Date: 11.07.2021



Exemplar Management System

Team 04

Kevin Schütz, Valentina Hummenberger, Julia Hammer

Project  
documentation

Table of Contents

[1. Introduction 3](#_Toc76666974)

[2. Implemented Requirements 3](#_Toc76666975)

[3. Overview of the system from the user point of view 3](#_Toc76666976)

[4. Overview of the system from the developer point of view 16](#_Toc76666977)

[4.1. Design 16](#_Toc76666978)

[4.1.1. Overview of the system 16](#_Toc76666979)

[4.1.2. Important Design Decision 16](#_Toc76666980)

[4.2. Implementation 17](#_Toc76666981)

[4.3. Code Quality 22](#_Toc76666982)

[4.4. Testing 29](#_Toc76666983)

[5. Installation instruction 32](#_Toc76666984)

[6. Screenshots of the provided common exemplars 32](#_Toc76666985)

Version History

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Creator | Changes |
| I  (Release 1) | 08.04.2021 | Kevin Schütz | Backend with Database, first implementation of User Dashboard, first implementation of Exemplar Dashboard |
| II  (Release 2) | 13.05.2021 | Valentina Hummenberger | Adaption of Libraries, Finalization of Exemplar Dashboard (e.g. comment section), changes in Database |
| III  (Release 3) | 25.06.2021 | Julia Hammer | Implementation of Communities and the according library, finalization of Database (hosted locally) |

Table : Version history

# Introduction

We implemented an Exemplar Management Tool as a desktop application according to the requirements presented by Prof. Luca Berardinelli in the course PR Software Engineering at the Johannes Kepler University in the summer term 2021.   
The application enables users to retrieve, rate, export or label exemplars, join communities and more. If they care to be creators, they may confirm accordingly whilst registration or opt in later. If a user is also declared a creator, she/he is enabled to share her/his exemplars. An exemplar is hereby a program designed by a creator, which tries to solve a specific Problem.

# Implemented Requirements

We implemented all the requirements pointed out in Moodle, except the optional requirement of the most accessed exemplar. We did not exactly contribute each of the requirements to a team member, but programmed together and finished where another member started, when indicated. If the requirements must be parted amongst the team members, we would conclude that Kevin was responsible for the basic requirements as well as a focus on the exemplar related requirements. Valentina was responsible for the requirements regarding the contributors and Julia did program the Communities and was responsible for the corresponding sort sections. We have not tracked our time but can safely assume that we all met the 150 hours of work. Detailed insights may be found on github.com.

# Overview of the system from the user point of view

The system is used as a local application, connected with a data base. One enters the application via the desktop icon and is asked to log in as follows:



1. Basic: Create/Retrieve/Update/Delete an Exemplar profile (Name, Contributors, Context, Problem, Solution,...)

Create: At the Hometab exemplars can be created if one chooses to be a creator:



Retrieve: An exemplar can be accessed at the homepage (see picture above), the Exemplar library, or the search function:



Update/Delete: In the exemplar dashboard an exemplar can be updated as well as deleted



2. Basic: Create/Retrieve/Update/Delete a User profile

Create: When one opens the application and does not yet have a User profile, there is the possibility to “register” and create a User profile:



Retrieve/Update/Delete:

When logged in the User Profile can always be retrieved, updated and deleted in the Home Tab:



3. Basic: Create/Retrieve/Update/Delete a Contributor profile. A Contributor is a registered User

We implemented the Contributor profile as an extended User profile. If one decides to contribute, the Button “Contributor?” can the chosen in the Registration form. There is also the possibility to opt later in the User profile. The basic process is already explained in requirement number 2. Furthermore there is the possibility to search for creators:



Creators can also be accessed via the Contributor library:



4. Basic: Label can be assigned to Exemplars by Users

In the Exemplar dashboard/ Exemplar tab any label can be assigned through clicking a button at the bottom and giving a chosen name:



5. Basic: Ratings can be assigned to Exemplars by Users

Ratings can be assigned quite similar as requirement 4 in the exemplar tab:



6. Basic: Create/Retrieve/Update/Delete communities of users. Each community contains a list of reference exemplars.

Create: In the Home Tab a new community can be created, if the name differs from any existing community:



Retrieve/Update/Delete: Through clicking “open selected” one can choose which community to retrieve and Update or Delete in the opened tab:



Furthermore there is the possibility to search for communities:



7. Basic: Users can comment and reply to comments about an Exemplar

By clicking “Leave Comment” in the exemplar section any comment can be given:



Also in the exemplar dashboard, any user can reply to a comment:



8. Queries: Show exemplars with specific labels attached.

In the exemplar library by marking “Filter by label” one can choose exemplars accordingly:

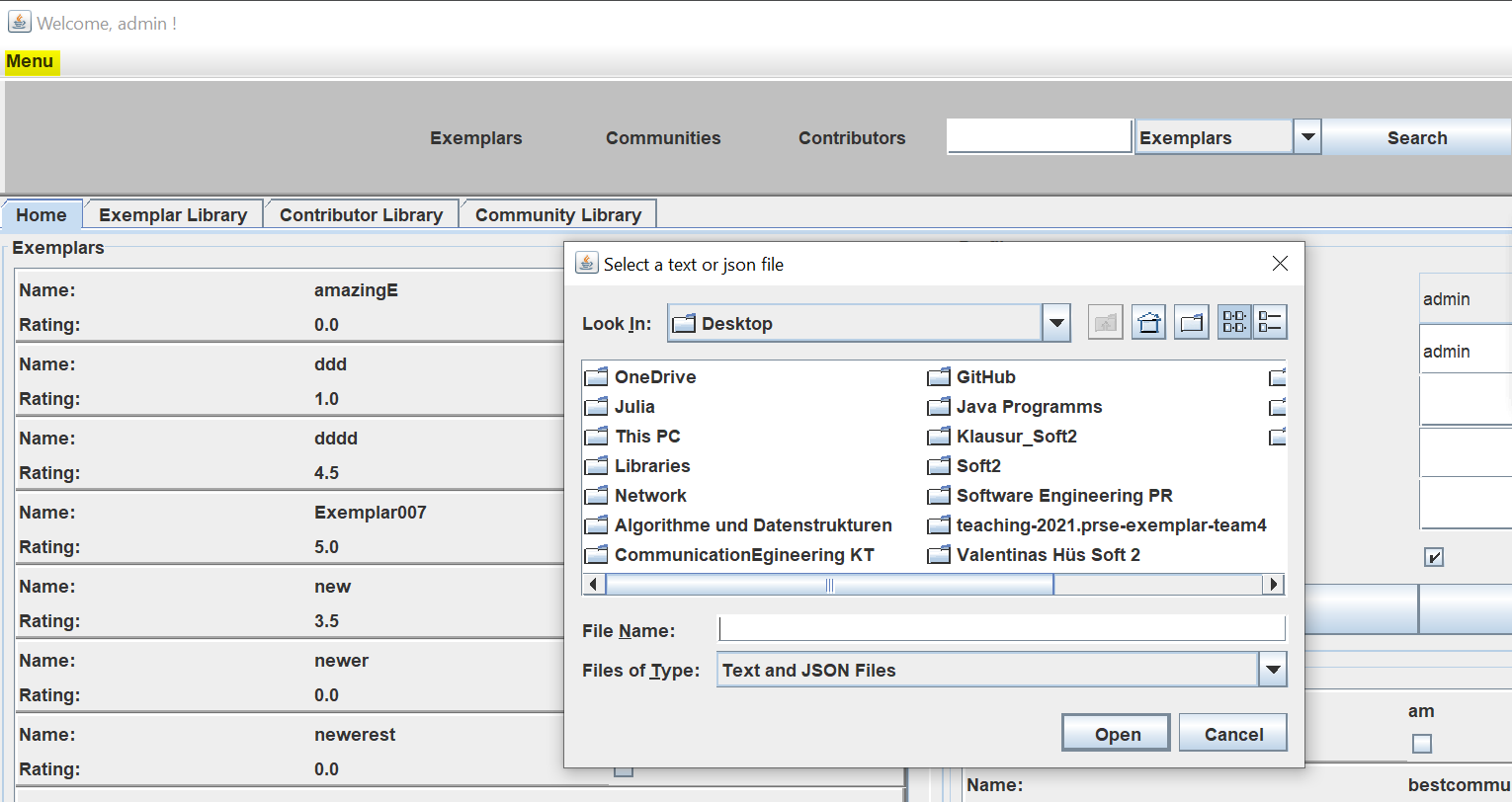


9. Import/Export: JSON-based import/export of Exemplars

In the exemplar tab through clicking on “export” a specific exemplar can be stored locally:



Import: Menu – Exemplars – Import:



10. Analyse: Exemplar Dashboard including contributors, users, labels, and ratings.

The exemplar dashboard can be accessed through the home tab, the exemplar library, or the “search” function. Here is a picture of our Exemplar dashboard, which contains the contributors, labels and ratings:



11. Analyse: Contributor Dashboard including contributed Exemplars, labels of contributed Exemplars, and overall rating of contributed Exemplars.

Out contributor dashboard can be accessed through the contributor library or the search function. It contains the contributed Exemplars, the labels of the exemplars an the overall rating of each exemplar plus an average rating over alle exemplars regarding the contributor:



12. Analyse: Trend analysis - Which are the most (accessed and) rated exemplars of the last week? () = optional requirement

In the exemplar library the most rated exemplars can the accessed by choosing the according button:



13. Sort: Classify Exemplars by avg. rating, by # of users

In the exemplar library the sorting by rating can be chosen:



14. Sort: Classify Top Contributors w.r.t. a particular label

In the contributor library contributors can be sorted according to a particular label:



15. Sort: Classify Top Contributors w.r.t. ratings of contributed Exemplars

In the contributor library the sorting by the average rating of an exemplar can be chosen:



# Overview of the system from the developer point of view

## Design

### Overview of the system

Our overall architecture may be described as a three-layered-one. The data layer is realized as a relational Microsoft SQL database which is hosted on Azure. The server is implemented as a Spring Boot application and acts as middlemen between the Frontend and the database. The client is represented by a desktop application which is written in Java using the Swing Framework.

We tried to stick to the model view controller design pattern in the frontend by dividing our project into three main areas.

The model contains all the entities and the http clients that connect to our backend to perform CRUD-Operations on our SQL database.

The controller package contains the business logic. We excluded the logic from the view by creating custom listeners that are triggered in the view but implemented in the controller(example see further down in 4.2).

The view package has all the graphic components of the application.

In the backend we implemented a REST-API using Spring Boot. We have one RequestController for every entity with different interfaces to perform operations on them (Create-Read-Update-Delete etc). We used Hibernate for object-relational-mapping between our Java classes and the entities in the database (see 4.2)

Our UML diagrams are quite large as they are auto generated by Plantuml and contain every little detail.

Therefore, we refer to github regarding the diagrams.

### Important Design Decision

*Description of the 3-5 most important design decisions according to the following scheme*

**Decision:** Contributors are users, who must declare, that they want to contribute at registration or opt later.

**Reason:** Since the requirements did not conclude how to implement creators specifically, we had to decide.

**Alternatives that were considered:** An extra log-in frame for contributors without the possibility to opt.

**Assumption:** We assumed that our implementation would be easier for users to handle and that, with the given possibility to opt, we might win more contributors. We assumed furthermore, that the more contributors there are, the better it will be for our program and communities.

**Consequences:** A somewhat lighter program missing another extra feature, making the handling easier for the users. Possibly there might be more contributors.

**Decision:** How to add the exemplars to the communities. We decided, that each exemplar shall be added to a specific community separately.

**Reason:** It was not defined, how communities shall work specifically. The Question was, whether communities focus on exemplars or users.

**Alternatives that were considered:** The alternative was to focus on the users and add alle exemplars of every user to the community.

**Assumption:** Since the focus of the exemplar management tool is on the exemplars rather than the users. We assumed that communities would be formed, not to boast whichever community contributed more exemplars and stash them, but to work together. We reasoned, that one might also want to add an exemplar from a contributor, not part of the community, to help with the community problems.

**Consequences:** Therefore we implemented, that every exemplar available can be added to any community. If a user/creator joins a community, his/her exemplars are NOT added to the communities exemplars automatically.

**Decision:** Menu Panel with libraries

**Reason**: At first, we did not have a menu panel on top. However, when the implementation process proceeded, the program seemed to be less user friendly and more confusing to handle.

**Alternatives that were considered:** Adding the libraries to the Home Tab.

Assumption: We assumed, that our implementation with the menu panel would be more user friendly. Since the menu can be addressed with any panel open, users are more likely to find it.

**Consequences:** Buttons on top and on the bottom of the program were added.

## Implementation

**Frontend**

As mentioned, we divided our client into three areas: the model itself, which represents all the relevant data and interacts with the REST-API, the view, which includes the design elements and finally the controller, which implements the business logic. Below we will try to give an overview of the core concepts used in order to ensure the separation between these parts.

Model

For the model we created entities/classes for Exemplars, Users, Ratings, Comments, Communities, etc.

Corresponding to these entities we implemented http clients (*java.net.http.HttpClient*) that extend an abstract and generic Client-class that dictates the common CRUD operations. These clients communicate with our backend. In order to map the JSON-Strings from the backend we used ObjectMappers (*com.fasterxml.jackson.databind.ObjectMapper).*

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

*Example of an add-method provided by the CommentClient that takes a comment and adds it to the database by sending it to the Spring server as HTTPRequest (POST)*

Ein Bild, das Text enthält.

Automatisch generierte BeschreibungIn order to be able to execute sorting and filtering operations, we used streams whenever it was possible *(java.util.stream*):

*Example of a listener that sorts a List of exemplars according to the average rating using streams.*

Controller

Our controllers consist of the *LoginController.java*, which is responsible for the login operation, and the *MainController.java*, that handles most of the logic.

In order to detach the logic that is required in the view from the view and implement it in our controllers, we used custom listeners. These listeners are Single-Abstract-Method interfaces that are members of our view classes. We generated Setters for these members and implemented them in our controllers. If, for example, the “Login”-button from the LoginFrame is clicked, the ActionListener that listens to the click-event activates a custom LoginListener and passes the information as parameter. The listener itself although is implemented in the LoginController and handles the passed information:

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

*The listener which is a member of the LoginFrame*

*The implementation of the listener inside the contoller*

In order outsource interactions with the model from the view to the controller we implemented an asynchronous method inside the MainController that regularly fetches (every 30 seconds) all the required data from the database. This method saves the data in static members of our MainController. During the update intervall, the view gets the required data from these members. Update, delete and add operations are carried out on the database as well as the local copy in order to ensure consistency.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

*Method that regularly updates the local copy of the data*

*Ein Bild, das Text enthält.

Automatisch generierte Beschreibung*

*Method from the view that relies on the data from the MainController*

View

For the graphical components we used the Swing library. We implemented the user interface by creating different frames, panels, listeners and events. The *JTabbedPane* plays an essential role in our implementation since the user can access most of the information via tabs (= panels which are added to an instance of the *JTabbedPane*):

*Method provided by the MainFrame that adds a Component as tab with a custom title*

The most important panels include the Home Panel, the Exemplar and Contributor Dashboard and the Library Panels. Most of the user operations can be carried out via buttons (for instance creating a new Exemplar or closing a tab). Clicks on these buttons can be handled by adding an ActionListener to the button that calls a custom listener(see above).

**Backend**

The server or backend of our system is realized as a Spring Boot application as mentioned above. We are creating a Docker container from this application and hosting it on Azure as Container Instance so that it can be accessed from every client remotely.

The core of this project are the entities required for the system. These entities are described as Java classes and use Hibernate in conjunction with the Spring Data JPA in order to implement object-relational-mapping between the classes and the database.

Using this method, 3 things are required for every entity:

@Entity

A class with the @Entity annotation needs to be implemented with different annotations in order to explicitly define IDs, relationships, tables, etc.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

*Example of a class with the required annotations*

The repository

The repository extends the *JpaRepository* that comes with the Spring Data JPA dependency. This is a generic interface that is typed with an entity and the datatype of the entity’s id. These repositories offer a wide range of CRUD-methods out of the box and can be extended by custom SQL-Queries that can be used as methods.

**Ein Bild, das Text enthält.

Automatisch generierte Beschreibung**

The repository for the entity shown above that has some custom queries defined

@RequestController

Ein Bild, das Text enthält.

Automatisch generierte BeschreibungThe RequestController offers an API for calling the methods provided by the repository as needed. This is the interface which is used by the frontend to interact with the database.

Small part of the controller for the running example

**Database**

As mentioned, we are using a relational SQL database. For this we chose a Microsoft SQL database and host it on Azure so that it is remotely accessible.

## Code Quality

**Sonarlint - Code Analysis Plugin**

* Issues bevor refactoring: 892 in 76 files
* Issues after refactoring for Release 3: 60 in 22 files
* Issuers after refactoring for Final Release: 48 in 20 Files

Most prominent issues:

**M**Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

## 

## Testing

We created the following test-classes containing 67 Unit Tests:

*Ein Bild, das Text enthält.

Automatisch generierte Beschreibung*

Our tests where mostly centered around the Http-Clients that are used to interact with the backend/ the database and connect all components. With this approached we achieved the following overall coverage:

Ein Bild, das Text, Screenshot, Monitor, schwarz enthält.

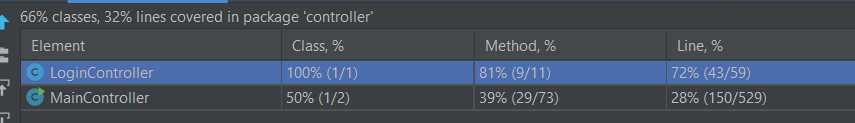
Automatisch generierte Beschreibung

The coverage for the model package that contains the entities and the clients and therefore the core of the application is as follows:

Ein Bild, das Tisch enthält.

Automatisch generierte Beschreibung

We also tried to test some core functionalities from our controller-classes with the following coverage:



Here we focused on the login and registration process to verify that these are working properly. We also tried to test some listeners and other methods that are testable but this proved to be difficult as many actions have to be confirmed or open a JOptionPane that has to be manually closed after the action is completed. Changing this would require us to alter the functionality of our application to an extent that is not in line with the focus on the user experience.

In the View, we achieved the following coverage:

Ein Bild, das Text, Screenshot, Monitor enthält.

Automatisch generierte Beschreibung

Circle CI:

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

*We could not enable all tests for Circle CI as some required user interaction*

***Acceptance Tests:***

|  |  |
| --- | --- |
| Test case ID | **Test\_addUser (TestUserClient)** |
| Designed by | Kevin, Julia, Valentina |
| Executed on | 26.6.2021 |
| Carried out by | Kevin |
| Tested Requirement | Create new User/Contributor |
| Requirement |  |
| Test steps | 1. Create new Test User 2. Add User to database |
| Test data | Test User |
| Expected result | The added User should be returned |
| Postcondition | A test User must be created. |
| Status | Passed |
| Comments |  |

|  |  |  |
| --- | --- | --- |
| Test case ID | | **Test\_getExemplar (TestExemplarClient)** |
| Designed by | | Kevin, Julia, Valentina |
| Executed on | | 26.6.2021 |
| Carried out by | | Kevin |
| Tested Requirement | | Retrieve Exemplar |
| Requirement | |  |
| Test steps | | 1. Create Test Exemplar and add it to database. 2. Fetch Exemplar by entering the name of the previously created Exemplar |
| Test data | | Test Exemplar |
| Expected result | | The same Exemplar which has previously been added to database should be returned. |
| Postcondition | | A test Exemplar must be created and added to the database. |
| Status | | Passed |
| Comments | |  |
| Test case ID | **Test\_addExemplar (TestExemplarClient)** | |
| Designed by | Kevin, Julia, Valentina | |
| Executed on | 26.6.2021 | |
| Carried out by | Kevin | |
| Tested Requirement | Create new Exemplar | |
| Requirement |  | |
| Test steps | 1. Create Test Exempar 2. Add it to the database | |
| Test data | Test Exemplar | |
| Expected result | The added Exemplar should be returned | |
| Postcondition | A test Exemplar must be created. | |
| Status | Passed | |
| Comments |  | |

# Installation instruction

*For a description on how to install and start the application we refer to our github repository, more specifically the README.md*

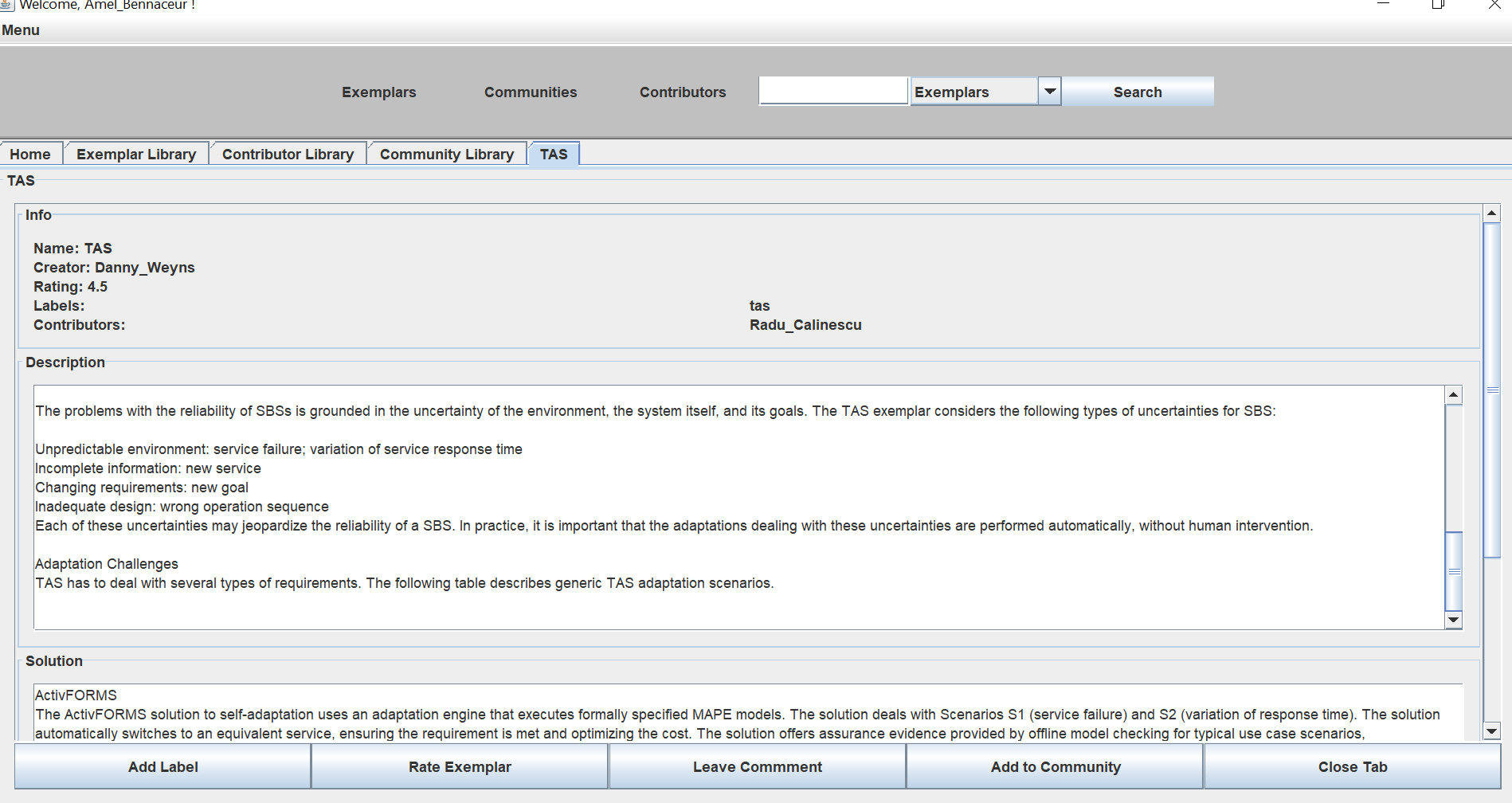
*https://github.com/jku-win-se/teaching-2021.prse-exemplar-team4*

# Screenshots of the provided common exemplars

Our program does not allow pictures to be uploaded. Therefore the copied exemplars are strictly textual based.

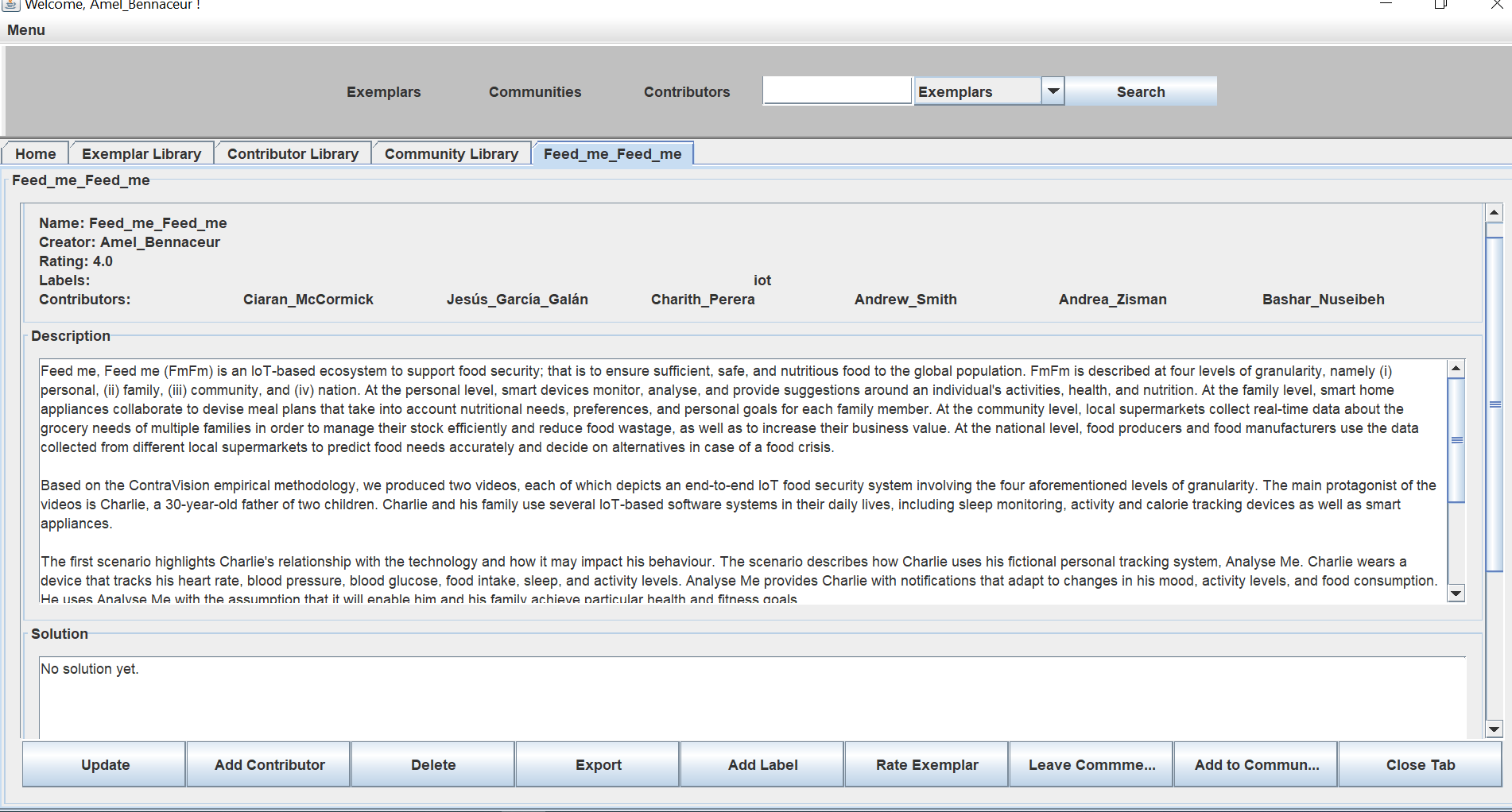
Exemplar Nr. 1. TAS:

Source: <https://www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/tas/>



Exemplar Nr. 2. Feed me, Feed me:

Source: <https://www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/feed-me-feed-me/>



Exemplar Nr. 3. ATRP

Source: <https://www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/model-problem-atrp/>

