Enabling Software Architecture Comparison with ExplorViz

Bachelor's Thesis

Daniel Teut

September 30, 2019

Kiel University
Department of Computer Science
Software Engineering Group

Advised by: Prof. Dr. Wilhelm Hasselbring
Christian Zirkelbach, M.Sc.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, 30. September 2019

Abstract

Comparing software architectures can be a useful but complex task. A visual tool like ExplorViz is useful for that as changes can be seen on a glance. Currently ExplorViz does not offer the functionality for directly comparing software architecture. Therefore we will extend it to allow the comparison of two and more software architectures in this thesis. Following this we conduct a usability study for the implemented software.

Contents

Chapter 1

Introduction

1.1 Motivation

Most software changes over the course of its lifetime. This is due to, for example, bug fixes, new features, refactoring and other changes. Sometimes a developer might want to quickly assess what those changes were and when they occurred. They may also want to compare two similar architectures that are meant too be merged. It would be useful to know what parts of each architecture need to be changed.

But comparing two or more complex things is never easy and this is especially true for software. Such a comparison can be a quite daunting task. A naive approach would be the comparison of the file structure or the actual code and while this approach might work for simple architectures, it will be a lot harder to gain any meaningful information from complex software. It is also very slow and can lead to errors.

To simplify this process one might try a visual approach by visualizing and comparing each version. *ExplorViz* is a tool for such an approach. Here the software is shown as a landscape in which the user can view an application in a three dimensional environment. Currently the tool shows only one version at a time. A developer could of course use two instances of the tool and compare the landscapes by hand. This has two major drawbacks: It increases the complexity of the task, since they to have to use multiple instances at a time, and is still error prone and slow. Therefore it would be useful to show two software versions in one merged landscape that shows every difference between them. Additionally a history of every change that occurred between those versions would further increase the understanding of the differences and how the software changed over time. This might be useful for bug fixing and to see if, for example, a particular versions introduced them. Since ExplorViz currently does not offer this functionality the goal of this thesis will be the development and implementation of such a comparison and history.

1.2 Goals

First we will present the overall goals. These will be reached during the thesis. This will allow us to add the outlined functionality to ExplorViz.

1.2.1 G1: Researching Related Work

First we need to identify similar approaches in the literature including other software comparison tools and even other ExplorViz extensions. This allows us to build upon those solutions and further our understanding of what can and has be done on this topic.

1.2.2 G2: Formulating an Approach for Software Architecture Comparison

We must identify what we want to present to the user and how to present it. For this, an approach for the backend and the frontend must be formulated, especially the user interface of the frontend. This will then allow us to implement this approach.

1.2.3 G3: Implementation of a Software Architecture Comparison Approach in ExplorViz

As a main goal, the comparing of software architectures needs to be implemented in ExplorViz. This will be accomplished be developing two extensions for the tool, one for the backend and one for the frontend. The backend extension is realized as a microservice and the frontend extension as an ember addon.

1.2.4 G4: Evaluation of the Implemented Approach

To improve the usability of the previously implemented software and to test its functionality, we will collect feedback by conducting an evaluation. This will allow us to identify future improvements to the software and the user interface. For this we will conduct a usability study.

1.3 Document Structure

The thesis is structured as follows. First we will show the foundations and tools needed for the rest of the thesis in Chapter 2. This is followed by Chapter 3 with the envisioned approach mentioned above in G2. After this in Chapter 4 we will describe how to implement this approach inside ExplorViz. Chapter 5 then presents an evaluation of the implemented software. Then we will present a few related works that occur in the literature in Chapter 6. Finally Chapter 7 will close with potential future works on this topic and an overall conclusion.

Chapter 2

Foundations and Technologies

This chapter outlines every tool and technology used in this thesis. Especially ExplorViz and the tools it uses are described here since we will develop our extension for ExplorViz.

2.1 ExplorViz

ExplorViz is a tool for software visualization. It was first introduced by Fittkau, Waller, Wulf, and Hasselbring [1]. It uses landscapes to represent software systems. These landscapes are generated by runtime monitoring data. They are build hierarchically with systems, node groups, nodes, and applications [2]. Systems represent a "logical union of multiple applications and servers". They contain multiple node groups which contain multiple nodes of their own. A node is an abstraction of a server. Node groups represent multiple servers with the same configuration. They are named by the IP range they occupy. This can be for example useful for cloud architectures that can be dynamically scaled [3]. Finally there are applications that are deployed on the nodes. These are software running on the servers.

Here the switch between the two dimensional landscapes and the three dimensional application view occurs. By selecting a single application the user can view it. Figure 2.1 shows such an application view. The figure also shows the timeline where the live monitoring data is visualized. Here a city metaphor is used where the components (i.e. packages in Java) are represented as districts and the so called classes (i.e classes in Java) are buildings [1]. The components are also ordered hierarchically so components that are a child of another component are set on top of said component. Additionally the communication between the classes is shown as lines between the buildings. The width of those lines and the height of the buildings correspond to the number of method calls and the number of instances respectively.

ExplorViz uses a microservice approach [8, 7, 9]. Therefore its backend and frontend are separated and can be deployed on different servers. Any additional functionality is added by so called extensions. These are realized as their own webservices with their own server and, if necessary, their own database.

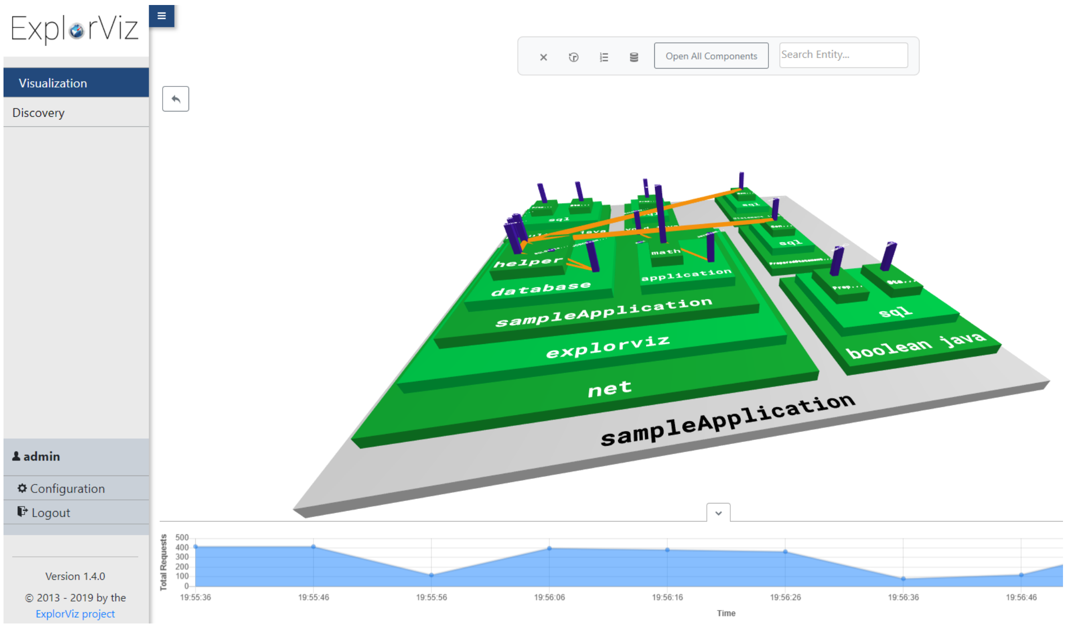


Figure 2.1. ExplorViz application view

2.2 Ember.js

Ember.js[1] is a Javascript webframework and is used by the ExplorViz frontend. It is based on the Model-View-Controller pattern. The view is handled by HTML patterns and relies on the Handelbars.js template engine[2]. These templates can be populated by a model defined by the developer. Each ember application defines routes which correspond to URL endpoints which can be accessed by the user. These routes can define models which fill the corresponding HTML templates. Additionally there are controllers for each route which define behavior for interactive elements in the template. The templates can contain components which have there own template and defined behavior similar to a controller of a route. They encapsulate certain elements of the website and increase the reusability. Each route and component has a lifecycle which the developer can interact with by so called *lifecycle hooks*. These hooks are simple methods that get called whenever a certain point in the lifecycle is reached. For routes this cycle consists of the transition from one route to another and for components the methods get called during the rendering of the component. For example, ExplorViz uses the "didRender" hook which gets called directly after a component was fully rendered, to start its rendering of landscapes. Another feature of Ember are *services* which are just simple javascript objects that provide properties and

[1] `https://emberjs.com`
[2] `https://handlebarsjs.com/`

functions. Ember allows for easy injection of those anywhere inside the application.

Ember uses addons to easily add functionality to an application. This is used by the ExplorViz frontend to realize its extensions. One such addon is *Ember data*. This addon is used to simplify the communication between the backend and frontend and to store data on the client side. For this purpose the service "store" gets automatically injected everywhere into the application. This service allows the developer to access the stored data and make queries to receive it. This data is represented by developer defined models. Additionally adapters can be defined to customize the communication between Ember data and the backend.

2.3 Jersey

Jersey[3] is a Java framework and is used by the ExplorViz backend for sending data from the backend to the frontend. This data is send via a REST (REpresentational State Transfer) API over HTTP(S). Jersey implements the JAX-RS API standard[4] for this. The backend exposes so called resources to the web, which can be accessed by the frontend. Each resource is represented by its own public class where each method can respond to a different query. The return values of those methods are the answers of the query. Java objects are automatically serialized, e.g. into the JSON format[5]. Jersey makes heavy use of Java annotations to set, for example, the type and path of its resources.

Additionally annotations are used for injecting classes into other classes. Jersey then automatically creates those objects when needed without the need for an explicit "new" call. This allows for simple management of different parts of the program and reduces a lot of boilerplate code.

2.4 NGINX

NGINX[6] is a reverse proxy. Since the backend consists of several servers with their own addresses a reverse proxy is used to allow the frontend to communicate with the backend. The reverse proxy provides a single host and port the frontend can use and depending on the URL the request is forwarded to the address of the appropriate microservice in the backend. In ExplorViz NGINX is simply setup as a docker image with a configuration file inserted.

[3] https://jersey.github.io/
[4] https://github.com/jax-rs
[5] http://json.org/
[6] https://www.nginx.com/

Chapter 3

Approach

This chapter provides an overview of the approach we will take to enable software architecture comparison in ExplorViz.

First we will give an overview on how ExplorViz is extended and then will present our approach for the backend and the frontend.

3.1 Overview

ExplorViz uses a microservice architecture[7, 9]. Figure 3.1 shows the software stack that ExplorViz uses. Each functionality in the backend is provided by its own service. For example the saving and loading of landscapes is handled by the HistoryService.

To now extend ExplorViz, we need to develop our own service for the backend and an ember addon to extend the frontend.

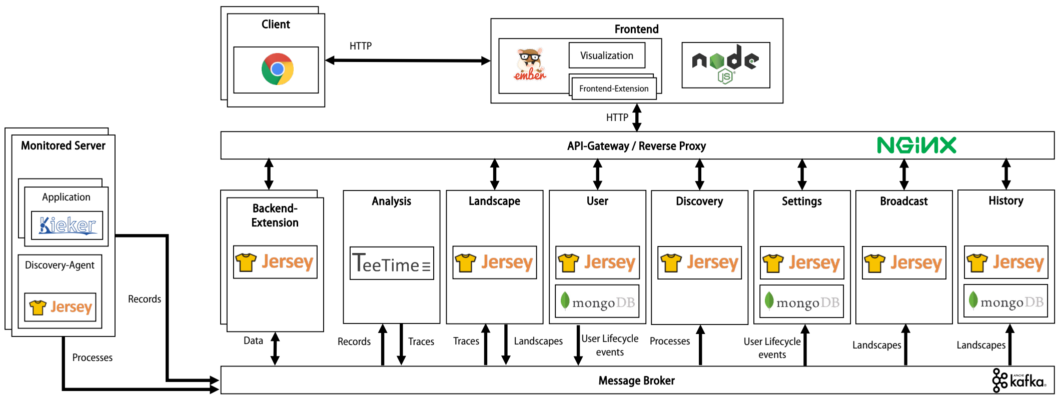


Figure 3.1. ExplorViz Software Stack [a]

[a]Source: `https://github.com/ExplorViz/Docs/blob/master/images/software-stack.pdf`

3.2 Backend Extension

The backend must provide two functionalities:

1. merge two landscapes and encode their differences inside a new merged landscape.
2. calculate the history between a list of landscapes.

We will present the approach for those problems in the following two sections.

3.2.1 Merging of Two Landscapes

The differences of the landscapes need to be saved somehow in the new merged landscape. The data model of ExplorViz allows for arbitrary attributes attached to each entity (i.e. each part of the hierarchy of the landscapes). We will use this to save the differences. Wegert [6] already presented a similar approach to this idea. We will reuse and extend some aspects of this approach. The author added these attributes to each component, class and communication of a landscape. The status attributes are ADDED, DELETED and ORIGINAL. A fourth attribute is used in the original approach but it is not relevant for us.

To explain the meaning of each status, let us assume that we have two landscapes that represent two versions of an software architecture: version one and version two.

An entity is

- ▻ ADDED, if it exists in version two but not in version one.
- ▻ DELETED, if it exists in version one but not in version two.
- ▻ ORIGINAL, if it exists in both versions.

The merged landscape therefore show the differences that occurred from version one to two. To determine if an entity exists in a version we use the *fully qualified name* for the components, classes and communications and the name for applications, i.e. we compare these strings to know if an entity changed between versions. The reasoning given for this by Wegert [6] is that names of classes and components are not necessarily unique but paired with the name of their package, which the fully qualified names provides, they can be expected by naming convention to be unique. We also assume that applications with the same name refer to the same application as the merging only considers each unique application name once.

The algorithm consists of iterating over the entities we want to compare and checking if they exist in the other versions. We then set the status accordingly.

3.2.2 Calculating the History Between Multiple Landscapes

The history requires the consideration of more than two landscapes. We need to calculate the changes that occurred between each landscape in the list. For this we use a modified version of Wegert [6]'s algorithm multiple times. First we compare the first and second landscape then the second and third and so on. This will allow us to find every change between the versions. Each change is saved with the corresponding name of the current application and the component, class or communication and the timestamp of the version the change occurred.

3.3 Frontend Extension

The frontend is responsible for showing the information the backend calculated, namely the merged landscape and the history of every entity in an application.

First the user needs to select which versions they want to compare. For this the user interface provides a timeline similar to the one from the visualization of ExplorViz. The timeline is populated by versions the user uploaded before. This upload is presented when the user presses the upload button.

In this timeline the user can select two versions: one to start the comparison and one to end it. The frontend will then show the merged landscape between those versions. This landscape will show all differences between those versions by coloring each status attribute in a different color, e.g. yellow for deleted entities, blue for added ones and green for original ones. These colors can be filtered to only show certain attributes.

If the user selects any application, component or class a history of these entities will be shown. This history consists of all changes which occurred from the start version to the end version. Each child of the entity which were added or removed in those versions will be shown along with the version in which these changes occurred.

Chapter 4

Implementation

In this chapter we present the actual implementation inside the ExplorViz framework. We start with an overview about our implementation and then go into further detail with the backend and frontend of our extension.

4.1 Overview

First we will give a general overview of the implementation and how we extend ExplorViz. Figure 4.1 shows the relevant components of ExplorViz and our extension. The components *ExplorViz Backend* and *ExplorViz Frontend* are the core components of ExplorViz. To add the desired functionality we need to extend them. This is done via the components *Comparison Extension Backend*[1] and *Comparison Extension Frontend* [2].

Both the core backend and the extension backend provide a REST resource the frontend can use. These are used with a reverse proxy to allow the frontend to access all functionalities under one address. We need to slightly modify the configuration for the reverse proxy to allow it to forward the resources of our backend extension.

4.2 Backend Extension

The backend is implemented as its own server in accordance to the microservice architecture. We start by extending the *explorviz-backend-extension-dummy* extension[3] provided by ExplorViz. This extension provides basic functionalities like setting up the server and a service for serializing and deserializing landscapes.

We need to implement the functionalities outlined in our approach in Section 3.2. These are presented to the frontend as resources i.e. URL endpoints. We will describe those resources and the services and models they use in the following two sections.

[1]`https://github.com/ExplorViz/explorviz-frontend-extension-comparison`, Branch: dte
[2]`https://github.com/ExplorViz/explorviz-backend-extension-comparison`, Branch dte
[3]`https://github.com/ExplorViz/explorviz-backend-extension-dummy`

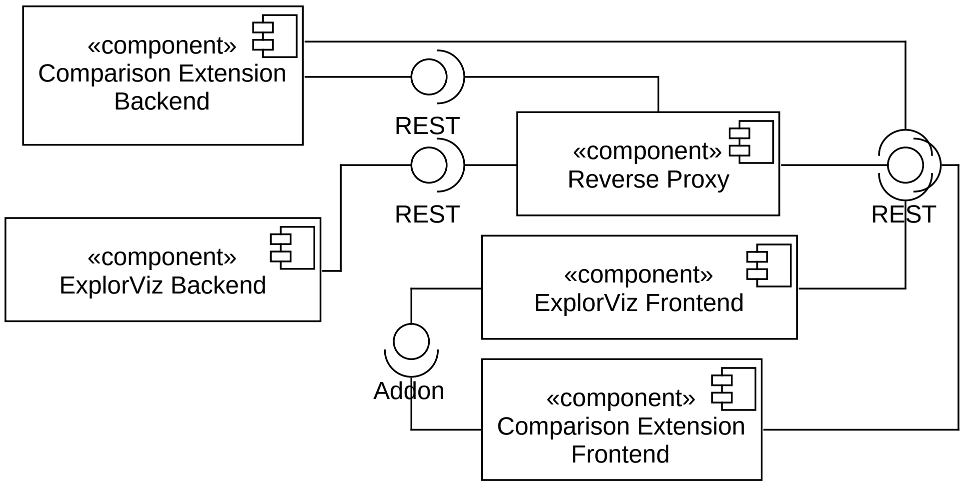


Figure 4.1. ExplorViz component diagram

4.2.1 Comparison of Two Landscapes

This functionality is provided by a a resource called *ComparisonResource*. It can be seen in Listing 4.1.

Listing 4.1. ComparisonResource

```java
@Path(value = "merged-landscapes")
public class ComparisonResource {

   @Inject
   private MergeService mergeService;

   @Inject
   private LandscapeRetrievalService landscapeRetrievalService;

   @GET
   public Landscape getMergedLandscape(@QueryParam("timestamp1") final long
       timestamp1,
         @QueryParam("timestamp2") final long timestamp2) throws IOException,
             DocumentSerializationException {

      return mergeService.mergeLandscapes(landscapeRetrievalService.
          retrieveLandscapeByTimestamp(timestamp1),
            landscapeRetrievalService.retrieveLandscapeByTimestamp(timestamp2));
    }
}
```

The resource gets two landscape timestamps as its arguments. Those are the timestamps of the landscapes the frontend wants to compare. These timestamps are a property of ExplorViz landscapes and represent the time of creation in UNIX time. Since ExplorViz only produces one landscape at a time, those timestamps can be assumed to be unique and are therefore used as an identifier for landscapes.

The timestamps now need to be transformed into their respective landscape objects, so we need to retrieve the saved landscapes from ExplorViz. This is done by the *landscapeRetrievalService*. Listing 4.2 shows the method that actually retrieves the landscape from the service. First a URL connection to the *HistoryService* of ExplorViz is established. This service is responsible for storing landscapes for ExplorViz and offers a URL endpoint to retrieve a single landscape by its timestamp. After we get the landscape from the service we deserialize it, to work with the actual object.

Listing 4.2. Method for landscape retrieval

```
public Landscape retrieveLandscapeByTimestamp(long timestamp) throws IOException,
     DocumentSerializationException {
   final HttpURLConnection landscapeConnection = (HttpURLConnection) new URL(
       urlPath + "?timestamp=" + timestamp).openConnection();

   landscapeConnection.setDoInput(true);
   landscapeConnection.setRequestMethod("GET");
   landscapeConnection.setRequestProperty("Content-Type", "application/json;
       charset=utf-8");
   landscapeConnection.setRequestProperty("Authorization", "Bearer " + token);

   if(landscapeConnection.getResponseCode() == 403) {
      getToken();
      return retrieveLandscapeByTimestamp(timestamp);
   }

   final InputStream inputStream = landscapeConnection.getInputStream();
   String jsonString = IOUtils.toString(inputStream, StandardCharsets.UTF_8);
   inputStream.close();

   return landscapeSerializationHelper.deserialize(jsonString);
}
```

These retrieved landscapes are now passed to the *MergeService,* where the actual

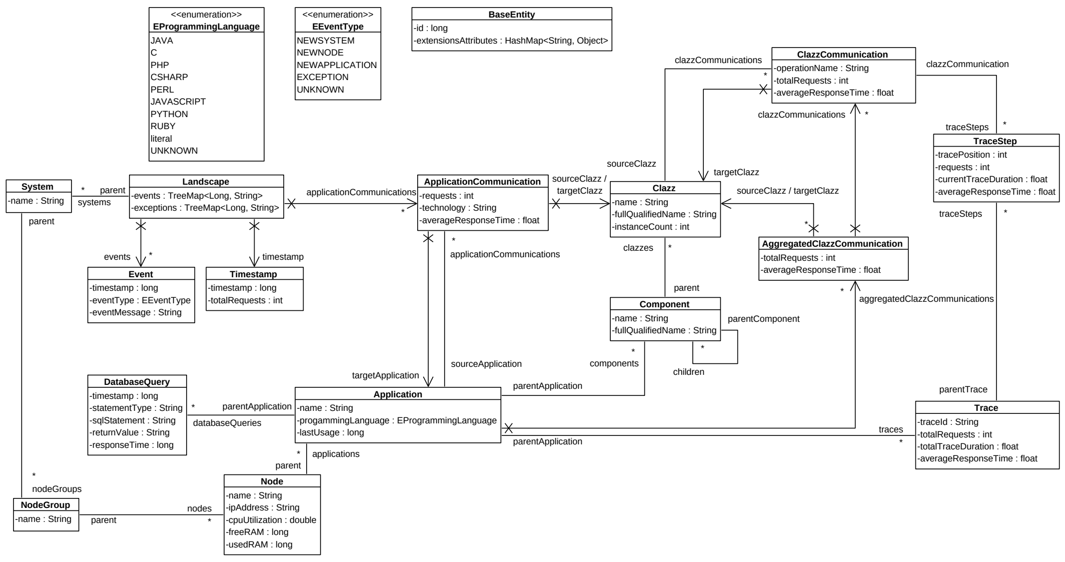


Figure 4.2. ExplorViz data model

calculation takes place. The service will then return the merged landscape that is then passed to the frontend.

To calculate the differences the service employs an algorithm based upon the algorithm described by Wegert [6]. Figure 4.2 shows the data model for ExplorViz landscapes. We can see that the BaseEntity has an attribute “extensionAttributes”. BaseEntity is the parent class of every entity in the model. The “extensionAttributes” allow us to add an attribute to every entity in the landscape. We will use that to mark the differences in the landscape.

To start we need to get every application in the landscapes. This is done by simply iterating over every System, NodeGroup and Node and concatenating the applications contained within. In this process we put every application in a map where the name of the application is the key. This allows for easier comparing. The comparison itself can be seen in Listing 4.3. This is the method that gets called by the resource. As can be seen we simply iterate over both application lists and check if they are present in the other list. Here the purpose of the map is clear. The check if the applications exists in the other list, is done via the “get” method of the map. We use a hash map and therefore on average the check runs in O(1). The status in the extension attributes is then set to the appropriate value. These values are saved in an enum that holds all three possible values. The values are set in the second landscape as we return this landscape as our merged one.

The second iteration shows a problem. Here the application would be marked as DELETED and therefore it exists in landscape one but not in landscape two and we need to add it to landscape two. This is done by getting the Node, NodeGroup and System of the application and checking if they already exist in landscape two. If they do, the application is added to them, otherwise they are added to landscape two and then the application is added.

The components, classes and communications of applications that are either ADDED or DELETED get also marked with the same status. This is done in the *markApplication* method. Here everything inside the application is iterated over and the status is set.

Listing 4.3. Landscape comparison

```java
public Landscape mergeLandscapes(Landscape landscape1, Landscape landscape2) {
      Map<String, Application> applications1 = MergerHelper.getApplicationsFromLandscape(landscape1);
      Map<String, Application> applications2 = MergerHelper.getApplicationsFromLandscape(landscape2);

      for (Map.Entry<String, Application> application2 : applications2.entrySet()) {
         Application application1 = applications1.get(application2.getKey());

         if (application1 != null) {
            application2.getValue().getExtensionAttributes().put(MergerHelper.STATUS, Status.ORIGINAL);
            mergeApplications(application1, application2.getValue());
         } else {
            markApplication(application2.getValue(), Status.ADDED);
         }
      }

      for (Map.Entry<String, Application> application : applications1.entrySet()) {
         if (!applications2.containsKey(application.getKey())) {
            Node node = application.getValue().getParent();
            NodeGroup nodeGroup = node.getParent();
            System system = nodeGroup.getParent();

            int systemIndex = indexOfSystem(landscape2, system.getName());

            if (systemIndex != -1) {
               system = landscape2.getSystems().get(systemIndex);
            } else {
               system.getNodeGroups().clear();
               system.setParent(landscape2);
               landscape2.getSystems().add(system);
            }

            int nodeGroupIndex = indexOfNodeGroup(system, nodeGroup.getName());

            if (nodeGroupIndex != -1) {
               nodeGroup = system.getNodeGroups().get(nodeGroupIndex);
            } else {
               nodeGroup.getNodes().clear();
               nodeGroup.setParent(system);
               system.getNodeGroups().add(nodeGroup);
            }

            int nodeIndex = indexOfNode(nodeGroup, node.getName());

            if (nodeIndex != -1) {
               node = nodeGroup.getNodes().get(nodeIndex);
            } else {
               node.getApplications().clear();
               node.setParent(nodeGroup);
               nodeGroup.getNodes().add(node);
            }

            node.getApplications().add(application.getValue());
            markApplication(application.getValue(), Status.DELETED);
         }
      }

      return landscape2;
   }
```

We call the *mergeApplications* method on every application that was marked as ORIGINAL, meaning both exists in both landscapes and they need to be compared. This method then compares their components, classes and communications. The algorithm is similar to the one used before.

Starting with the components we first need a list of all components of the two applications. Listing 4.4 shows the corresponding method. We start with the components that are referenced directly by the application itself. Then we iterate over every component and add it to the list of components. The same method is then called recursively on all children of the component. So in the end we have a list of components without the hierarchy. We again put the components in a map with their fully qualified name as a key, so we can uniquely identify each component in the application.

Listing 4.4. Component flattening

```java
public static Map<String, Component> flatComponents(List<Component> components) {
      Map<String, Component> flatComponents = new HashMap<>();

      flatComponentsInternal(components, flatComponents);

      return flatComponents;
   }

   private static void flatComponentsInternal(List<Component> components, Map<
       String, Component> flatComponents) {
      for (Component component : components) {
         List<Component> children = component.getChildren();
         flatComponents.put(component.getFullQualifiedName(), component);

         if (!children.isEmpty()) {
            flatComponentsInternal(children, flatComponents);
         }
      }
   }
```

These two lists are then iterated over and marked in the same way as the applications. As before we need to add the components to the second landscape if they were marked as DELETED. This can be seen in Listing 4.5. We first identify the original parent component of the component that we want to add to the second landscape. If such a parent does not exist, we can add to the second landscape directly since it is a top level component. Otherwise we will search for the parent in the second landscape and change the parent of our component to it. If such a parent doesn't exist, we do not need to do anything, since the parent must also be marked as DELETED and therfore will be added in the future. This parent still has the original hierarchy saved and therefore it the component is then added with their parent. We also record all components that were added to the second landscape.

Listing 4.5. Adding a component to the landscape

```java
Component parentIn1 = component.getValue().getParentComponent();

if (parentIn1 == null) {
   application2.getComponents().add(component.getValue());
} else {
   Component parentIn2 = components2.get(parentIn1.getFullQualifiedName());

   if (parentIn2 != null) {
      parentIn2.getChildren().add(component.getValue());
      component.getValue().setParentComponent(parentIn2);
```

```
    }

    addedComponentsTo2.put(component.getKey(), component.getValue());
}

component.getValue().getExtensionAttributes().put(MergerHelper.STATUS, Status.
    DELETED);
```

The classes are also compared in a similar fashion. We again get all classes in the application by simply iterating over the previous component lists and getting all their classes. These classes are then compared in the exact same fashion. Any DELETED classes are just added to their respective components in landscape two. Since this is done after components were added the needed components must exist. We use the previous component list and the recorded added components to find the needed components.

Finally the communications are compared. Here we don't need to create a list of communications, since these are already saved in the applications. So we get the *aggregated-CalzzCommuniactions* of the applications and iterate over them. Then we put them in the same map structure as the others, i.e. the fully qualified name as the key and the actual object as the value. The comparison is then exactly the same. DELETED communications are just added to the list of aggregated communications of the second landscape.

4.2.2 Calculating the History

Listing 4.6 shows the resource that is responsible for providing the history. Here the frontend provides a list of timestamps for the versions it wants the history of. As before we get all landscapes from the history service and then call the *HistoryService* to calculate the actual history. The listing also shows that this history is returned as its own object. This object has three fields, one for the component history, one for the class history and one for the communication history. The component history and class history have the type `Map<String, Map<String, Map<Long, Status>>>`. Here for every application a number of components/classes are saved. Then for every one of those components/classes a number of timestamps and their respective changes are saved. The applications are saved with their name and the components/classes are saved with their fully qualified name. This combination of application name and fully qualified name assigns each component/class that gets added to the history a unique identifier in the landscape.

Listing 4.6. HistoryResource

```
@Path(value = "histories")
public class HistoryResource {

    @Inject
    private HistoryService historyService;
```

```

   @Inject
   private LandscapeRetrievalService landscapeRetrievalService;

   @GET
   public History getMergedLandscape(@QueryParam("timestamps[]") final List<Long>
        timestamps) throws IOException, DocumentSerializationException {
      List<Landscape> landscapes = new ArrayList<>(timestamps.size());

      for(Long timestamp : timestamps) {
         landscapes.add(landscapeRetrievalService.retrieveLandscapeByTimestamp(
             timestamp));
      }

      return historyService.computeHistory(landscapes);
   }
}
```

The communications are saved as a list of *communicationHistory* objects. These objects hold the source and target class names and the application name. Again the fully qualified name is used so every class is uniquely identified.

The actual calculation of the history is done inside the HistoryService and uses the same algorithm as before. The difference is now that we not only compare two landscapes but a list of them. Listing 4.7 shows how this is done. We first compare version one and two then version two and three and so on. This allows us to record every change between the versions.

Listing 4.7. History between multiple landscapes

```java
public History computeHistory(List<Landscape> landscapes) {
      History history = new History();

      for (int i = 0; i < landscapes.size() - 1; i++) {
         compareLandscapes(landscapes.get(i), landscapes.get(i + 1), history);
      }

      return history;
   }
```

Every two landscapes are then compared to each other. This is done exactly like before, only here the status is instead recorded into the history object instead of the landscape itself. Also we only record ADDED or DELETED statuses. Listing 4.8 shows the component comparison as an example. As can be seen if a difference occurs the current timestamp, application name and component name is recorded together with the actual change.

Listing 4.8. History between multiple landscapes

```java
for (Map.Entry<String, Component> component : flatOldComponents.entrySet()) {
   if (!flatNewComponents.containsKey(component.getKey())) {
      history.addHistoryToComponent(applicationName, component.getValue().
          getFullQualifiedName(), timestamp, Status.DELETED);
   }
}

for (Map.Entry<String, Component> component : flatNewComponents.entrySet()) {
   if (!flatOldComponents.containsKey(component.getKey())) {
      history.addHistoryToComponent(applicationName, component.getValue().
          getFullQualifiedName(), timestamp, Status.ADDED);
   }
}
```

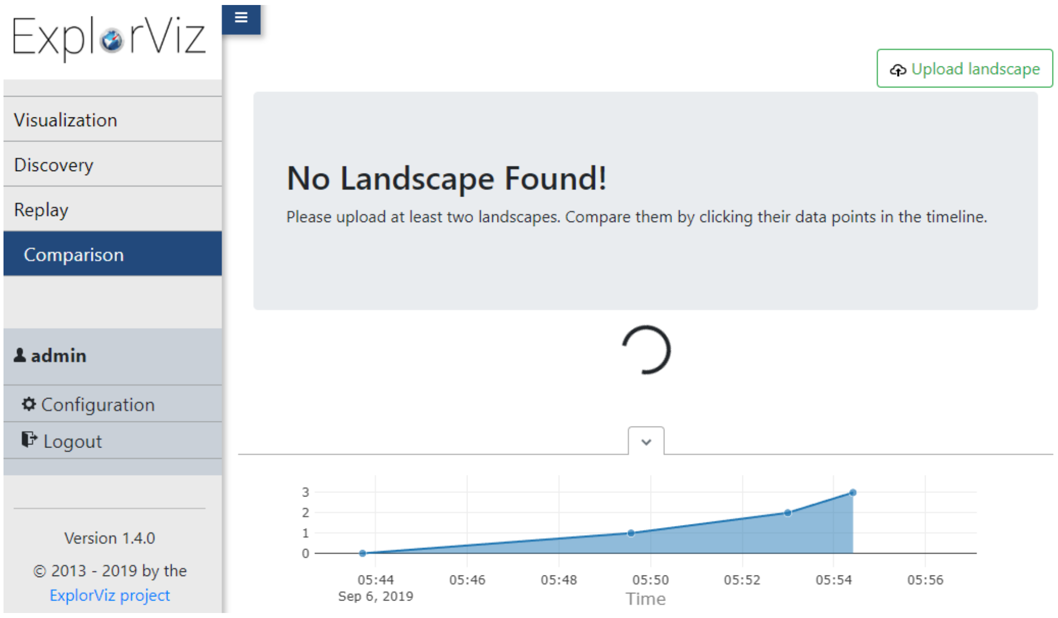
4.3 Frontend Extension

The frontend extension is responsible for presenting the data, that the backend calculated, to the user. This is done via an ember addon that extends the ExplorViz frontend. On the actual ExplorViz site our addon can be accessed by the *comparison* route. This route is added to the navigation bar so the user can navigate to it. Figure 4.3 shows a screenshot of the user interface without a merged landscape. The lower area consists of the timeline in were the user can upload the different version. In the middle area the landscape will be rendered. The button in the top right allows the user to upload new landscapes. We will further present the implementation of the landscape upload, the timeline, the rendering of a merged landscape and the presentation of the history.

4.3.1 Uploading Landscapes

For the upload we also utilize the HistoryService in the backend. This services provides a resource that allows for landscape uploads. These landscapes are JSON files which can be downloaded via the visualization route. This download functionality and a replay route which displays uploaded landscapes are a core feature of the frontend.

Our upload function is mostly a copy of the upload function of the replay route. We make an AJAX request to the backend with the file. Additionally we push the timestamp of the uploaded landscape into the timeline to allow the user to select it. We just push the timestamp into the array that holds them. This is only done if the upload was successful.


No Landscape Found!

Please upload at least two landscapes. Compare them by clicking their data points in the timeline.

Figure 4.3. User interface

We also modified the function so the user can upload multiple landscapes at a time. For this we modify the file dialog so it will allow for multiple file selection and then we iterate over those files and make the request for each of them.

4.3.2 Timeline

The timeline started as a copy of the timeline that ExplorViz uses in it visualization route. We then modify it in two ways: First we change it so the timestamps that get shown in the timeline are saved in our *merged-landscape-repository* service. This service holds the current landscape, application, history and a list of timestamps in the timeline. If the timeline has to be rendered it pulls its timestamps from there. This allows us to be independent of the rest of ExplorViz so we can display our own timestamps. Secondly we remove the request in the y-axis since those are not relevant for our purposes.

If the user selects two timestamps in the timeline we get a subarray from all timestamps with those two timestamps and every timestamp in between and we use this array to make a request to our history and comparison route in the backend. The history resource gets the whole content of the array and the comparison resource the first and last element. The request are handled by the ember store via adapters. If the calls return we push the answers into our merged-landscape-repository and trigger an update of the landscape rendering.

4.3.3 Rendering of a Merged Landscape

As explained in Section 2.2 ember uses a lifecycle for its components to allow developers to manipulate them. ExplorViz uses the *didRender()* hook to render its landscapes. This is done in two components: *landscape-rendering* for the landscape view and *application-rendering* for the application view. To allow us to render our merged landscape we have to extend those components. We do this in the components *merged-landscape-rendering* and *merged-application-rendering* respectively. Both components are then rendered in our comparison route.

Listing 4.9 shows the extended method for the landscape rendering. This method "createPlane" gets called when a new application is added to the scene. In line 12 and following we check for the status of the application and color them accordingly. The actual color comes from the *comparisonConfiguration*, a configuration service that just holds the values of all needed colors in our expansion. More importantly we override the landscapeRepo variable with our own *merged-landscape-repository* service, so we render our own landscape.

Listing 4.9. Landscape Rendering

```
 1 export default class MergedLandscapeRendering extends LandscapeRendering {
 2    @service('merged-landscape-repository')
 3    landscapeRepo!: MergedLandscapeRepository;
 4
 5    @service('comparison-configuration')
 6    comparisonConfiguration!: ComparisonConfiguration;
 7
 8    createPlane(model) {
 9       const emberModelName = model.constructor.modelName;
10       let color;
11       const applicationStatus = model.get('extensionAttributes.status');
12
13       if(emberModelName == 'application') {
14          if(applicationStatus == 'ADDED') {
15             color = this.get('comparisonConfiguration.mergedLandscapeColors.
                    addedApplication');
16          } else if(applicationStatus == 'DELETED') {
17             color = this.get('comparisonConfiguration.mergedLandscapeColors.
                    deletedApplication');
18          } else if(applicationStatus == 'ORIGINAL') {
19             color = this.get('comparisonConfiguration.mergedLandscapeColors.
                    originalApplication');
20          }
21       } else {
        color = this.get('configuration.landscapeColors.' + emberModelName);
      }

    const material = new THREE.MeshBasicMaterial({
      color: color
    });
    ...
}
```

The application view is implemented in a similar fashion. Listing 4.10 for example, shows the coloring of the classes. Again we check the attributes and set the color to the corresponding value. Here the filtering of entities is also visible. This allows the user to gray out entities with a specific status. These filters are just booleans inside the configuration service. The user can toggle them via buttons in the navigation bar. We simply check if the relevant flag is set and otherwise set the color to the deselected value.

Listing 4.10. Class Rendering

```
const clazzAttribute = clazz.get('extensionAttributes.status');
          let clazzColor;

          if(clazzAttribute == 'ADDED' && this.get('comparisonConfiguration.
              comparisonToggle.added')) {
            clazzColor = this.get('comparisonConfiguration.mergedApplicationColors
                .addedClazz');
          } else if (clazzAttribute == 'DELETED' && this.get('
              comparisonConfiguration.comparisonToggle.deleted')) {
            clazzColor = this.get('comparisonConfiguration.mergedApplicationColors
                .deletedClazz');
          } else if (statusAttribute == 'ORIGINAL' && this.get('
              comparisonConfiguration.comparisonToggle.original')) {
            clazzColor = this.get('configuration.applicationColors.clazz');
          } else {
            clazzColor = this.get('comparisonConfiguration.mergedApplicationColors
                .deselectedClazz');;
          }

          this.createBox(clazz, clazzColor, true);
```

4.3.4 Displaying the history

We need to present the calculated history to the user. As described in we want to display the history for a selected entity in the application view. To allow this, we use the *highlighter* service of ExplorViz. It allows us to easily find the selected entity in the current application. Our history is then presented as a table split into three parts: the components, classes and communications.

Listing 4.11 shows the implementation of the component history as an example. First we have to determine what entity is currently selected. We then check if the selected entity is a component. The actual history for the currently loaded landscape is also saved in the merged-landscape-repository and is loaded here. We also get all children of the component and their children. The highlighted component is also added to this list. This allows us display the history for every component that is hierarchically below the highlighted component. We then iterate over those components and check if the component name is in the history. If such an entry exists it is added to the return value of the function. This is then displayed by the template of the component.

Listing 4.11. Component History

```
get componentHistory() {
   const highlightedEntity = this.get('highlighter.highlightedEntity');

      if(highlightedEntity instanceof Component) {
            const components = highlightedEntity.getAllComponents();
            components.push(highlightedEntity);
      const applicationName = this.get('landscapeRepo.latestApplication.name');
            const histories = [];
            const latestHistory = this.get('landscapeRepo.latestHistory.
                componentHistory.' + applicationName);
            const self = this;

            components.forEach((component) => {
            const historyEntry = latestHistory[component.get('fullQualifiedName')
                ];

            if(historyEntry) {
                  histories.push({name: component.get('name'), historyEntry: self.
                      convertHistoryEntry(historyEntry)});
            }
         });

      return histories;
    }

    return null;
  }
```

The function for the classes is similar. The class function distinguishes between a selected class and a selected component. The component displays all histories of classes it contains, if any exist, and, if a class is selected, only the history for this single class is shown.

The communication is displayed similarly. Here a difference between classes and communications is made. For a class, every communication it has, is shown. A selected communication only shows the history of itself.

Chapter 5

Evaluation

In this chapter, we will present the evaluation of our approach and implementation. First we will describe the overall goal of our evaluation in Section 5.1, followed by the methods used for the evaluation in Section 5.2. Then we will present the conducted usability experiment in Section 5.3 and finally summarize the evaluation in Section 5.4.

5.1 Goals

Our main goals are the evaluation of the functionality and usability of our software. A user should be able to effortlessly and intuitively use our tool to compare two versions and look at the history between them. We want the participants to interact with our tool and test if they can employ it to easily compare software versions. Furthermore we want to identify any potential errors, bugs or bad user interface decisions in our software.

5.2 Methodology

Here we describe the methods used in our evaluation. We conduct a survey based on a questionnaire, which we will describe in Section 5.3.1. The questionnaire includes a series of tasks, which the participants are asked to complete. During this time we also note how well the participants navigate through the tool and what errors they make. Additionally we note the completion time of the tasks.

5.3 Usability Experiment

Here we describe the actual experiment we conducted. We first start by describing the questionnaire followed by the soft- and hardware setup for the experiment. After that the actual course of the experiment is outlined. Then we present the results and our discussion of them. Finally, we discuss threats to the validity of our experiment.

5.3.1 Questionnaire

We use an online questionnaire[1] for our evaluation. It is divided into three sections. These are presented below. The complete questionnaire can be found in Section A.2.

Personal Information

First we ask the participants for personal information, especially regarding their previous knowledge. The first question asks for the job title or, if they are a student, their subject. Then we want to know how many years they have worked in that job or in which semester they currently are in and what degree they are pursuing. This allows us to assess how knowledgeable our participants are in the general field of computer science. The final three questions regard their previous knowledge in the following topics:

- ExplorViz
- Software Visualization Tools
- Java Programming

The participants are asked to rate those experiences on a four point scale (None-Beginner-Intermediate-Expert). The perceived difficulty of the following tasks might correlate with the previous knowledge of the participants, especially with the ExplorViz tool. Knowledge in other tools of that kind might also influence how easy they can work with ExplorViz. Finally, since our landscapes are based on a Java program, knowledge here might help understanding the parallels between ExplorViz landscapes and the architecture of a Java program. This might allow the participants to intuitively grasp the different aspects of an ExplorViz landscape.

Tasks for Comparing Landscapes

The second section of the questionnaire consists of a few tasks the participants are asked to solve. The purpose of those tasks is for the participants to interact with the tool and familiarize themselves with it.

We start with a short text asking the participants to upload four prepared landscapes. Then the next six questions task the participant with finding an added/deleted component (question 6/7), class (question 8/9) and communication (question 10/11). These tasks are mostly used so the participants navigate through our tool and to measure how intuitive features like the filter are.

Question 12 is intended to test how intuitive the changes of the applications in the landscape view are. The participants are asked to consider a particular application(`remoteSampleApplicationClient`) and to write down what happened to it.

[1] `https://docs.google.com/forms`

Questions 13 and 14 regard the history feature. We ask what changes occurred in two entities and in which version those changes occurred. These two entities are the added communication in the `sampleApplication` application (question 13) and the `sampleClient` class in the `remoteSampleApplication` application (question 14). These questions will allow us to asses how usable the history feature is and if it presents the data in a readable fashion.

Debriefing Questions

Lastly, we ask four questions about the usability of the tool. The participants answer those questions on a five point Likert scale [5]. The questions were in order:

15. How easy did you find the tasks?

16. How easy was it to use the tool?

17. How easy was it to navigate through the site?

18. How useful did you find the history feature?

Questions 15 to 17 had to be rated from very easy to very hard and question 18 from not useful to very useful. Finally we asked for any additional feedback towards our tool in question 19. These questions allow us to assess how usable our tool is and how much functionality was added by our history feature.

5.3.2 Experimental Set-Up

Tutorial Set-Up

We use the tutorial extension from ExplorViz[2] to demonstrate the basic functionality of ExplorViz to the participants. For this we prepare a tutorial using one of the landscapes described below. This tutorial consists of a series of steps starting with the landscape view. Then the participant is taught what applications are and asked to open a specific one. In the application view the same is done for components. To demonstrate classes they now have to open more components until they can see some classes of those components. Next the ability to highlight different entities is demonstrated and they have to highlight a specific component to progress. Finally communications are explained and the participant is required to highlight a specific one to complete the tutorial.

[2]`https://github.com/ExplorViz/explorviz-frontend-extension-tutorial` and `https://github.com/ExplorViz/explorviz-backend-extension-tutorial`

Table 5.1. Hardware specifications

CPU	Intel Core i5-6500 4x 3.20 Ghz
RAM	16 GB
Display Size	24 inch
Display Resolution	1920 x 1200 pixels

Generating Landscapes for the Experiment

We generate four landscapes for our experiment by monitoring the "kiekerSampleApplication" provided by ExplorViz. To allow the comparison we use two different versions in the GitHub repository [3].

We start by monitoring the later version by executing the provided script and starting the ExplorViz backend and frontend. Then we download the monitored landscape through the visualization route.

The same thing is then repeated for the earlier version to generate a second landscape with a later timestamp. To generate a history we repeat this process now again with the later and then the earlier version again. We save those landscape files to later use them in the experiment.

Configuration

The experiment is conducted using two machines. One is used by the participants for the tasks and the other allows them to fill out the questionnaire. The first machine is a desktop computer. Its specifications can be seen in Table 5.1.

The operating system is Windows 10 Professional 64-bit and we use the Java Development Kit Version 12.0.2. The ExplorViz version is 1.4.0.

We prepare the machine by running the ExplorViz frontend and backend together with our backend and frontend extension. The backend is started in Eclipse via the provided run configurations and the frontend in the Windows powershell with "ember serve". Additionally we install the ExplorViz tutorial extension and upload the previously prepared tutorial. The prepared landscapes are placed on the desktop.

The second machine is a laptop that is placed next to the first one, so the participants can easily reach it. We open the questionnaire on it so the participants can fill it in.

5.3.3 Execution of the Experiment

We start by letting the participants run through the tutorial. Then we explain the overall motivation behind our extension and why we developed it. Further we explain possible scenarios in which such a tool might be useful.

[3]`https://github.com/ExplorViz/sampleApplication/commit/3205bb90e3f40d0fcf40624fc43f21b6867699f5` and `https://github.com/ExplorViz/sampleApplication/commit/809c0d9e4570252948e40c4f4f69d5ac8a862f79`

Table 5.2. Participant experiences

Skill	None	Beginner	Intermediate	Expert
ExplorViz	3	3	1	1
Software Visualization Tools	4	3	1	0
Java Programming	0	5	2	1

Then we ask them to fill out the personal information section of the questionnaire followed by the tasks. After they upload the landscapes, we point out the functionalities on the navigation bar, namely the history and the filter. When they complete the tasks, the experiment ends with the debriefing questions. We take the time time at the beginning and end of the tasks. The whole experiment procedure can be found in Section A.1.

5.3.4 Results

Here we present the results of our experiment and our questionnaire, starting with the personal information, followed by the comparing tasks. We then close with the debriefing questions.

Personal Information

Our study consisted of eight participants, six of them bachelor computer science students, one master student and one former bachelor computer science student. The bachelor semester ranged from the 3rd to the 8th semester and the master student was in the 2nd semester. Their experiences in the described subjects is shown in Table 5.2.

Tasks for Comparing Landscapes

Here we present the results of the comparison tasks. In Table 5.3 the percentage of participants who solved the tasks correctly can be seen. The raw data is shown in Section A.3.

We also measured the time the participants took to complete the tasks. The mean time was 12.6 minutes.

Debriefing Questions

This section was designed to gather feedback. The mean results are presented in Table 5.4.

Additionally we asked for any additional feedback on our tool in question 5. These answers can be seen in the raw data in Section A.3.

Table 5.3. Correct tasks

Question	Correct Answers	Incorrect Answers	Percentage of correct answers
6	7	1	87.5%
7	7	1	87.5%
8	6	2	75%
9	8	0	100%
10	7	1	87.5%
11	6	2	75%
12	8	0	100%
13	8	0	100%
14	8	0	100%

Table 5.4. Debriefing Results

Question	Mean
15	2.5
16	2.6
17	1.7
18	4.0

5.3.5 Discussion of Results

In this section we will discuss the previously described results again divided into the three sections of the questionnaire.

Personal Information

All participants were computer science students, therefore it is not surprising that all reported at least a beginner level of experience in Java, since they probably all gained experience during their bachelor or master course. We can also see that the experience in ExplorViz and software visualization tools in general seem to correlate, i.e. if a participant reported no experience in ExplorViz they also didn't report any or not very much experience in software visualization tools. There is also a clear divide between participants that prevoiusly worked with ExplorViz and those who didn't. Only the two participants that worked previously with ExplorViz reported their experience as intermediate or expert.

Tasks for Comparing Landscapes

Here we wanted the participants to work with our tool and measure how easy and intuitive it is. On problem that some participants faced was the timeline. We had to explain how to start the comparison of landscapes and what the datapoints represented. Here we should try to explain this functionality better inside the tool.

As can be seen in Table 5.3 all tasks had at most one or two incorrect answers. We observed during the experiment that most of those errors were due to misunderstandings what a component, class or communication is in ExlorViz. This is further supported by the fact that those mistakes only occurred with participants who had no former experience with ExplorViz. A better explanation or tutorial would help us here to further improve our tool.

Another error was that participants misinterpreted the color of the original classes as added classes. When the filter was used, this mistake did not happen. Therefore we need to change these colors to better reflect the status of the classes and resolve this ambiguity.

Especially the tasks related to the history feature (questions 12, 13 and 14) have a very high correctness, which indicates that the usage is easy and not very error prone. Although on participant started to compare multiple different landscapes and tried to find out which timestamp correlates to which version in the timeline. This association should be made automatically by the tool.

Some participants also used the history to identify the asked for changes and used the wrong version. We need to better represent which timestamp corresponds to which version in the timeline. Further we need to better communicate that the changes in the landscape are actually between the selected datapoints.

The mean time for the tasks was relatively short, which indicates that the tool is somewhat fast to use.

Debriefing Questions

We wanted to know if the tool was easy to use and intuitive. For the first three questions a lower value means that the participants rated the tasks, tool usage and navigation as easier. As can be seen in Table 5.4 the participants thought the tasks were easy to solve, the tool easy to use and it was easy to navigate. This suggests that, even without previous knowledge about ExplorViz, our tool was intuitive to use.

The fourth question asked for the usefulness of the history feature. Here a higher value means that the participants rated the feature as more useful. The result in Table 5.4 shows that the overall rating of the usefulness of this feature was quite high. This indicates that this function is a useful addition to our tool.

The last question asked for any additional feedback. One common complaint was that the colors for the different statuses are not intuitive and should be changed. Also some colors (e.g. the yellow for the status deleted) make it hard to read the white text of the labels. Here we might need use different colors or offer a customization option for those.

One participant suggested that the versions in the timeline should also be able to be loaded in in non timestamp order. This would definitely enhance the functionality since it wouldn't rely on the landscape timestamps to order them in the timeline. The user could potentially order them in any order they want.

Another suggestion was to make the history feature more prominent by, for example, opening it up, if an entity got selected, or highlight the respective button better. This would also make it easier to use the tool without an introduction as the one in the experiment.

The format of the timestamps was another issue. Here a better readable format would be sufficient.

Finally a bug in the search feature was revealed. It does simply not work. This needs to be fixed.

5.3.6 Threats to Validity

Our study only included eight participants. Statistically meaningful results are not really possible with this low number. Another study with more participants would help solve this problem and further validate our results.

Another threat are the chosen tasks. They might not fully represent a typical workflow with the tool. Related to this are the chosen landscapes as those were quite arbitrary. We might need to construct a better real life example for this.

Additionally, the experience level differences between participants, especially regarding ExplorViz, was quite high. We had participants with no former experience and participants with a lot of experience. This might skew our results as the less experienced participants might have had more problems with ExplorViz overall instead of our tool specifically.

5.4 Summary

To summarize, overall we received positive feedback from our participants which was also represented in our debriefing questions. The tool seems to be easy to use after a short introduction with still a few user interface problems. Our tasks were solved correctly which suggests that the functionality of our tool is given and the intended problems can be solved with it. The questionnaire and the raw results can be found in Appendix A. Additionally they can be found on Zenodo[4] together with all needed artifacts to replicate the experiment.

[4] `https://doi.org/10.5281/zenodo.3465317`

Chapter 6

Related Work

In this chapter we will present related work for this topic. We start with an ExplorViz extension by Wegert [6] and follow with another extension by Hackel [4].

The first extension was developed by Wegert [6]. Here the goal was to compare two software versions with the help of ExplorViz. To accomplish this a new extension for front- and backend was developed. This extension was created before ExplorViz shifted to microservice architecture [7, 9] and therefore it does not work with the current ExplorViz version. The extension allows for the comparison of two landscapes. In this two landscapes all applications are compared and the changes are merged into a single landscape. If an application does not exist in both landscapes no comparison is done. The differences between the landscapes are described with four statues: ADDED, DELETED, ORIGINAL, EDITED. ADDED means an entity was added, DELETED it was deleted and ORIGINAL that no change occurred. EDITED means that an entity contains an ADDED or DELETED entity. These changes are visualized in the frontend by coloring the merged landscape according the the statuses.

Another extension for ExplorViz was presented by Hackel [4]. Here the the goal was to test if a software architecture conforms to a model. To construct this model an extension called "Modeleditor" for the front- and backend is provided. This extension allows the user to create a landscape in the frontend by adding the entities. This can be done for every landscape in the data model. With such a model a "real" landscape can be compared with it. For this the extension "ArchConfCheck" is provided. It compares a landscape with a model and provides a new landscape where the entities are marked with three statuses. The statues are ASMODELLED if the entity exists in the landscape and in the model, WARNING if it exists in the landscape but not in in the model and GHOST if an entity exists in the model but not in the landscape. These status are then visualized as colors in the frontend, i.e. every status is colored differently.

Chapter 7

Conclusions and Future Work

In this chapter we will summarize our thesis and draw a conclusion. Furthermore we will discuss any future work on our approach and implementation.

7.1 Conclusions

This thesis had the goal of developing an extension for ExplorViz to allow it to compare software architectures. We first described our approach and implementation for this extension. Finally we conducted an evaluation to gather information about the functionality and usability of our implemented software.

Our software was developed as an extension to ExplorViz and consisted of a backend and a frontend. Our backend is responsible for comparing and merging two landscapes and calculating the history between a list of landscapes. We record those differences with the three status ADDED, DELETED and ORIGINAL and added them directly to the landscape and a our own history model. This data was presented to the frontend which was then responsible for displaying that data to the user. The differences in the merged landscape were visualized with the colors of the entities and the history showed every difference between the versions for the selected entity in a table format.

The evaluation was conducted as a usability experiment. We designed a questionnaire and performed the experiment with multiple participants. Our tasks were easily solved and with a high correctness and we conclude that our extension is usable and functional. Although we also identified some problems with the user interface and our experiment might not be statistically relevant due to the low participant number.

Our extension allows a developer to compare two software versions visually, so changes in the architecture can be easily identified. It also provides the ability to get a history of every entity in an application that comprises every change that occurred in the versions between the compared landscapes.

7.2 Future Work

Our software can be extended with several features in the future. We will describe some ideas that came up during the development process and suggestions from the evaluation.

First there is the issue of persisting the calculated landscapes and history. Currently if the user closes the comparison site in the frontend everything that was calculated is lost. Especially large landscapes or a lot of versions between them can take a while to calculate. It would be useful if this is only needed one time and then everything could be saved. This would be accomplished by adding a database to the backend and saving the merged landscape and history to it if they are requested by the frontend. We would need to assign them some sort of unique id to allows us to reference them back. The frontend could then request those ids from the database. A file based approach would also be possible. Here we would allow the user to up- and download previously generated landscapes and histories. These would probably saved in the JSON format as are the currently downloadable landscapes. We could also present previously saved landscapes and histories to the user so they can easily select them.

Another possible feature would be the customization of the colors in the merged landscapes. ExplorViz already offers a colorpicker extension for its normal landscapes. We could use this extension to implement a color choice for our landscapes. This would allow for example users with colorblindness to choose colors that are better visible for them. A number of presets would also possible.

Another possible feature would be that the versions in the timeline can be ordered in non timeline order. Currently the versions are ordered automatically in the order of the timestamps of the landscapes. We would need a way to allow the user to order the datapoints in the timeline perhaps via drag and drop.

Finally we could display more information within the merged landscape and the history. Currently only changes to the architecture are calculated and for the communications we only consider the aggregated communications, i.e. all method calls between two classes. If for example a communication would get another method call our current implementation would not detect that. We would need to compare the communications separately and display them to the user. For example we could show in the history, if a user selects a communication, every method call that changed in this communications in a similar fashion as it is currently shown. Other useful information that could be compared is for example the instance count of classes or applications to observe the load of the system. The same is true for system or node metrics which would also be useful for performance analysis. Also the communications between applications in the landscape view are currently not considered. This would also be a useful addition.

Bibliography

[1] F. Fittkau, J. Waller, C. Wulf, and W. Hasselbring. 2013. Live trace visualization for comprehending large software landscapes: the explorviz approach. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*. (September 2013), 1–4 (cited on page 3).

[2] Florian Fittkau, Alexander Krause, and Wilhelm Hasselbring. 2017. Software landscape and application visualization for system comprehension with explorviz. *Information and Software Technology*, 87, (July 2017), 259–277. `http://eprints.uni-kiel.de/33464/` (cited on page 3).

[3] Florian Fittkau, Sascha Roth, and Wilhelm Hasselbring. 2015. Explorviz: visual runtime behavior analysis of enterprise application landscapes. In AIS (cited on page 3).

[4] Tim Hackel. 2018. *Architekturkonformitätsüberprüfung von softwarelandschaften mittels explorviz*. Diplomarbeit. Kiel University, (September 2018). `http://eprints.uni-kiel.de/44272/` (cited on page 35).

[5] Rensis Likert. 1932. A technique for the measurement of attitudes. *Archives of psychology* (cited on page 29).

[6] Josefine Wegert. 2018. *Visualizing software architecture comparison of a web-based financial application in explorviz*. Masterarbeit. Kiel University, (Mai 2018). `http://eprints.uni-kiel.de/43099/` (cited on pages 8, 9, 14, 35).

[7] Christian Zirkelbach, Alexander Krause, and Wilhelm Hasselbring. 2019. Modularization of research software for collaborative open source development. In *The Ninth International Conference on Advanced Collaborative Networks, Systems and Applications (COLLA 2019)*. (June 2019), 1–7. `http://eprints.uni-kiel.de/46777/` (cited on pages 3, 7, 35).

[8] Christian Zirkelbach, Alexander Krause, and Wilhelm Hasselbring. 2018. On the modernization of explorviz towards a microservice architecture. In CEUR Workshop Proceedings (cited on page 3).

[9] Christian Zirkelbach, Alexander Krause, and Wilhelm Hasselbring. 2019. On the modularization of explorviz towards collaborative open source development. Forschungsbericht. Kiel University, (April 2019). `http://eprints.uni-kiel.de/46829/` (cited on pages 3, 7, 35).

Appendix A

Usability Experiment

A.1 Experiment Procedure

- ▻ present the tutorial and let the participant solve it
- ▻ explain the motivation behind the extension:
  - ▻ it is useful to compare software versions, e.g. for bug fixing
  - ▻ this can be very complicated
  - ▻ code comparison is possible but quite cumbersome
  - ▻ a visual approach works better
  - ▻ the extension automates this process and creates a version history
- ▻ present them the questionnaire and let them fill in the personal questions
- ▻ they start the tasks
- ▻ after uploading the landscapes, show them the history and the filter function
- ▻ let them fill in the debriefing questions

A.2 Questionnaire

Evaluation Bachelorarbeit

Personal Information

1. **What is your job title? If you are a student what is your subject?**

   ____

2. **How many years did you have this job or if you are a student, what is your current semester and what degree you are pursuing?**

   ____

Experience

Please rate your experience with the following:

3. **ExplorViz**
   Markieren Sie nur ein Oval.

   ◯ None
   ◯ Beginner
   ◯ Intermediate
   ◯ Expert

4. **Software Visualization Tools**
   Markieren Sie nur ein Oval.

   ◯ None
   ◯ Beginner
   ◯ Intermediate
   ◯ Expert

5. **Java Programming**
   Markieren Sie nur ein Oval.

   ◯ None
   ◯ Beginner
   ◯ Intermediate
   ◯ Expert

Comparison Tasks

Next you will try to solve some tasks using ExplorViz.

Please navigate to the comparison site and upload the four landscapes, which can be found on the desktop in the folder "landscapes". The history of a component can be opened by highlighting it and opening the history from the navigation bar.

Please compare the first and the final version. Try to find a example for the following. Note the name and the application it is in. If you think it doesn't exist just leave the field blank.

6. **An added component**

7. **A deleted component**

8. **An added class**

9. **A deleted class**

10. **An added Communication**

11. **A deleted Communication**

12. **What happend to the "remoteSampleApplicationClient"?**

13. **Consider the added Communication in the sampleApplication. When was this communication added? In which version?**

14. **What changed in the sampleClient Class in the remoteSampleApplicationClient in the different versions? What change occured when?**

Usability questions

15. **How easy did you find the tasks?**
Markieren Sie nur ein Oval.

	1	2	3	4	5	
very easy	⬭	⬭	⬭	⬭	⬭	very hard

16. **How easy was it to use the tool?**
Markieren Sie nur ein Oval.

	1	2	3	4	5	
very easy	⬭	⬭	⬭	⬭	⬭	very hard

17. **How easy was it to navigate through the site?**
Markieren Sie nur ein Oval.

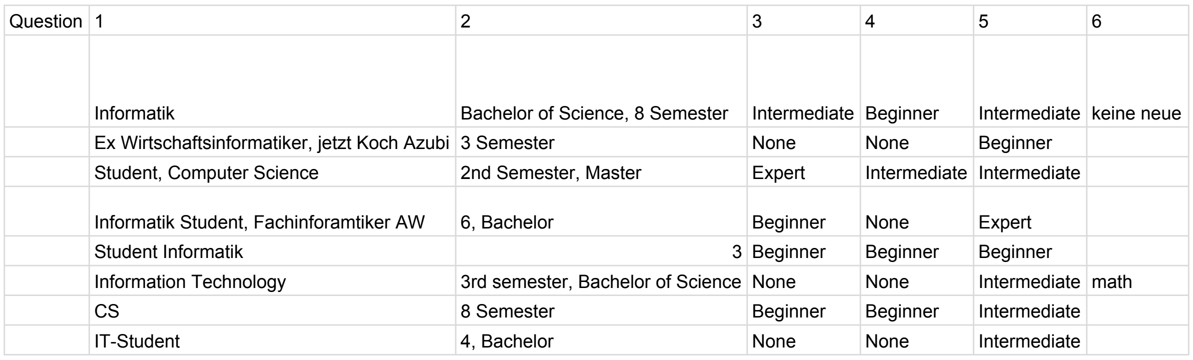
	1	2	3	4	5	
very easy	⬭	⬭	⬭	⬭	⬭	very heard

18. **How useful did you find the history feature?**
Markieren Sie nur ein Oval.

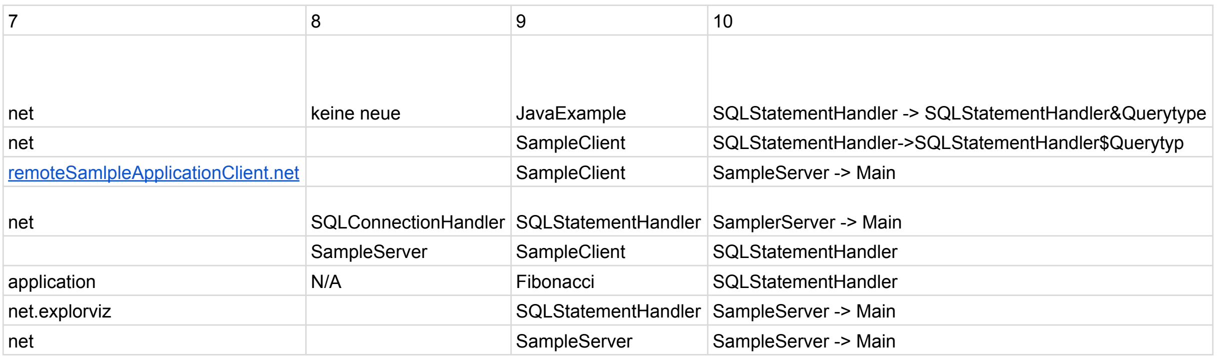
	1	2	3	4	5	
not useful	⬭	⬭	⬭	⬭	⬭	very useful

19. **Do you have any additional feedback?**

A.3 Raw Data



Question	1	2	3	4	5	6
	Informatik	Bachelor of Science, 8 Semester	Intermediate	Beginner	Intermediate	keine neue
	Ex Wirtschaftsinformatiker, jetzt Koch Azubi	3 Semester	None	None	Beginner	
	Student, Computer Science	2nd Semester, Master	Expert	Intermediate	Intermediate	
	Informatik Student, Fachinforamtiker AW	6, Bachelor	Beginner	None	Expert	
	Student Informatik	3	Beginner	Beginner	Beginner	
	Information Technology	3rd semester, Bachelor of Science	None	None	Intermediate	math
	CS	8 Semester	Beginner	Beginner	Intermediate	
	IT-Student	4, Bachelor	None	None	Intermediate	



7	8	9	10
net	keine neue	JavaExample	SQLStatementHandler -> SQLStatementHandler&Querytype
net		SampleClient	SQLStatementHandler->SQLStatementHandler$Querytyp
remoteSamlpleApplicationClient.net		SampleClient	SampleServer -> Main
net	SQLConnectionHandler	SQLStatementHandler	SamplerServer -> Main
	SampleServer	SampleClient	SQLStatementHandler
application	N/A	Fibonacci	SQLStatementHandler
net.explorviz		SQLStatementHandler	SampleServer -> Main
net		SampleServer	SampleServer -> Main

11	12	13	14	15	16	17	18
Main$ApplicationTask -> JavaExample	gelöscht	...664	deleted everything ..664	3	2	2	3
SQLStatementHandler->createStatemen()	got deleted	664	Deleted 664	2	2	1	5
SQLStatementHandler -> createStatement()	It was removed entirely	1567741979664	It was reledet in the version with timestamp 1567741979664	2	3	1	2
	Alles wurde gelöscht	Letzte Version	Deleted, 1567741979664	3	2	2	3
ApplicationTask	deletet	664	664	2	2	1	5
JavaExample->Fibonacci	Complete deleted :(	664	664 - deleted	2	2	1	4
SQLStatementHandler -> createSatement()	It got deleted	664	Deleted	4	4	3	5
SQLStatementHandler -> createStatement()	everything got deleted	664	got deleted in 644	2	4	3	5

19
-andere farben wählen in der leiste. rot für löschen. Den text in den buttons weiß machen, damit besser lesbar. -die suche nach komponenten funktioniert nicht wie erwartet. -landscapes sollten auch in einer nicht kronolgischen reihenfolge in die applikation reinladbar sein. (graph unten) -deslektieren von landschaften durch erneuten klick auf den roten datenpunkt.
always output history, no need to hover, so you can just remove it if you need the space as the user
The buttons for hiding or showing deleted or added components/communications etc. don't seem intuitive to me for some reason. The colors, like yellow, make the labels hard to read.
Klassennamen nicht überall lesbar Mehrere Graphik Beispiele wie welche Veränderung aussieht
andere Farbe für Hide Added,
Die Zeitstempel in einem gängigen Datumsformat wären besser, ebenso dass die History nicht automatisch nach 3 Sekunden ausgeblendet wird.
Maybe highlight the History-Button.
Very difficult to select the component I aimed for. Timestamps are very unintuitive, rather use a normal date.