04] Du Li and Rui Li. Preserving operation effects relation in group editors. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, CSCW '04, pages 457–466, New York, NY, USA, 2004. ACM.

[LL05] Rui Li and Du Li. Commutativity-based concurrency control in groupware. In *Collaborative Computing: Networking, Applications and Worksharing, 2005 International Conference on*, pages 10 pp.–, 2005.

[Mic06] Brenda M Michelson. Event-driven architecture overview. *Patricia Seybold Group*, 2, 2006.

[Nie94] Jakob Nielsen. Enhancing the explanatory power of usability heuristics. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '94, pages 152–158, New York, NY, USA, 1994. ACM.

[Osh09] Roy Osherove. *The Art of Unit Testing: With Examples in .Net*. Manning Publications, 1 edition, 7 2009.

[Osm11] Addy Osmani. Writing modular javascript with amd, commonjs and es harmony. Online, October 2011.

[Osm12] Addy Osmani. *Learning JavaScript Design Patterns*. O'Reilly Media, 1 edition, 8 2012.

[Osm13] Addy Osmani. *Developing Backbone.js Applications*. O'Reilly Media, 5 2013.

[PCSF08] C. Michael Pilato, Ben Collins-Sussman, and Brian W. Fitzpatrick. *Version Control with Subversion*. O'Reilly Media, second edition edition, 9 2008.

[PF93] Ken Perlin and David Fox. Pad: an alternative approach to the computer interface. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '93, pages 57–64, New York, NY, USA, 1993. ACM.

[PMW09] Hasso Plattner, Christoph Meinel, and Ulrich Weinberg. *Design Thinking*. mi-Wirtschaftsbuch, München, 2009.

[RB13] John Resig and Bear Bibeault. *Secrets of the JavaScript Ninja*. Manning Publications, pap/psc edition, 1 2013.

[Ree79] Trygve Reenskaug. Thing-model-view-editor – an example from a planningsystem. http://de.scribd.com/doc/6414921/Original-MVC-Pattern-Trygve-Reenskaug-1979, May 1979.

[SC02] Chengzheng Sun and David Chen. Consistency maintenance in real-time collaborative graphics editing systems. *ACM Trans. Comput.-Hum. Interact.*, 9(1):1–41, March 2002.

[Sch01] R. Schollmeier. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In *Peer-to-Peer Computing, 2001. Proceedings. First International Conference on*, pages 101–102, 2001.

[Sel99] Paula Selvidge. How long is too long to wait for a website to load. *Usability news*, 1(2), 1999.

[SJZ$^+$98] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Trans. Comput.-Hum. Interact.*, 5(1):63–108, March 1998.

[SZ05] Ravi Sandhu and Xinwen Zhang. Peer-to-peer access control architecture using trusted computing technology. In *Proceedings of the tenth ACM symposium on Access control models and technologies*, SACMAT '05, pages 147–158, New York, NY, USA, 2005. ACM.

[Tak12] Mikito Takada. Single page apps in depth, October 2012.

[Tro13] Mark Ethan Trostler. *Testable JavaScript*. O'Reilly Media, 1 edition, 1 2013.

[Wal12] Rick Waldron. Javascript: Object.observe, August 2012.

[Wer13] Thomas Werkmeister. Extensible backend architecture for data aggregation of heterogeneous sources for project-zoom, 2013. Bachelor thesis. Hasso Plattner Institute.

Appendix A.

REST interface

A.1. Datatypes

identifier	example
int	3
double	3.5
bool	true
”string”	”test”
”object_id”	”512d2218c2c1804377000005”[1]
”date”	”2013-02-26T02:00:00Z”[2]
{object}	`{ ”test”: 123 }`

A.2. General

```
GET /resources
# returns the 50 first resources

GET /resources?limit=100
# returns the 100 first resources

GET /resources?offset=30&limit=10
# returns 10 resources from the 30th

GET /resources/:id
# returns all details of a specific resource


PUT /resources/:id
# replaces this item
=> 200 OK
or
=> 400 Bad Request
   {
     errors: []
   }
```

[1]Mongo ObjectId, `http://docs.mongodb.org/manual/reference/object-id/`, accessed 25/06/13
[2]ISO 8601, `http://en.wikipedia.org/wiki/ISO_8601`, accessed 25/06/13

```
PATCH /resources/:id
# incrementally updates this item
=> 200 OK
or
=> 400 Bad Request
   {
     errors: []
   }


POST /resources
# creates a new item
=> 200 OK
or
=> 400 Bad Request
   {
     errors: []
   }


DELETE /resources/:id
# deletes this item
=> 200 OK
or
=> 400 Bad Request
   {
     errors: []
   }
```

A.3. projects

```
GET /projects
{
  ”limit”: int,
  ”offset”: int,
  ”content”: [
    {
      ”id”: ”object_id”,
      ”name”: ”string”,
      ”length”: ”string”,
      ”season”: ”string”,
      ”year”: ”string”,
      ”_graphs”: [”object_id”, …],
      ”_tags”: [”object_id”, …]
    },…
  ]
}

GET /projects/:id
{
  ”id”: ”object_id”,
  ”name”: ”string”,
  ”length”: ”string”,
  ”season”: ”string”,
  ”year”: ”string”,
  ”_graphs”: [”object_id”, …],
  ”_tags”: [”object_id”, …]
}

GET /projects/:id/artifacts
{
  ”limit”: int,
  ”offset”: int,
  ”content”: [
    {
      ”id”: ”object_id”,
      ”name”: ”string”,
      ”path”: ”string”,
      ”projectName”: ”string”,
      ”resources”: [
        {
          ”hash”: ”string”,
          ”name”: ”string”,
          ”typ”: ”string”
        },…
      ],
      ”createdAt”: int,
      ”isDeleted”: bool,
      metadata: {object}
    },…
  ]
}


POST /projects/:id/graphs
=> 303 See other
Location: /graphs/:graph_group
```

A.4. artifacts

```
GET /artifacts/:id
{
  ”id”: ”object_id”,
  ”name”: ”string”,
  ”path”: ”string”,
  ”projectName”: ”string”,
  ”resources”: [
    {
      ”hash”: ”string”,
      ”name”: ”string”,
      ”typ”: ”string”
    },…
  ],
  ”createdAt”: int,
  ”isDeleted”: bool,
  metadata: {object}
}

GET /artifacts/:id/:typ/:name
=> 200 OK
Content-Type: application/octet-stream
```

A.5. tags

```
GET /tags
{
  ”limit”: int,
  ”offset”: int,
  ”content”: [
    {
      ”color”: {”r”: int, ”g”: int, ”b”: int},
      ”id”: ”object_id”,
      ”name”: ”string”
    },…
  ]
}

GET /tags/:name
{
  ”color”: {”r”: int, ”g”: int, ”b”: int},
  ”id”: ”object_id”,
  ”name”: ”string”
}
```

A.6. users

```
GET /users
{
  ”limit”: int,
  ”offset”: int,
  ”content”: [
    {
      ”id”: ”object_id”,
      ”email”: ”string”
      ”firstName”: ”string”,
      ”lastName”: ”string”
    },…
  ]
}

GET /users/:email
{
  ”id”: ”object_id”,
  ”email”: ”string”
  ”firstName”: ”string”,
  ”lastName”: ”string”
}
```

A.7. graphs

```
GET /graphs
{
  ”limit”: int,
  ”offset”: int,
  ”content”: [
    {
      ”_project”: ”object_id”,
      ”id”: ”object_id”,
      ”clusters”: [
        {
          ”comment”: ”string”,
          ”content”: [int, int, …],
          ”id”: int,
          ”phase”: ”string”,
          ”waypoints”: [{ ”x”: int, ”y”: int },… ]
        },…
      ],
      ”edges”: [
        {
          ”from”: int,
          ”to”: int
        },…
      ],
      ”group”: ”string”,
      ”nodes”: [
        {
          ”id”: int,
          ”typ”: ”string”,
          ”position”: { ”x”: int, ”y”: int },
          ”payload”: {object}
        },…
      ],
      ”version”: int,
    },…
  ]
}

GET /graphs/:graph_group
{
  ”_project”: ”object_id”,
  ”id”: ”object_id”,
  ”clusters”: [
    {
      ”comment”: ”string”,
      ”content”: [int, int, …],
      ”id”: int,
      ”phase”: ”string”,
      ”waypoints”: [{ ”x”: int, ”y”: int },… ]
    },…
  ],
  ”edges”: [
    {
      ”from”: int,
      ”to”: int
    },…
  ],
  ”group”: ”string”,
  ”nodes”: [
    {
      ”id”: int,
      ”typ”: ”string”,
      ”position”: { ”x”: int, ”y”: int },
      ”payload”: {object}
    },…
  ],
  ”version”: int,
}

GET /graphs/:graph_group/:version
# Same as GET /graphs/:graph_group

PATCH  /graphs/:groupId/:version
{
  ”version”: int
}
```

Appendix B.

JSON Patch example

Listing B.1 – Initial JSON document

```
{ ”foo”: ”bar” }
```

Listing B.2 – JSON patch

```
[
  {
    ”op”: ”replace”, ”path”: ”foo”,
    ”value”: {
      ”baz”: ”bar”
    }
  },
  { ”op”: ”add”, ”path”: ”foo/qux”, ”value”: 123 }
]
```

Listing B.3 – Resulting JSON document

```
{
  ”foo”: {
    ”baz”: ”bar”,
    ”qux”: 123
  }
}
```

Eidesstattliche Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und dafür keine anderen als die genannten Quellen und Hilfsmittel verwendet habe.

Norman Rzepka

Potsdam, 29. Juni 2013