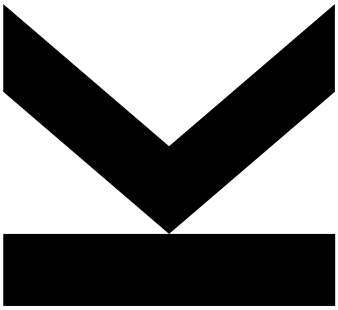
JOHANNES KEPLER
UNIVERSITÄT LINZ

Date: 11.07.2021

PROJECT DOCUMENTATION



Exemplar Management System

Team 04

Kevin Schütz, Valentina Hummenberger, Julia Hammer

JOHANNES KEPLER
UNIVERSITÄT LINZ
Altenberger Straße 69
4040 Linz, Österreich
jku.at
DVR 0093696

Table of Contents

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 1: Version history

1. Introduction

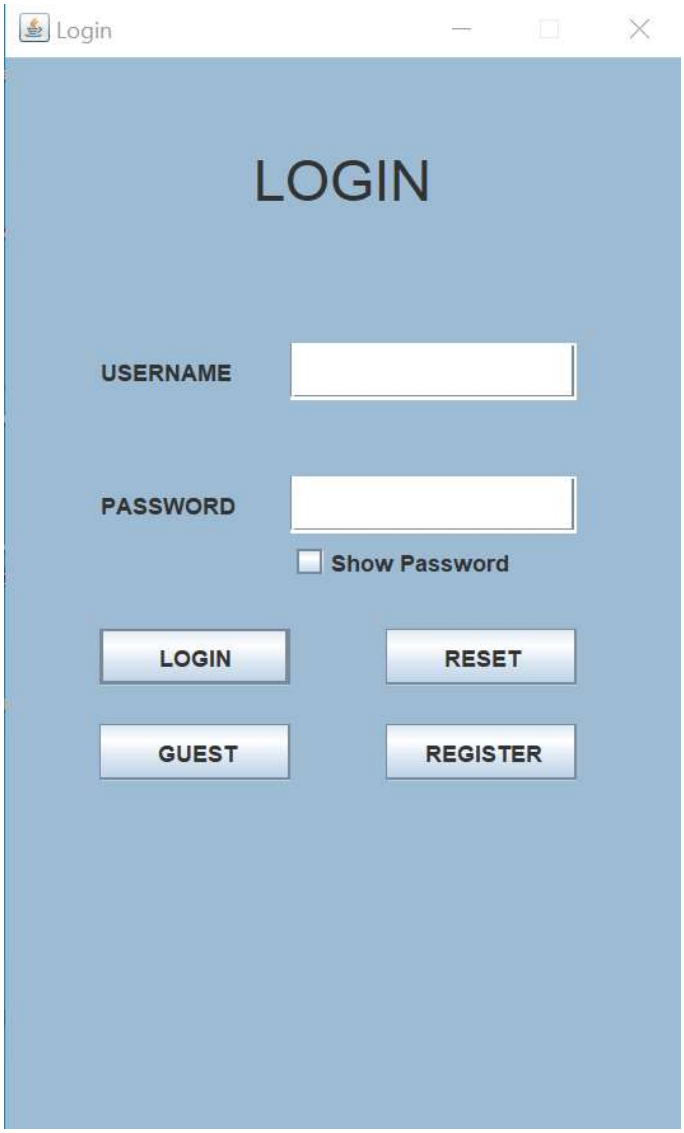
We implemented an Exemplar Management Tool as a desktop application according to the requirements presented by Prof. Luca Berardinelli in the course PR Software Engineering at the Johannes Kepler University in the summer term 2021.
The application enables users to retrieve, rate, export or label exemplars, 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.

2. 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.

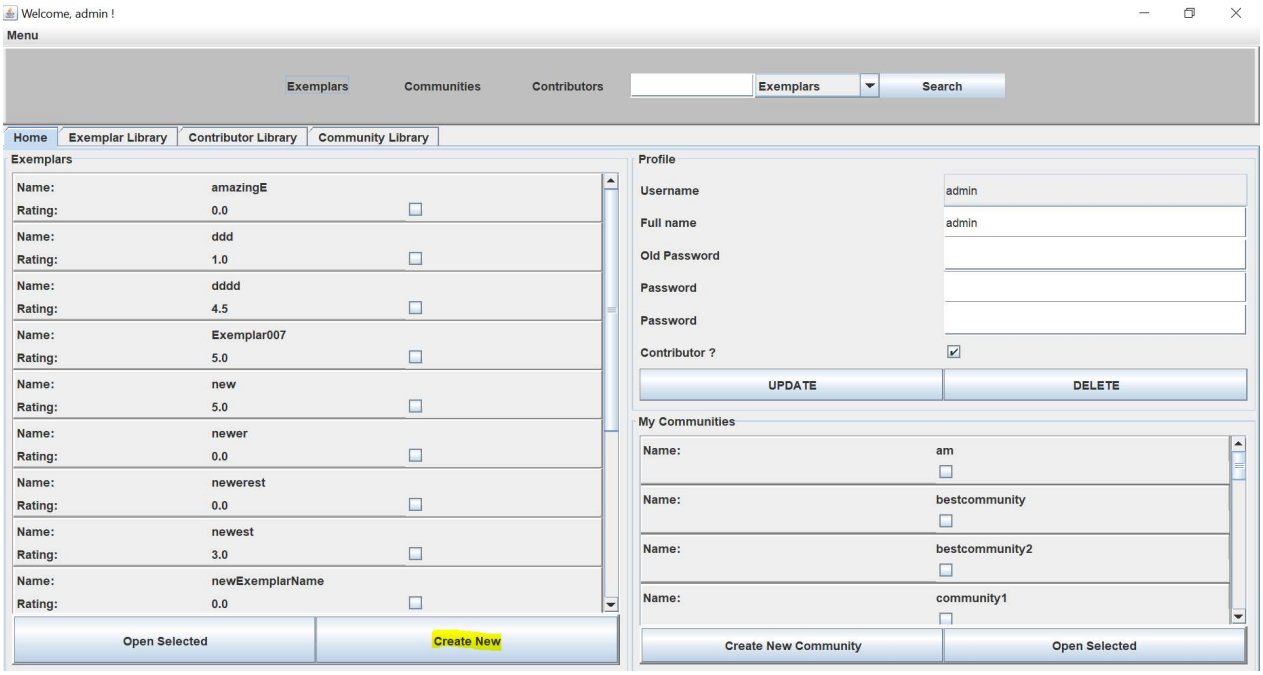
3. Overview of the system from the user point of view

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:

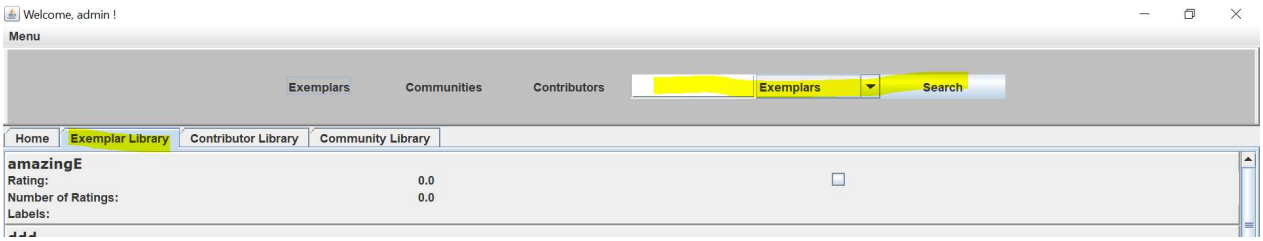


1. Basic: Create/Retrieve/Update/Delete an Exemplar profile (Name, Contributors, Context, Problem, Solution,...)

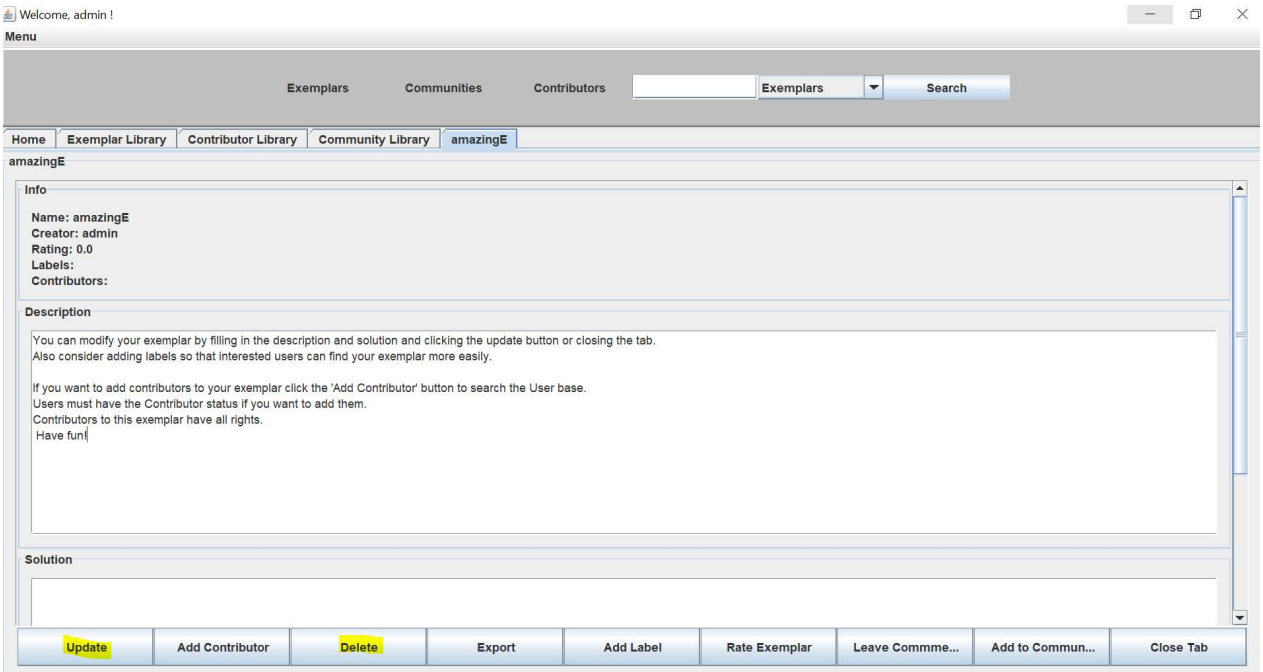
Create: At the Hometab exemplars can be created if one chooses to be a creator:



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

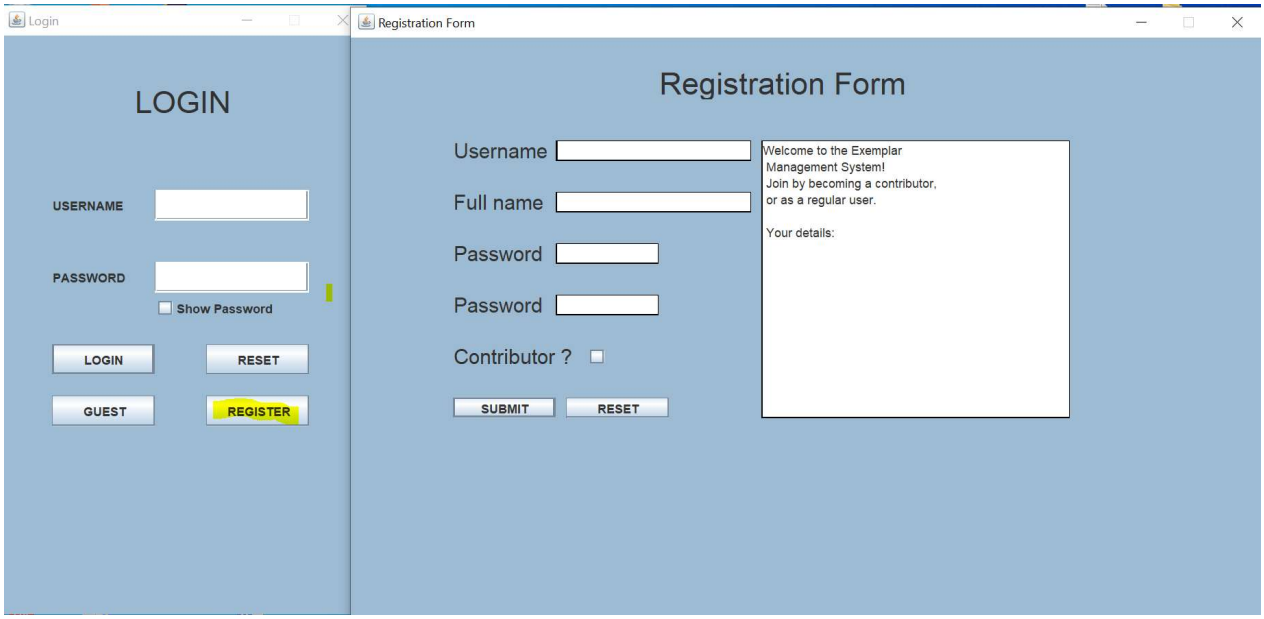


Update/Delete: In the exemplar dashboard an exemplar can be updated as well as deleted



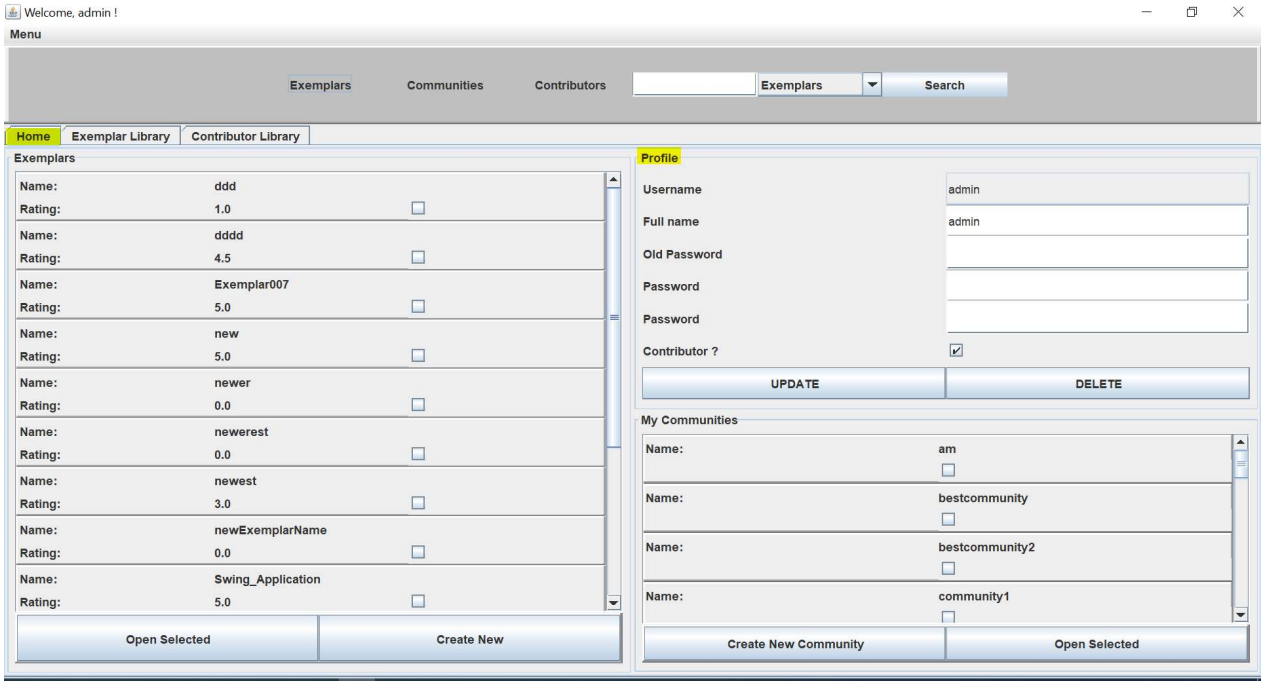
2. Basic: Create/Retrieve/Update/Delete a User profile

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:

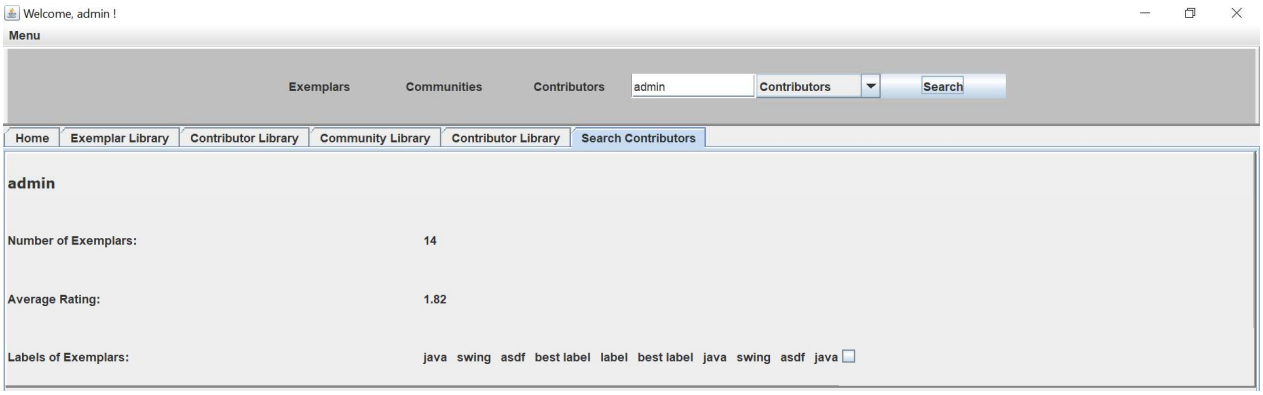
When logged in the User Profile can always be retrieved, updated and deleted in the Home Tab:



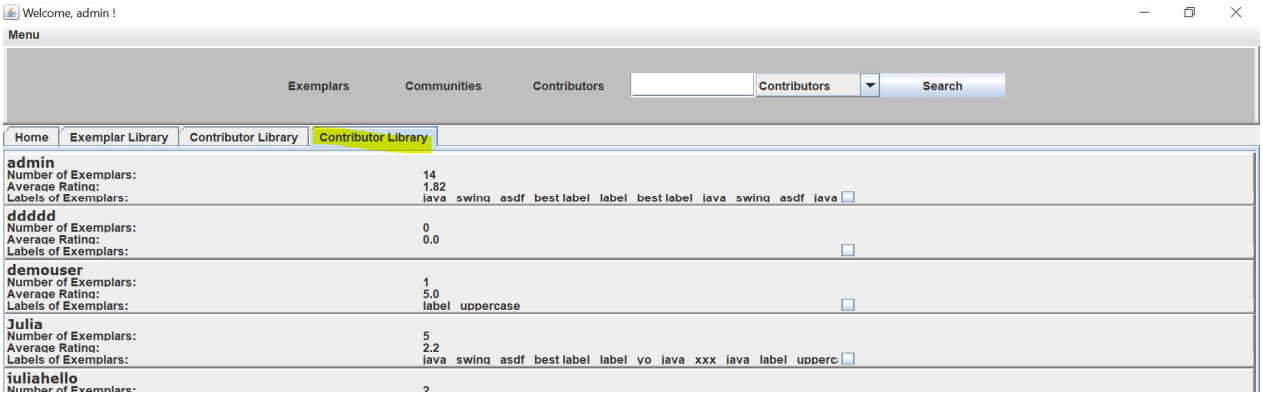
3. Basic: Create/Retrieve/Update/Delete a Contributor profile. A Contributor is a registered User

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:

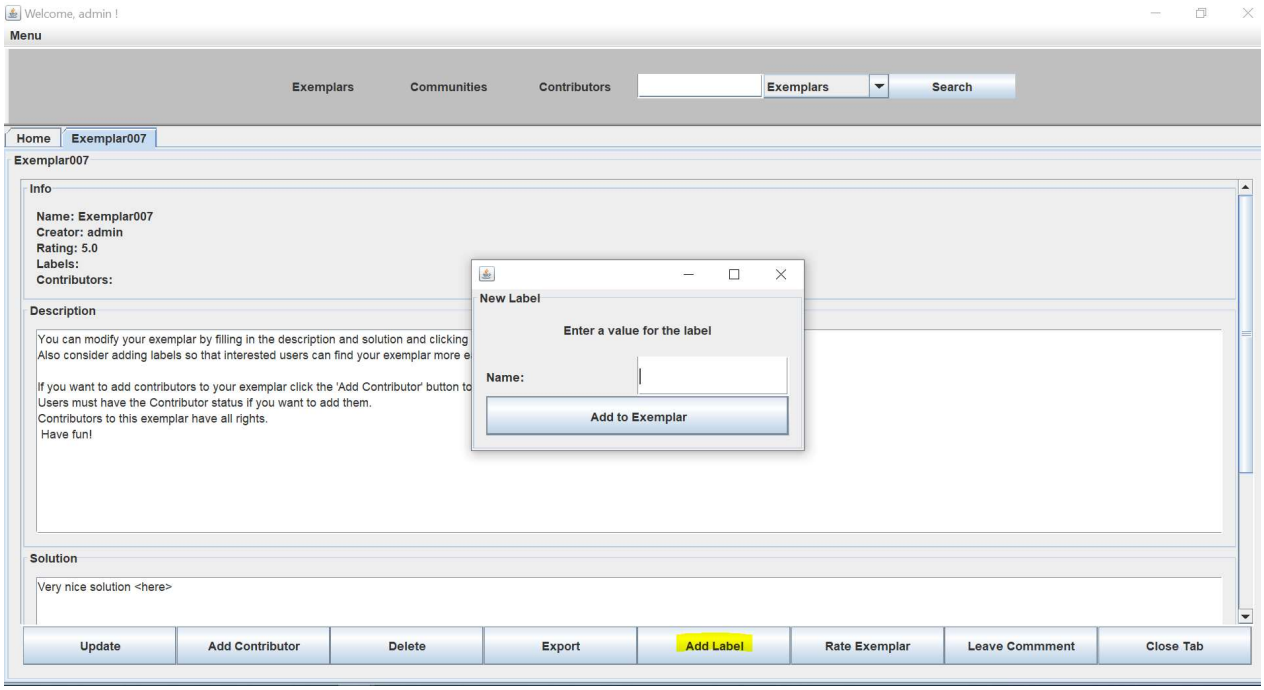


Creators can also be accessed via the Contributor library:



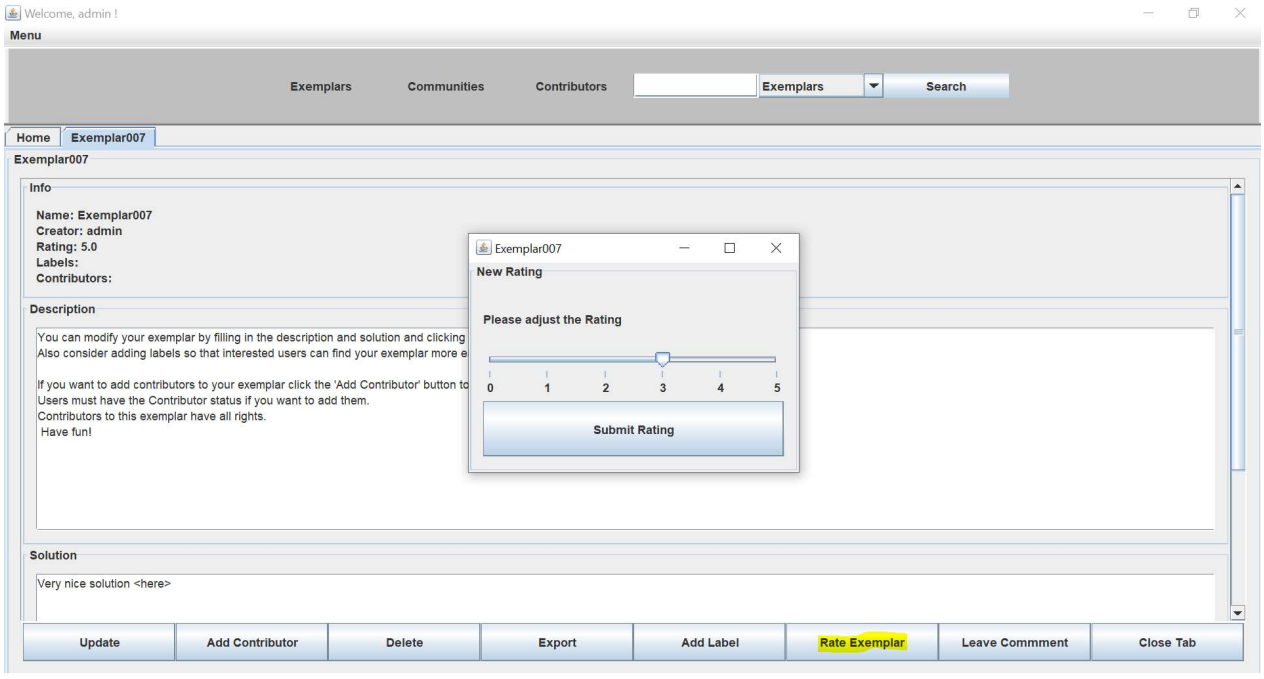
4. Basic: Label can be assigned to Exemplars by Users

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



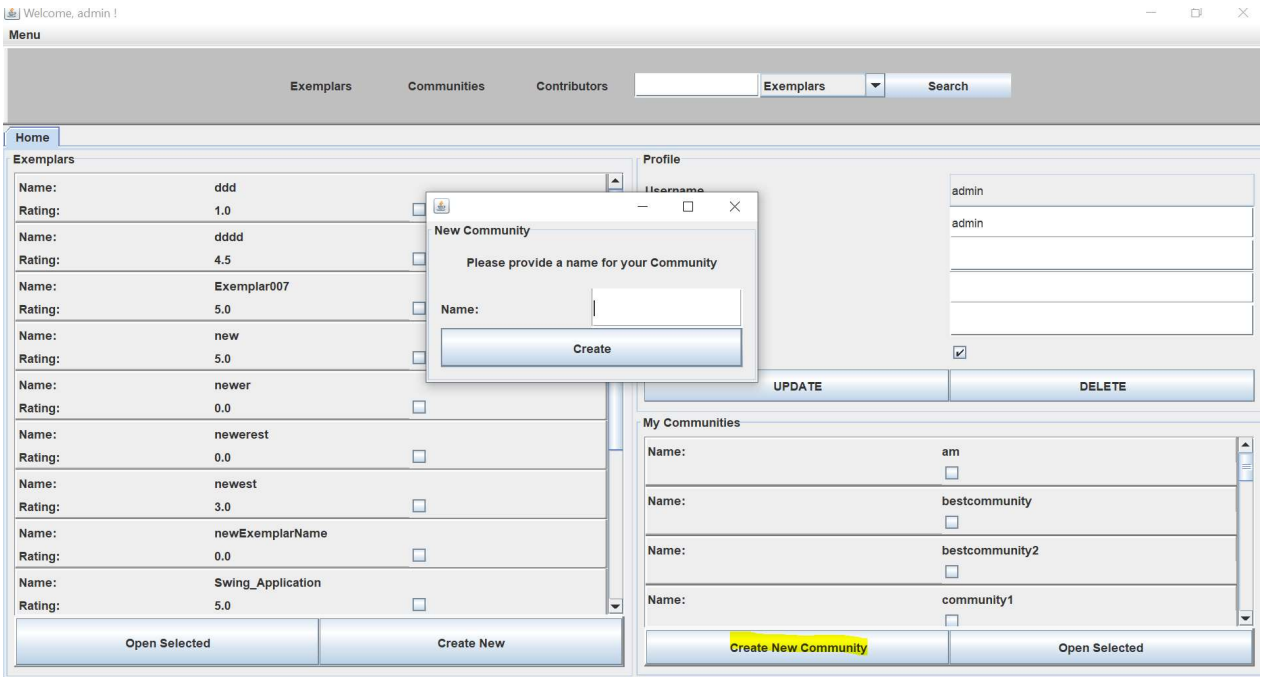
5. Basic: Ratings can be assigned to Exemplars by Users

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

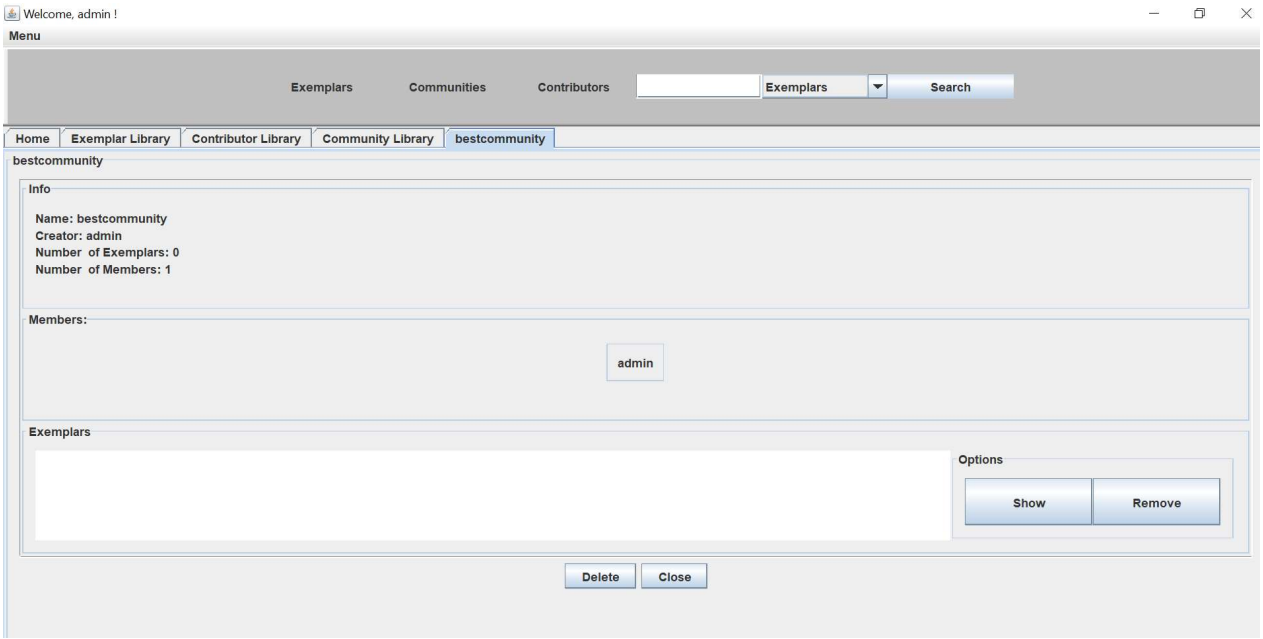


6. Basic: Create/Retrieve/Update/Delete communities of users. Each community contains a list of reference exemplars.

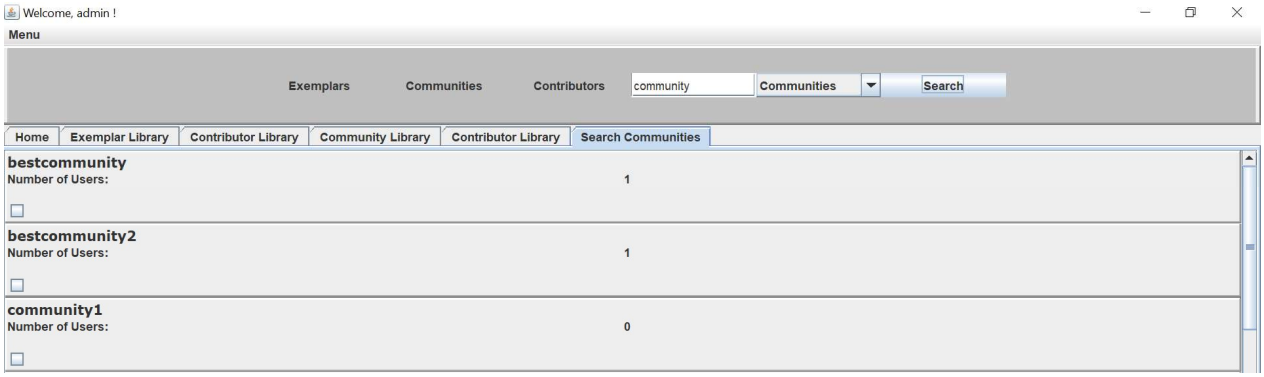
Create: In the Home Tab a new community can be created, if the name differs from any existing community:



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

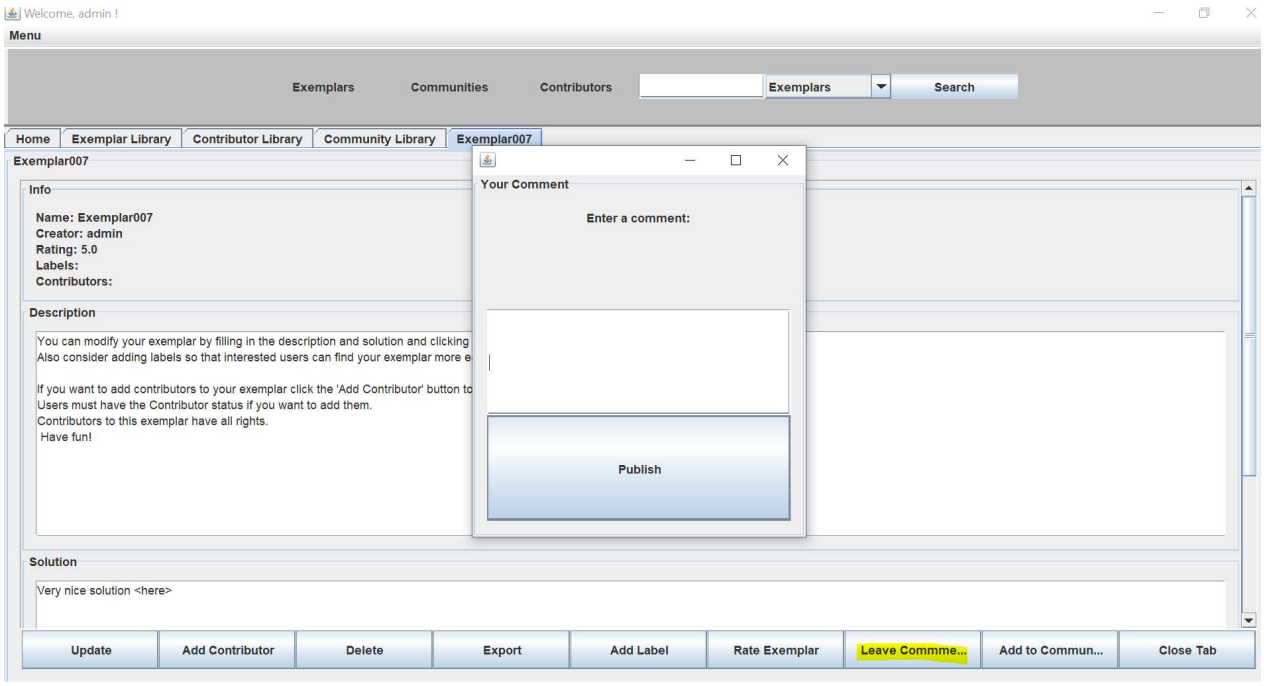


Furthermore there is the possibility to search for communities:

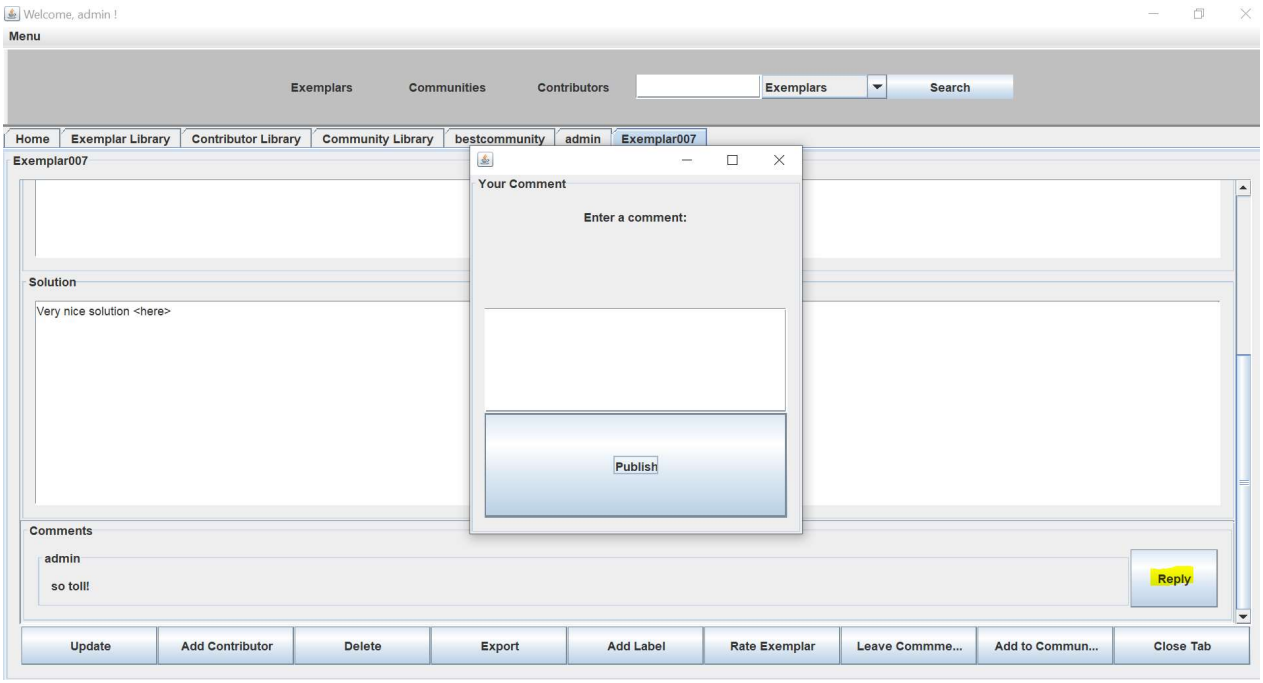


7. Basic: Users can comment and reply to comments about an Exemplar

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

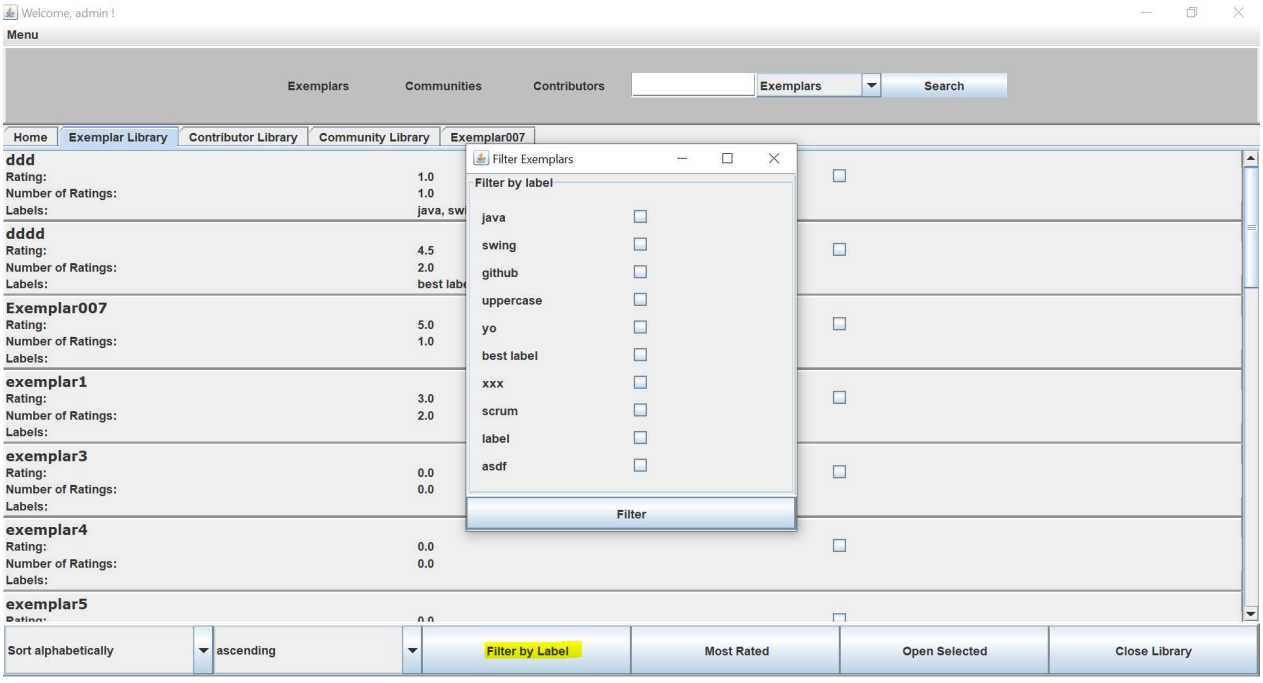


Also in the exemplar dashboard, any user can reply to a comment:



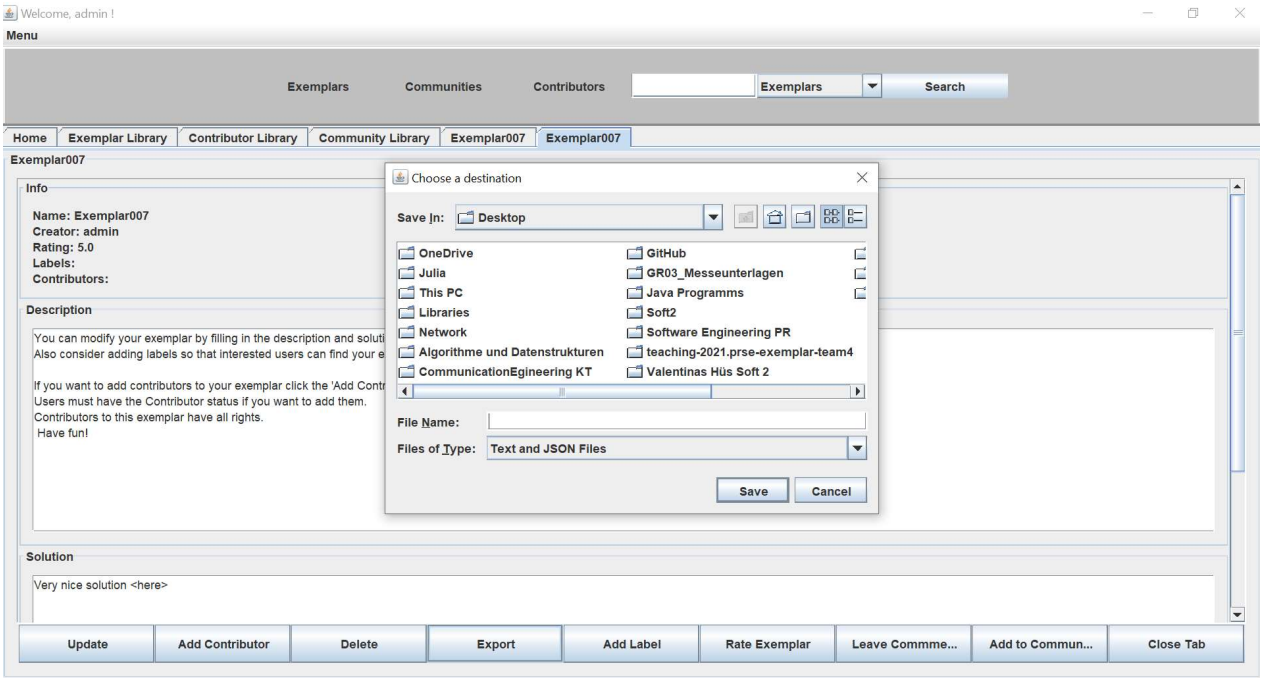
8. Queries: Show exemplars with specific labels attached.

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

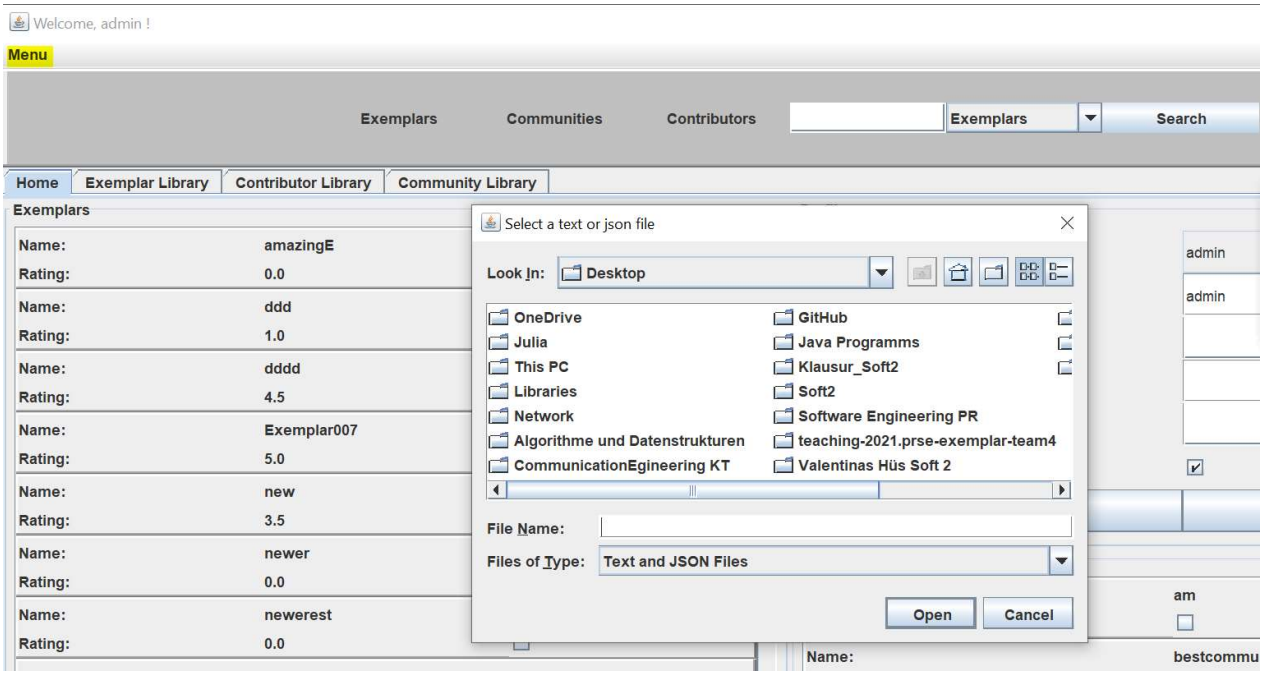


9. Import/Export: JSON-based import/export of Exemplars

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

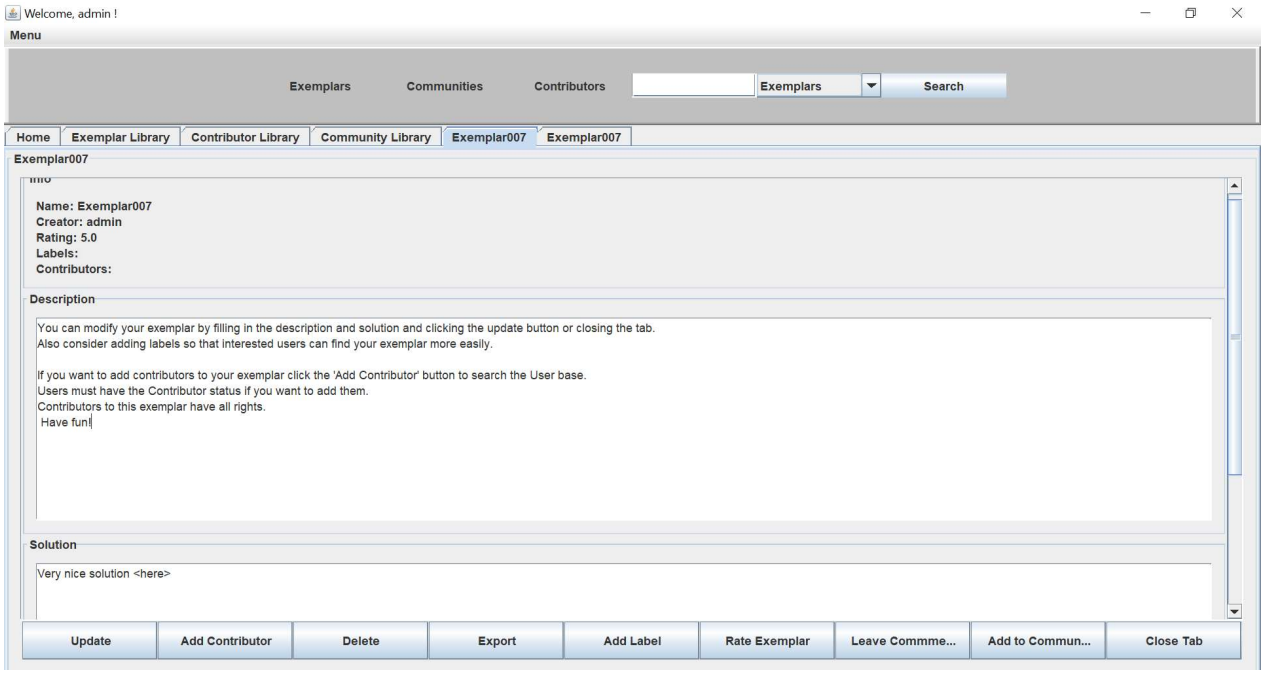


Import: Menu – Exemplars – Import:



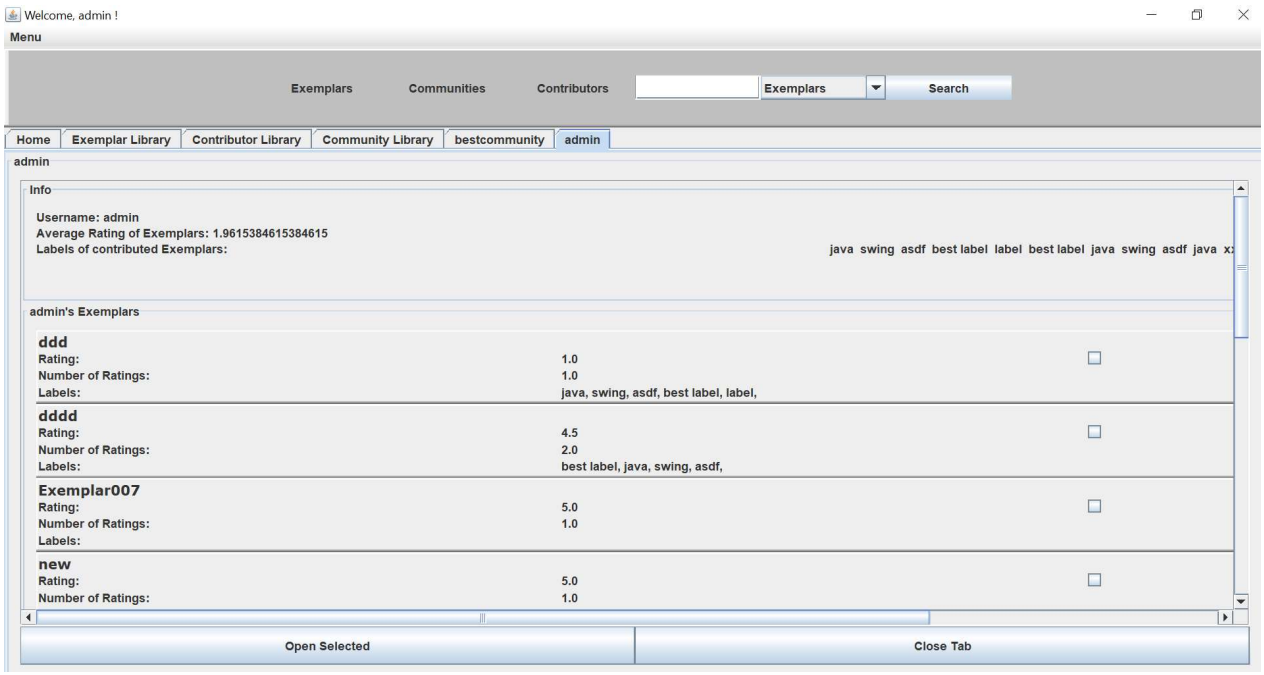
10. Analyse: Exemplar Dashboard including contributors, users, labels, and ratings.

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:



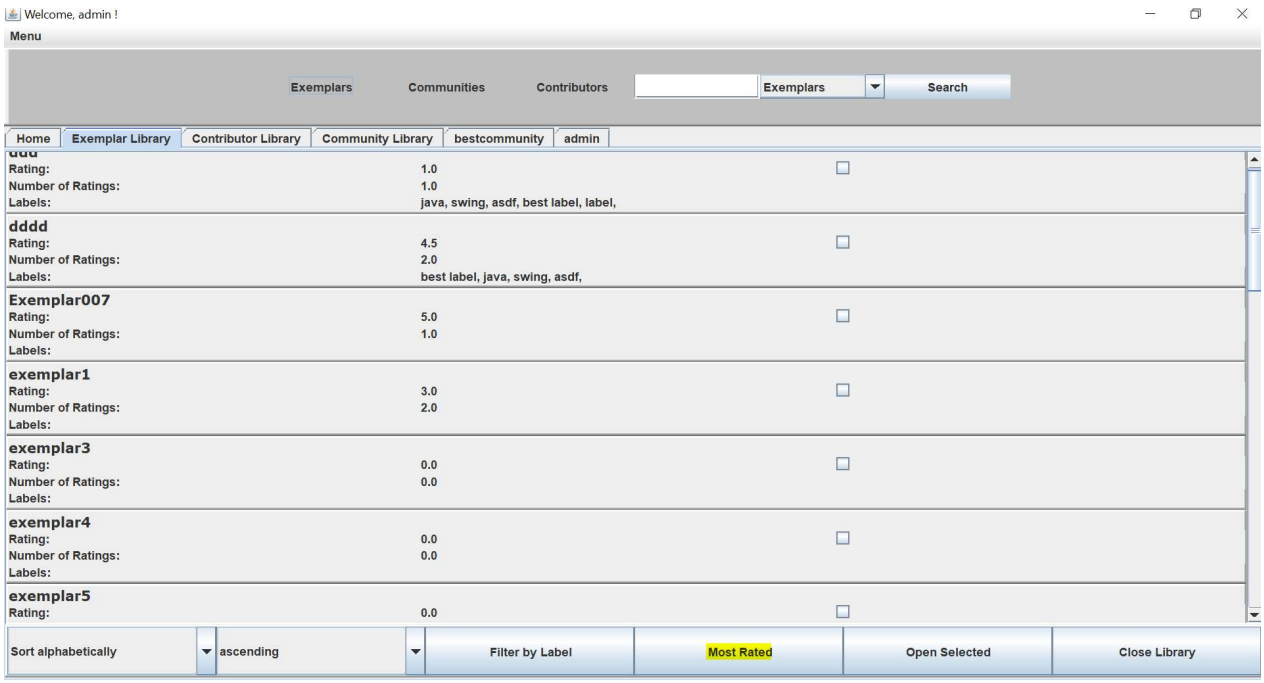
11. Analyse: Contributor Dashboard including contributed Exemplars, labels of contributed Exemplars, and overall rating of contributed Exemplars.

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:



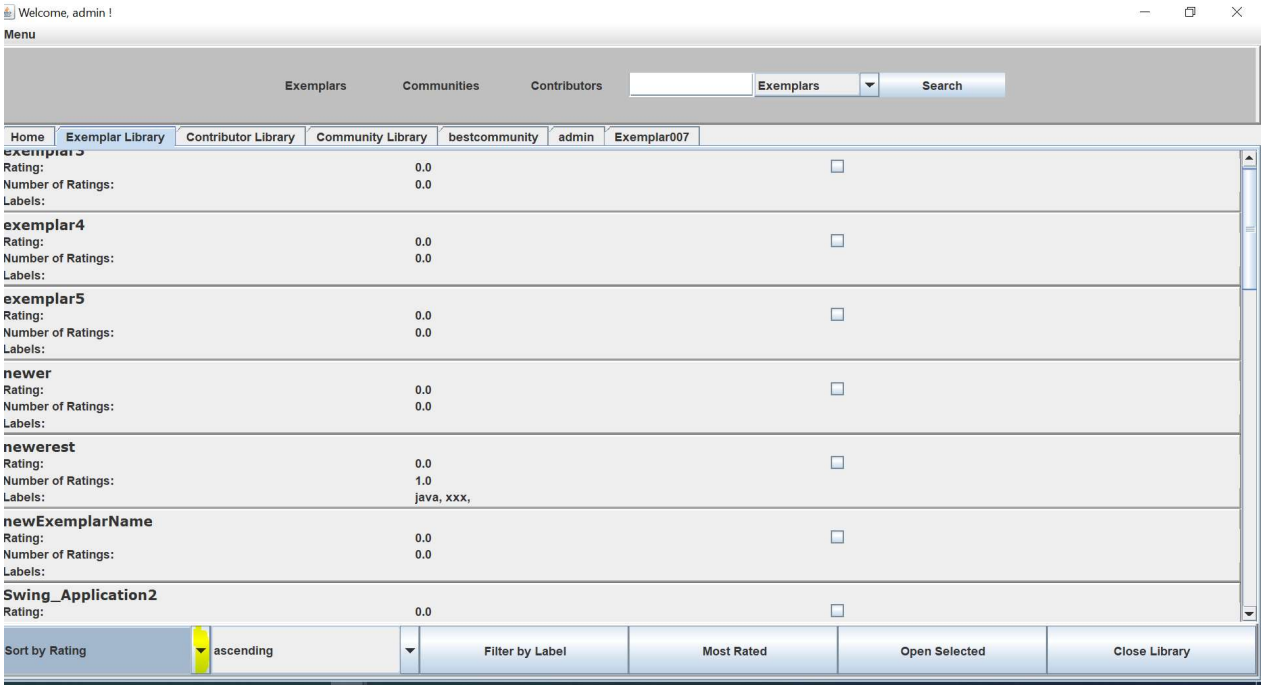
12. Analyse: Trend analysis - Which are the most (accessed and) rated exemplars of the last week? () = optional requirement

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



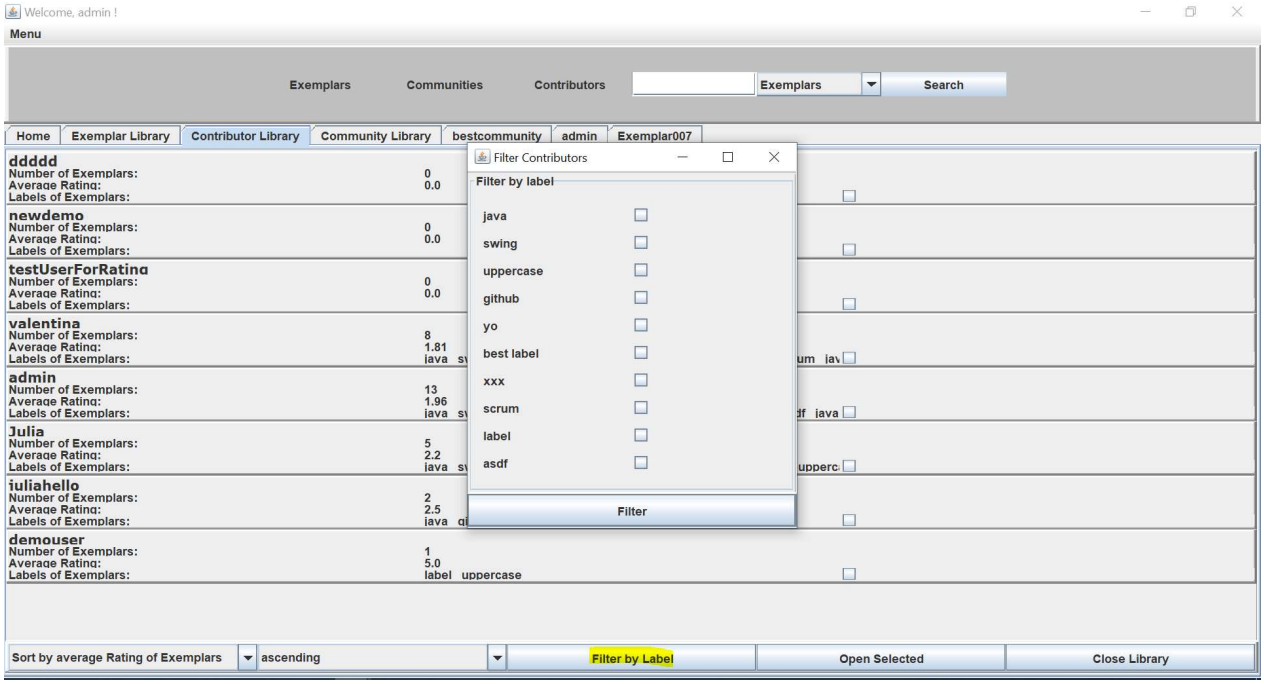
13. Sort: Classify Exemplars by avg. rating, by # of users

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



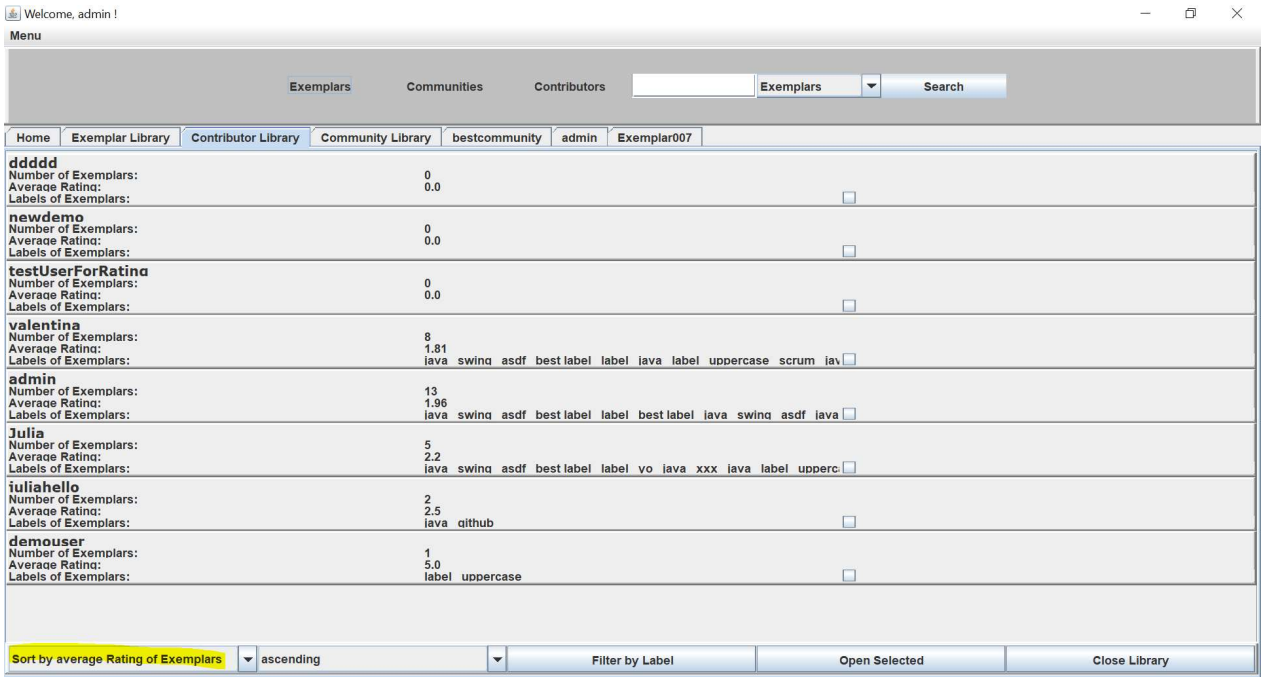
14. Sort: Classify Top Contributors w.r.t. a particular label

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:



4. Overview of the system from the developer point of view

4.1. Design

4.1.1. 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.

4.1.2. 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.

4.2. 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).*

```java
@Override
public Comment add(Comment value) throws IOException, InterruptedException {
    request = HttpRequest.newBuilder()
            .uri(URI.create(url))
            .header( name: "Content-Type",  value: "application/json")
            .POST(HttpRequest.BodyPublishers.ofString(mapper.writeValueAsString(value)))
            .build();
    response= client.send(request, HttpResponse.BodyHandlers.ofString());
    try{
        return mapper.readValue(response.body(), Comment.class);
    }catch (Exception e){
        return null;
    }
}
```

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)

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

```java
/**
 * Sort by avg rating
 */
if(sortingComboBox.getSelectedIndex() == 1) {
        allExemplars = allExemplars.stream().
                sorted(Comparator.comparingDouble(e -> exemplarAvgRatingNumberOfRatingsMap.get(e)[0])).collect(Collectors.toList());
        filteredExemplars=filteredExemplars.stream().
                sorted(Comparator.comparingDouble(e -> exemplarAvgRatingNumberOfRatingsMap.get(e)[0])).collect(Collectors.toList());
    if(sortingComboBox2.getSelectedIndex() == 1){
        Collections.reverse(allExemplars);
        Collections.reverse(filteredExemplars);
    }
}
```

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:

```java
package view.listeners.login;

public interface LoginListener {
    public void loginRequested(String username, String password);
}
```

The listener which is a member of the LoginFrame

```
loginFrame.setLoginListener(this::processLoginRequest);
```

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.

```
/**
 * Method that updates all data avery 30 seconds
 */
void asyncLoadData(){
    try {
        exemplars = exemplarClient.getAll();
        communities = communityClient.getAll();
        users = userClient.getAll();
        ratings = ratingClient.getAll();
        comments= commentClient.getAll();
        dataLoaded = true;
    } catch (IOException e) {
        e.printStackTrace();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    try {
        Thread.sleep( millis: 30000);
        asyncLoadData();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

Method that regularly updates the local copy of the data

```
/**
 * Fetches all the communities from the database
 * @param searchTerm a string used to search communities by a specific term
 */
public void fetchCommunities(String searchTerm){
    allCommunities = MainController.communities
            .stream().filter(c->c.getName().toLowerCase().contains(searchTerm.toLowerCase()))
            .collect(Collectors.toList());
```

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*):

```
public void addTab(String title, Component component) { tabPanel.addTab(title, component); }
```

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.

```
@Entity
public class Exemplar {
    @Id
    private String name;

    @Column(length=2048)
    private String problem;
    @Column(length=2048)
    private String solution;

    @ManyToOne
    private User creator;

    @ManyToMany
    private List<User> contributors;

    @ManyToMany(cascade = CascadeType.ALL)
    private List<Label> labels;
```

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.

```
public interface ExemplarRepository extends JpaRepository<Exemplar, String> {



    @Query(value = "Select * from exemplar e where e.creator_username = ?1",
        nativeQuery = true)
    List<Exemplar> findExemplarsForCreator(String creator);

    @Query(value = "select e.name, e.problem, e.solution, e.creator_username from exemplar e join exemplar_contributors c on e.name = c.exemplar_name\n" +
            "where c.contributors_username =?1", nativeQuery = true)
    List<Exemplar> findExemplarsForContributor(String contributor);


    @Query(value="(Select * from exemplar e where e.creator_username = ?1) \n" +
            "UNION\n" +
            "(\n" +
            "select e.name, e.problem, e.solution, e.creator_username \n" +
            "from exemplar e join exemplar_contributors c on e.name = c.exemplar_name\n" +
            "where c.contributors_username =?1)"
                    ,nativeQuery = true)
    List<Exemplar> findExemplarsForUser(String user);

    @Query(value="select * from exemplar where name like ?1", nativeQuery = true)
    List<Exemplar> findExemplarsNameLikeXY(String search);


    @Query(value="select * from exemplar e join exemplar_labels l on l.exemplar_name = e.name where labels_value = ?1", nativeQuery = true)
    List<Exemplar> findExemplarsByLabels(String label);
}
```

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

@RequestController

The 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.

```
@RestController
@RequestMapping("/exemplars")
public class ExemplarController {
    public static ExemplarRepository repository;

    public ExemplarController(ExemplarRepository exemplarRepository) { this.repository=exemplarRepository; }

    public static ExemplarRepository getRepository() { return repository; }

    public void setRepository(ExemplarRepository repository) { this.repository = repository; }

    @GetMapping("")
    public List<Exemplar> getExemplars() { return repository.findAll(); }
```

Small part of the controller for the running example

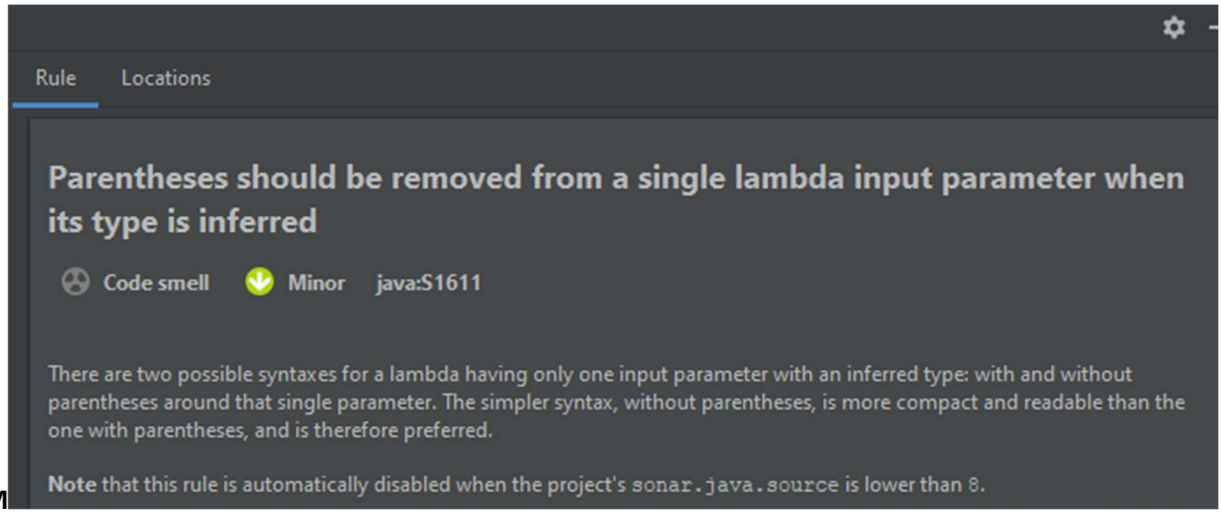
Database
As mentioned, we are using a relational SQL database. For this we chose a Microsoft SQL database and host it on Azure so that it is remotely accessible.

4.3. Code Quality

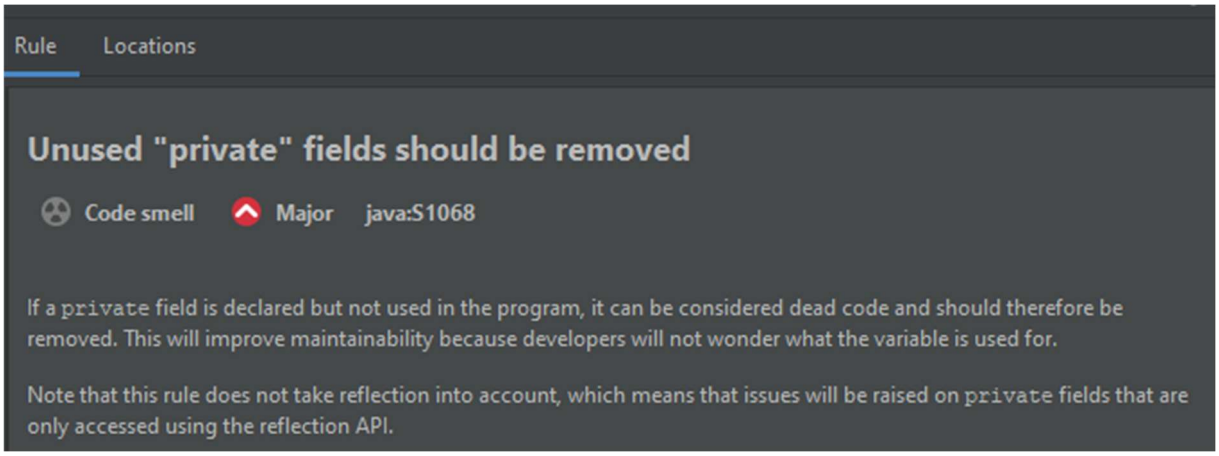
Sonarlint - Code Analysis Plugin

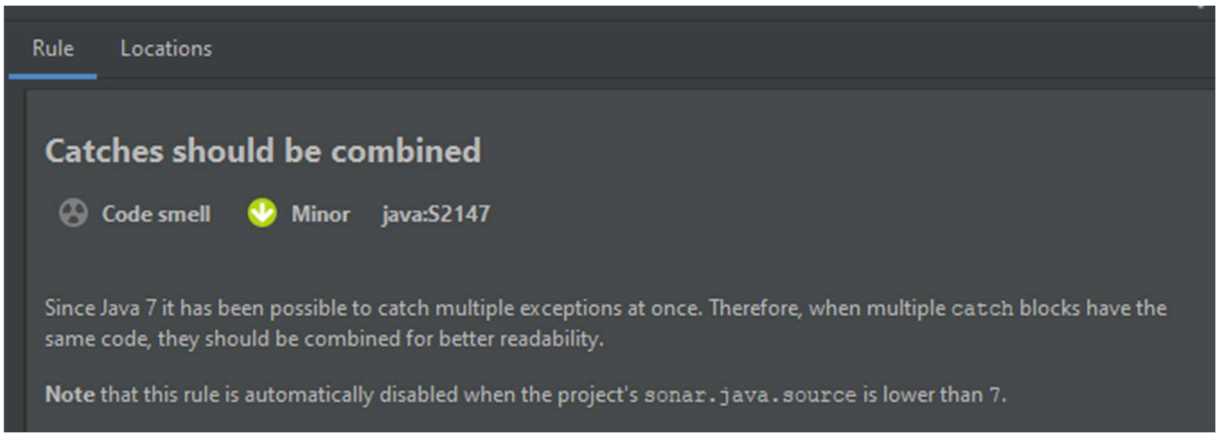
- Issues bevor refactoring: 892 in 76 files
- Issues after refactoring for Release 3: 60 in 22 files
- Issuers after refactoring for Final Release: 48 in 20 Files

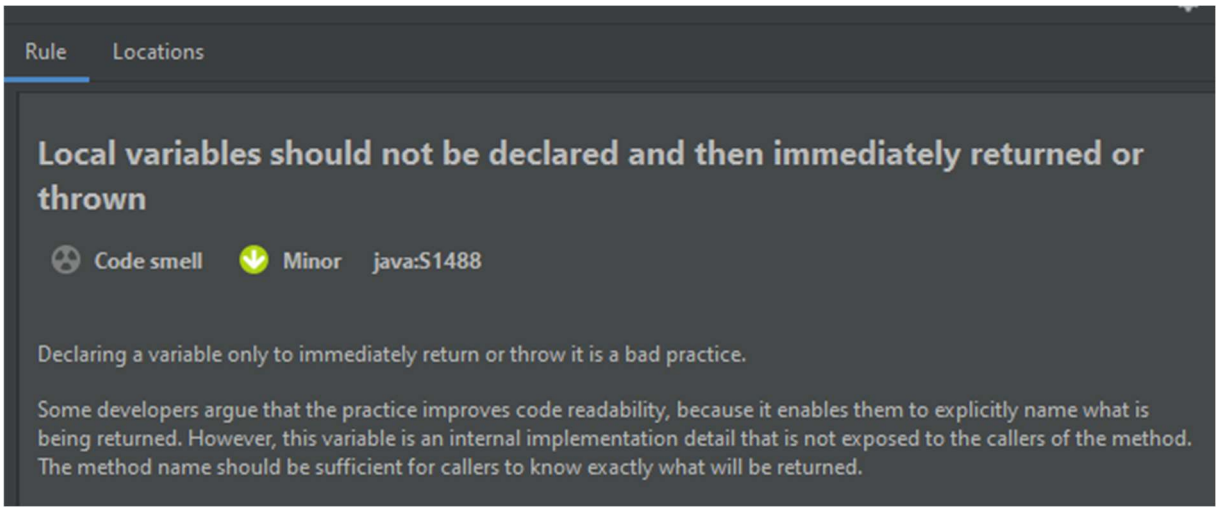
Most prominent issues:

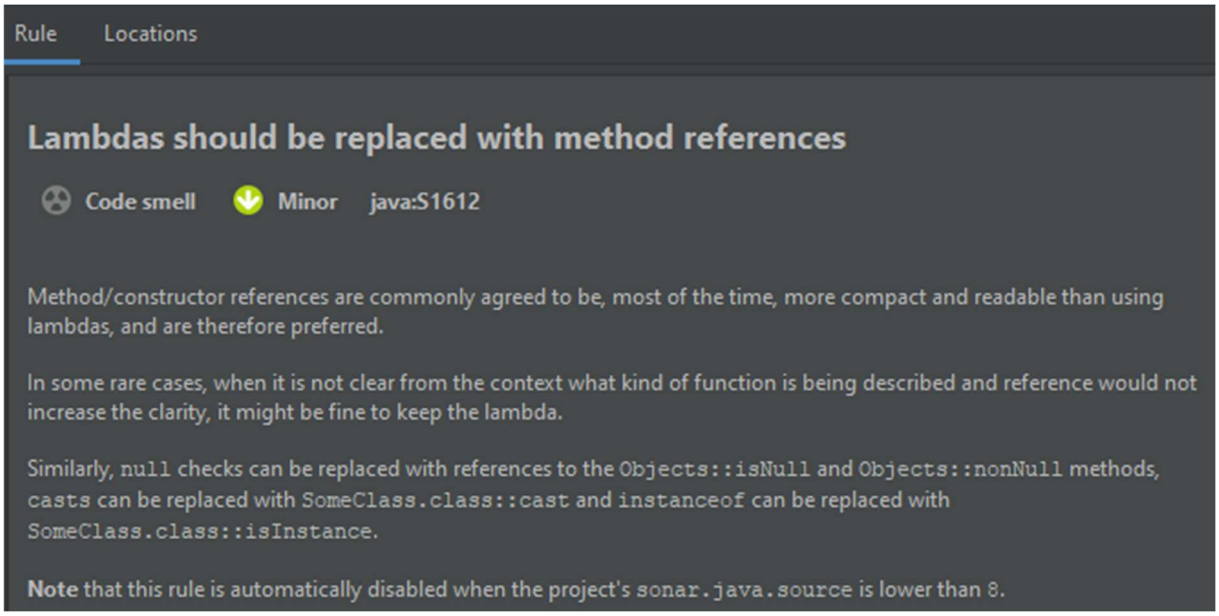


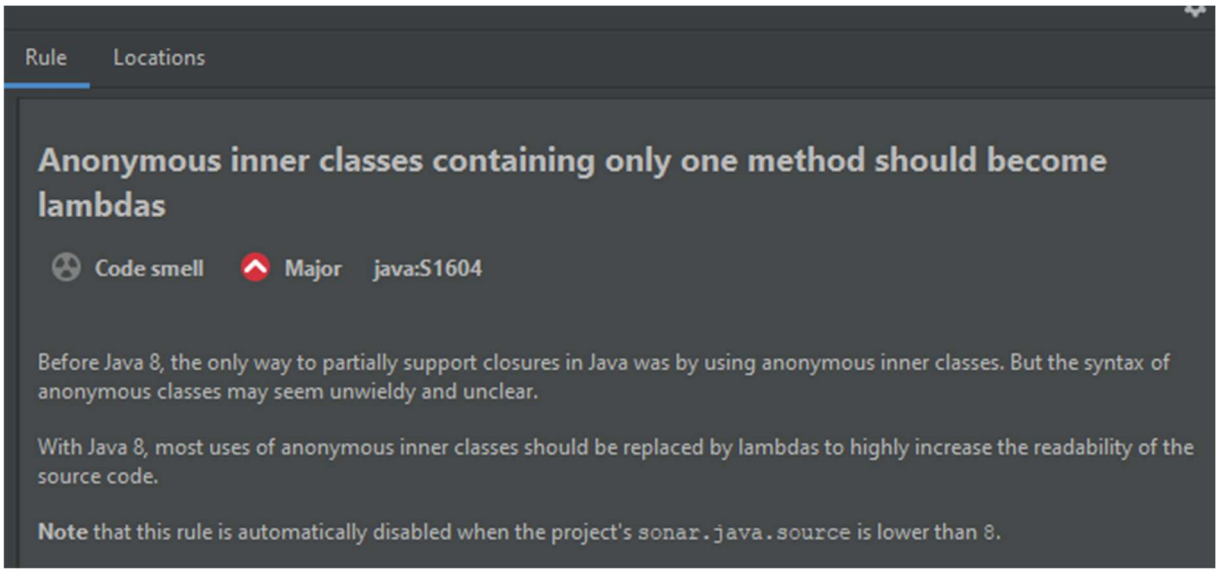
M

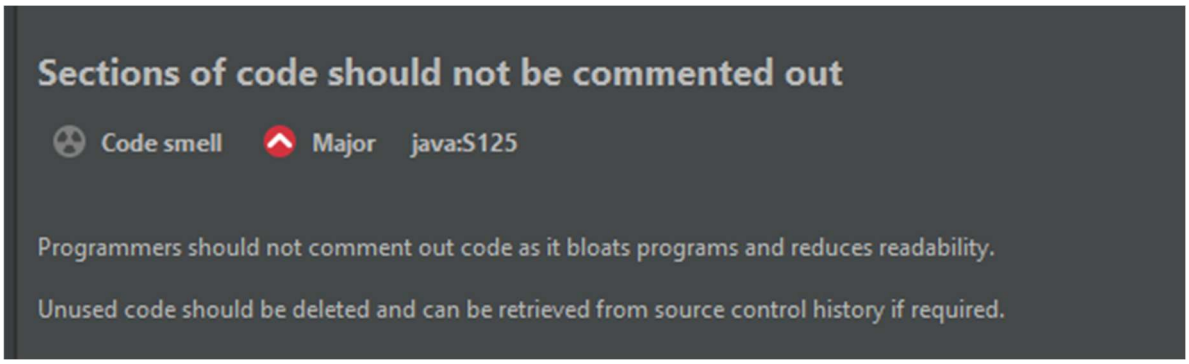


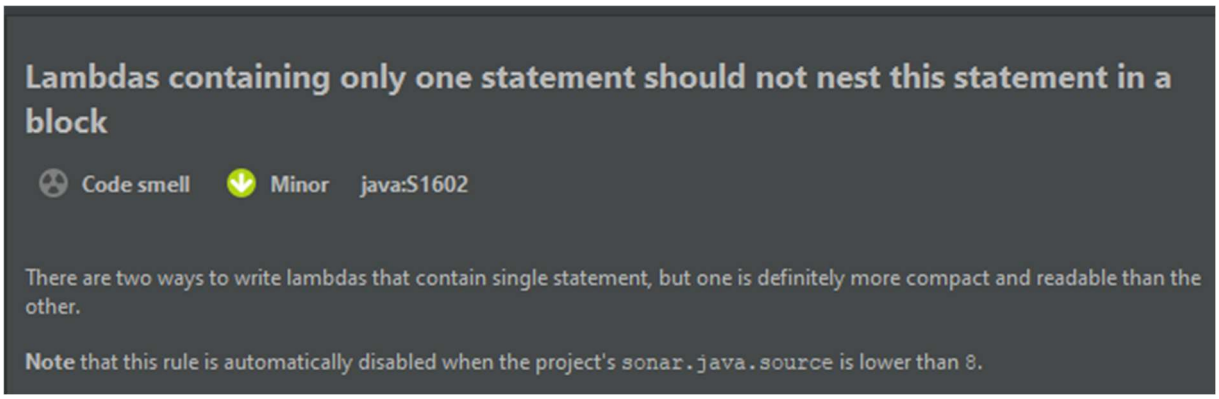

Rule
Locations
Catches should be combined
Code smell
Minor
java:S2147
Since Java 7 it has been possible to catch multiple exceptions at once. Therefore, when multiple catch blocks have the same code, they should be combined for better readability.
Note that this rule is automatically disabled when the project's sonar.java.source is lower than 7.

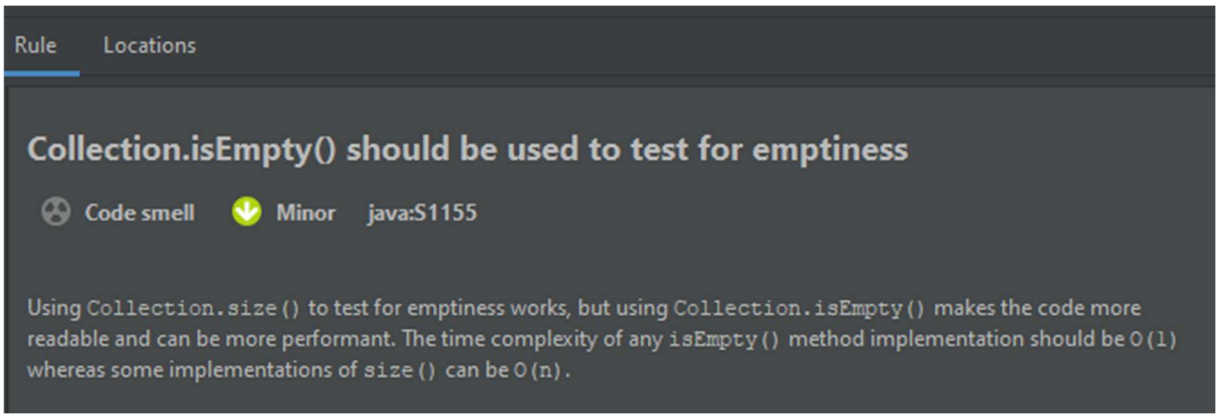

Rule
Locations
Local variables should not be declared and then immediately returned or thrown
Code smell
Minor
java:S1488
Declaring a variable only to immediately return or throw it is a bad practice.
Some developers argue that the practice improves code readability, because it enables them to explicitly name what is being returned. However, this variable is an internal implementation detail that is not exposed to the callers of the method. The method name should be sufficient for callers to know exactly what will be returned.

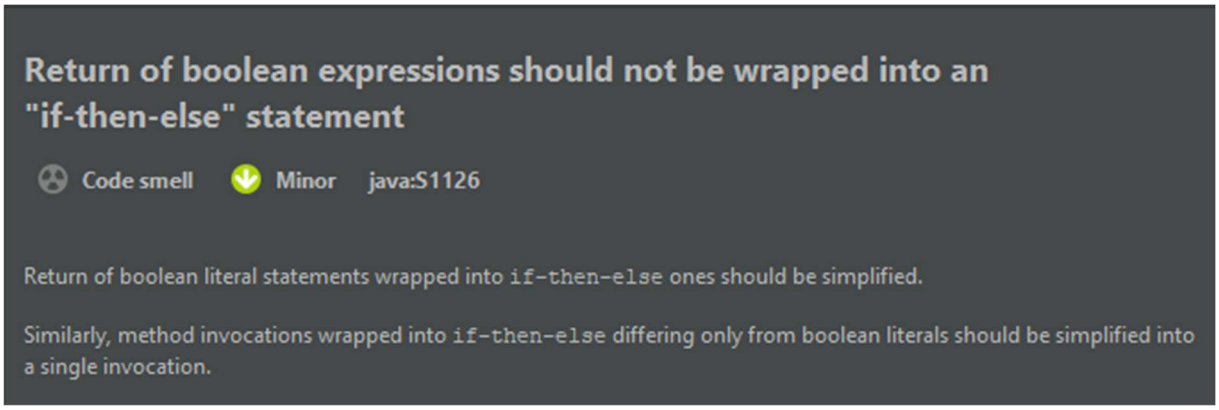

Rule
Locations
Lambdas should be replaced with method references
Code smell
Minor
java:S1612
Method/constructor references are commonly agreed to be, most of the time, more compact and readable than using lambdas, and are therefore preferred.
In some rare cases, when it is not clear from the context what kind of function is being described and reference would not increase the clarity, it might be fine to keep the lambda.
Similarly, null checks can be replaced with references to the Objects::isNull and Objects::nonNull methods, casts can be replaced with SomeClass.class::cast and instanceof can be replaced with SomeClass.class::isInstance.
Note that this rule is automatically disabled when the project's sonar.java.source is lower than 8.

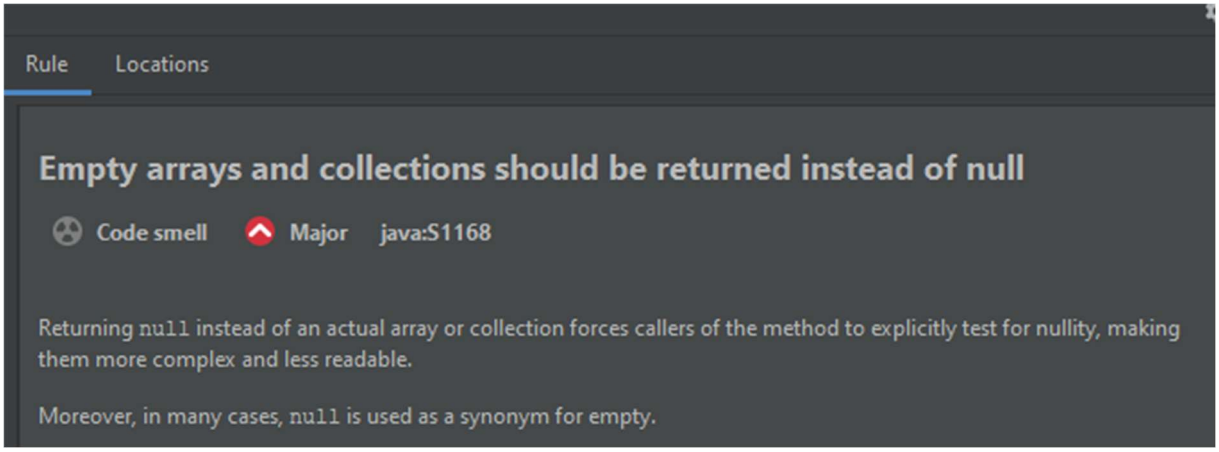

Rule
Locations
Anonymous inner classes containing only one method should become lambdas
Code smell
Major
java:S1604
Before Java 8, the only way to partially support closures in Java was by using anonymous inner classes. But the syntax of anonymous classes may seem unwieldy and unclear.
With Java 8, most uses of anonymous inner classes should be replaced by lambdas to highly increase the readability of the source code.
Note that this rule is automatically disabled when the project's sonar.java.source is lower than 8.

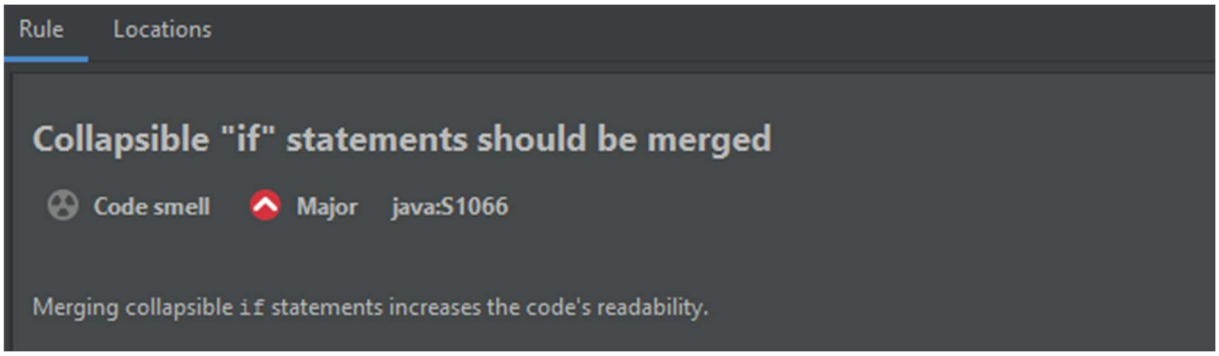

Sections of code should not be commented out
Code smell
Major
java:S125
Programmers should not comment out code as it bloats programs and reduces readability.
Unused code should be deleted and can be retrieved from source control history if required.

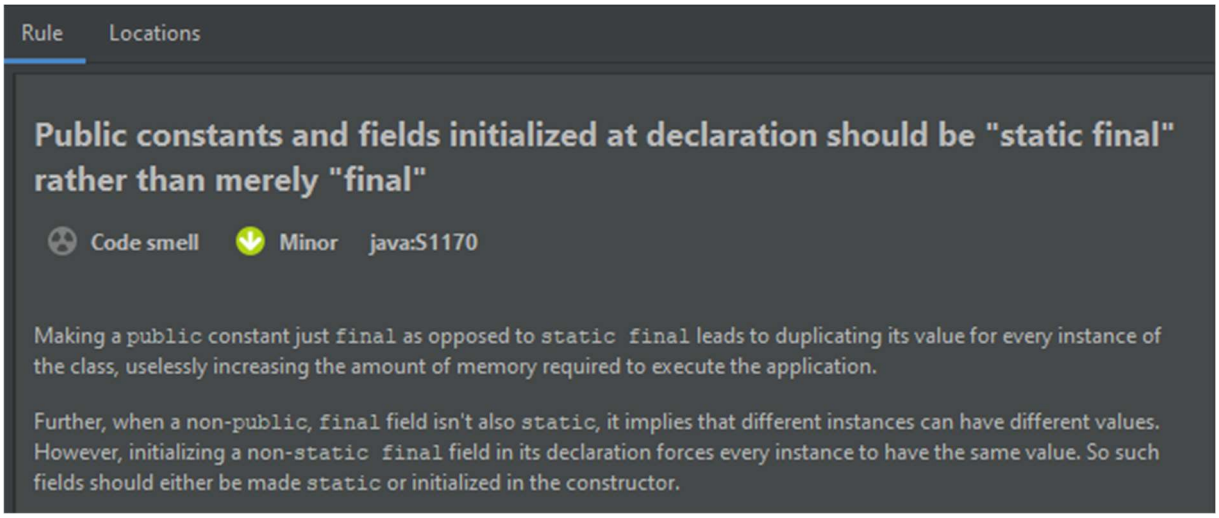

Lambdas containing only one statement should not nest this statement in a block
Code smell
Minor
java:S1602
There are two ways to write lambdas that contain single statement, but one is definitely more compact and readable than the other.
Note that this rule is automatically disabled when the project's sonar.java.source is lower than 8.

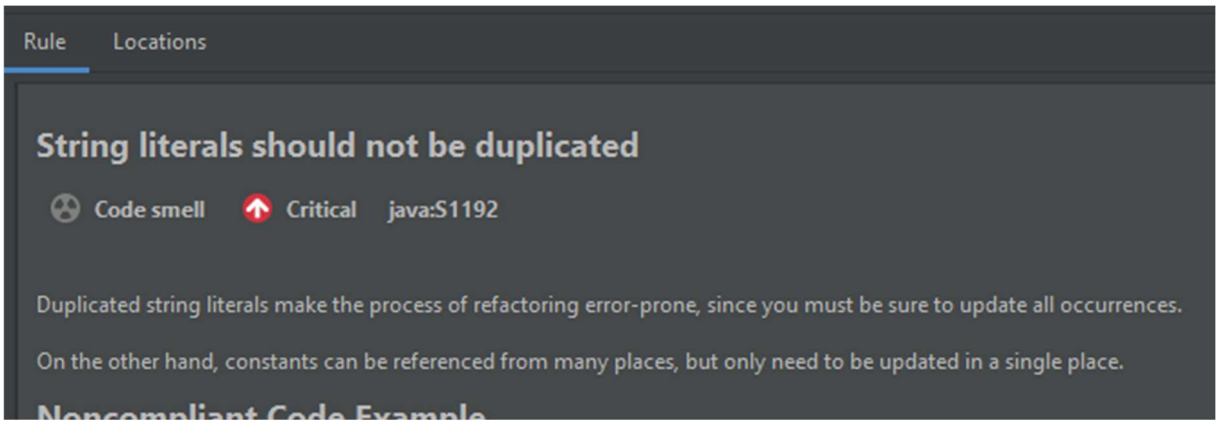

Rule
Locations
Collection.isEmpty() should be used to test for emptiness
Code smell
Minor
java:S1155
Using Collection.size() to test for emptiness works, but using Collection.isEmpty() makes the code more readable and can be more performant. The time complexity of any isEmpty() method implementation should be O(1) whereas some implementations of size() can be O(n).

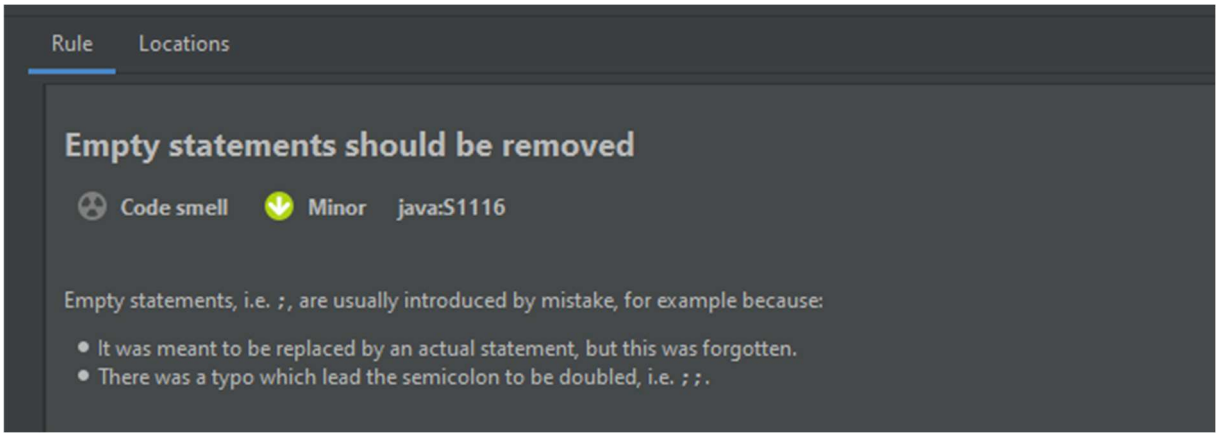

Return of boolean expressions should not be wrapped into an "if-then-else" statement
Code smell
Minor
java:S1126
Return of boolean literal statements wrapped into if-then-else ones should be simplified.
Similarly, method invocations wrapped into if-then-else differing only from boolean literals should be simplified into a single invocation.

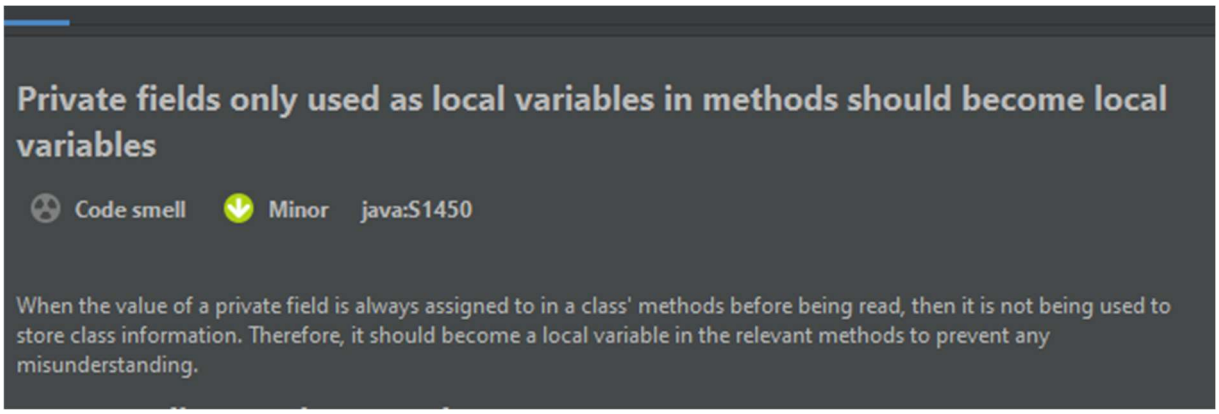

Rule
Locations
Empty arrays and collections should be returned instead of null
Code smell
Major
java:S1168
Returning null instead of an actual array or collection forces callers of the method to explicitly test for nullity, making them more complex and less readable.
Moreover, in many cases, null is used as a synonym for empty.

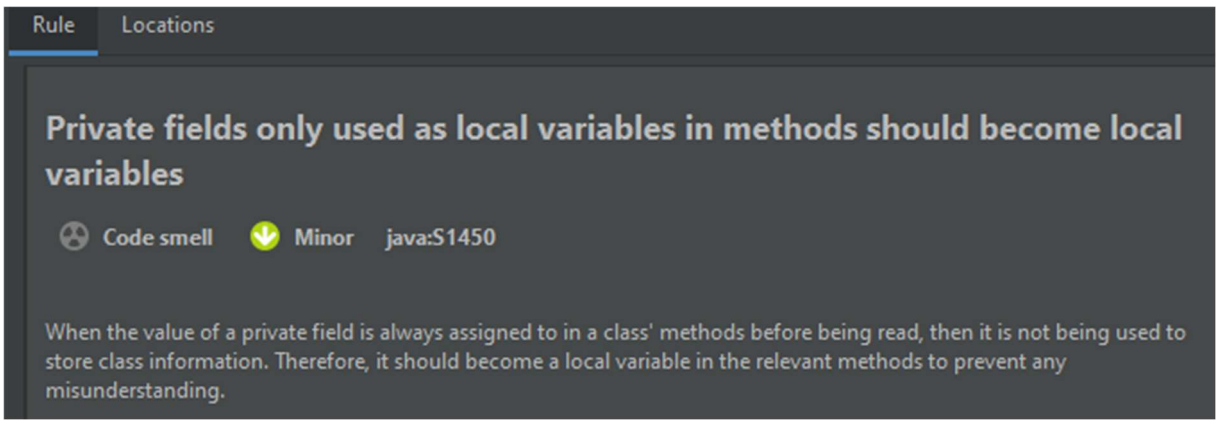

Rule
Locations
Collapsible "if" statements should be merged
Code smell
Major
java:S1066
Merging collapsible if statements increases the code's readability.

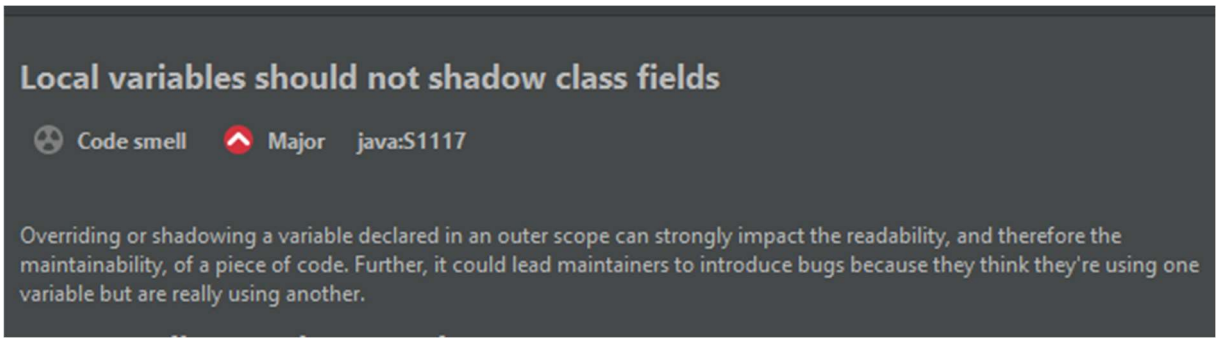

Rule
Locations
Public constants and fields initialized at declaration should be "static final" rather than merely "final"
Code smell
Minor
java:S1170
Making a public constant just final as opposed to static final leads to duplicating its value for every instance of the class, uselessly increasing the amount of memory required to execute the application.
Further, when a non-public, final field isn't also static, it implies that different instances can have different values. However, initializing a non-static final field in its declaration forces every instance to have the same value. So such fields should either be made static or initialized in the constructor.

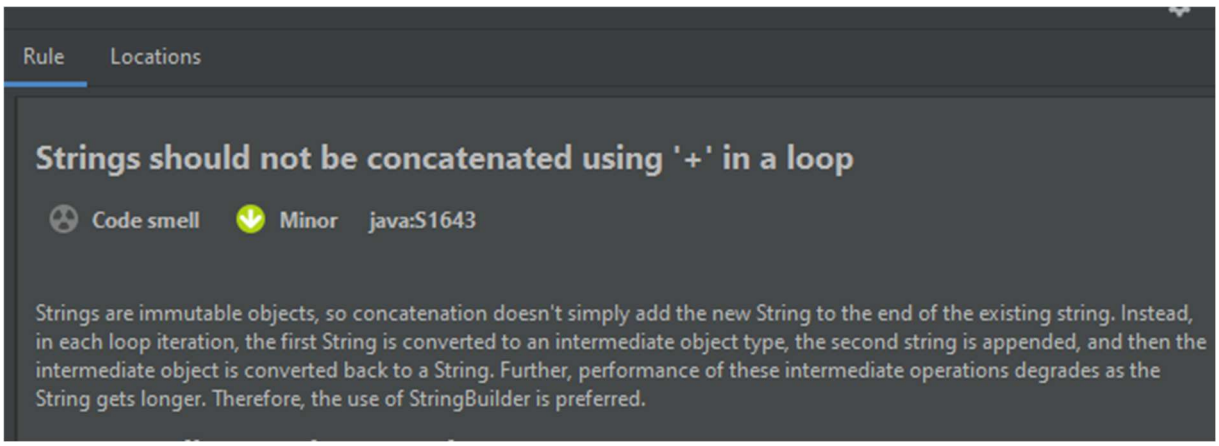

Rule
Locations
String literals should not be duplicated
Code smell
Critical
java:S1192
Duplicated string literals make the process of refactoring error-prone, since you must be sure to update all occurrences.
On the other hand, constants can be referenced from many places, but only need to be updated in a single place.

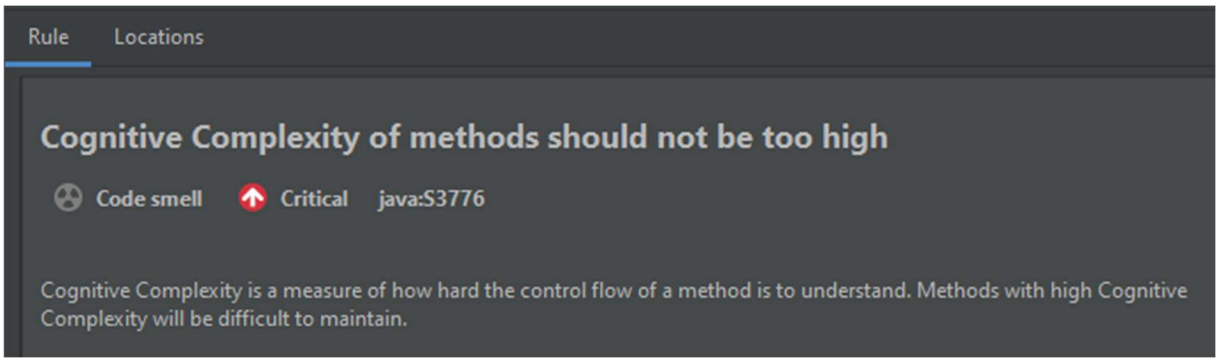

Rule
Locations
Empty statements should be removed
Code smell
Minor
java:S1116
Empty statements, i.e. ;, are usually introduced by mistake, for example because:
It was meant to be replaced by an actual statement, but this was forgotten.
There was a typo which lead the semicolon to be doubled, i.e. ;;.

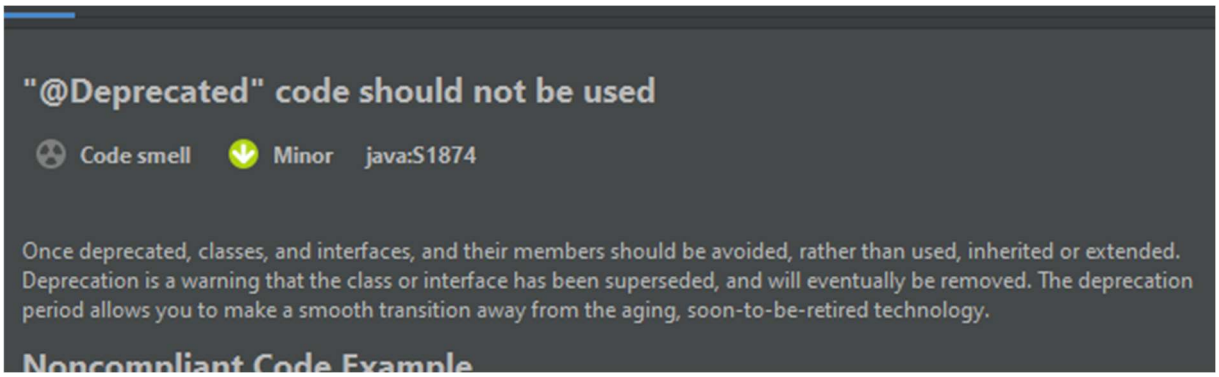

Private fields only used as local variables in methods should become local variables
Code smell
Minor
java:S1450
When the value of a private field is always assigned to in a class' methods before being read, then it is not being used to store class information. Therefore, it should become a local variable in the relevant methods to prevent any misunderstanding.

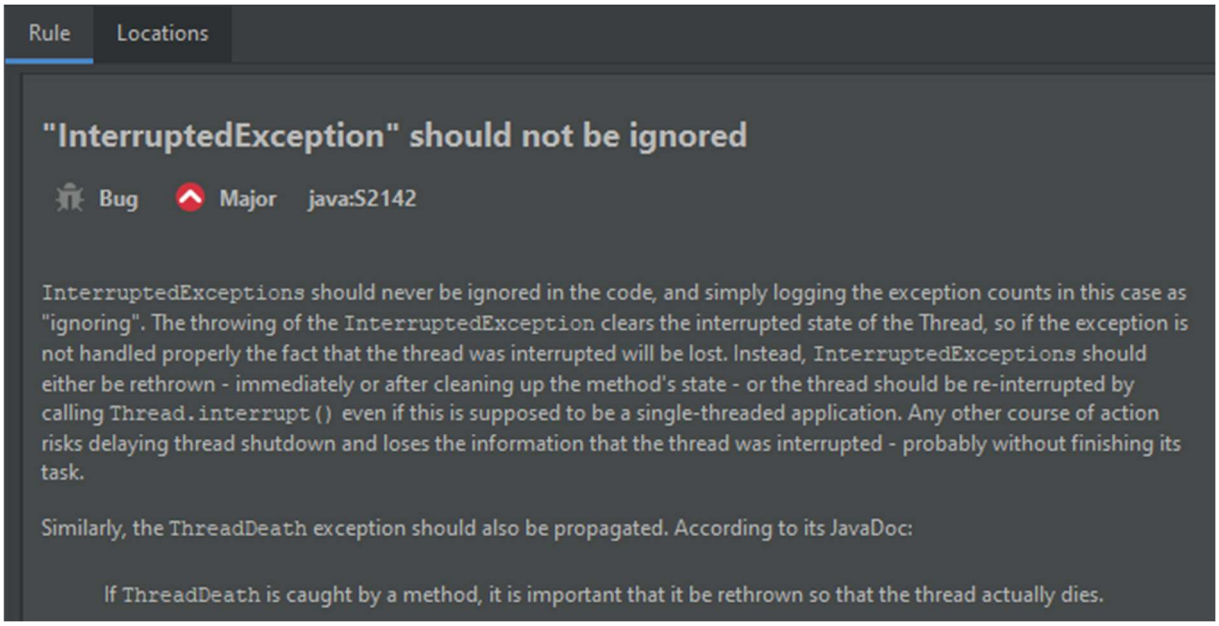

Rule
Locations
Private fields only used as local variables in methods should become local variables
Code smell
Minor
java:S1450
When the value of a private field is always assigned to in a class' methods before being read, then it is not being used to store class information. Therefore, it should become a local variable in the relevant methods to prevent any misunderstanding.


Local variables should not shadow class fields
Code smell
Major
java:S1117
Overriding or shadowing a variable declared in an outer scope can strongly impact the readability, and therefore the maintainability, of a piece of code. Further, it could lead maintainers to introduce bugs because they think they're using one variable but are really using another.


Rule
Locations
Strings should not be concatenated using '+' in a loop
Code smell
Minor
java:S1643
Strings are immutable objects, so concatenation doesn't simply add the new String to the end of the existing string. Instead, in each loop iteration, the first String is converted to an intermediate object type, the second string is appended, and then the intermediate object is converted back to a String. Further, performance of these intermediate operations degrades as the String gets longer. Therefore, the use of StringBuilder is preferred.

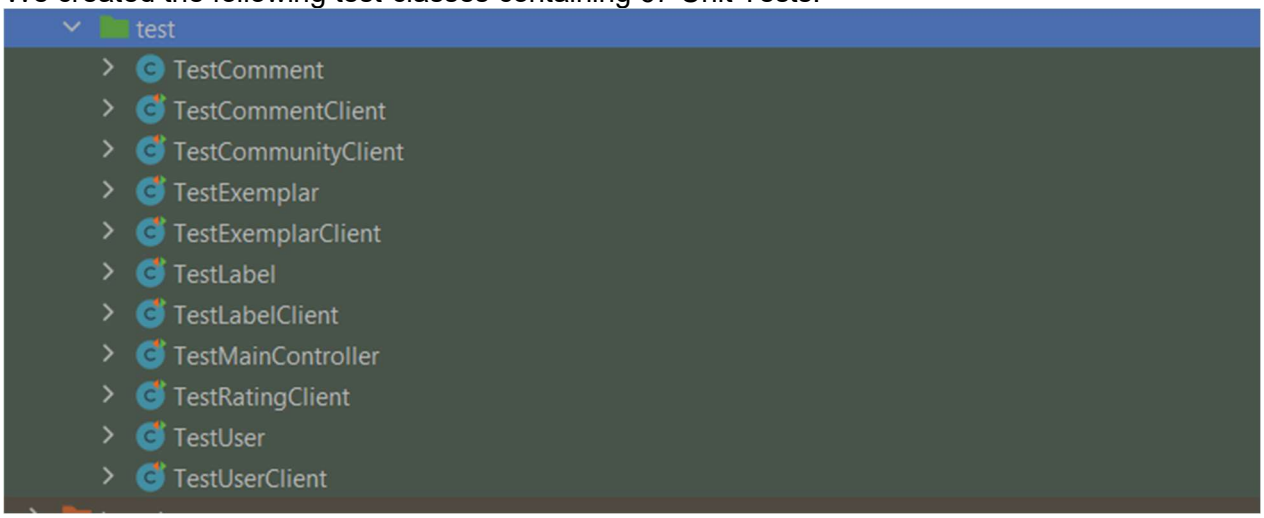

Rule
Locations
Cognitive Complexity of methods should not be too high
Code smell
Critical
java:S3776
Cognitive Complexity is a measure of how hard the control flow of a method is to understand. Methods with high Cognitive Complexity will be difficult to maintain.


"@Deprecated" code should not be used
Code smell
Minor
java:S1874
Once deprecated, classes, and interfaces, and their members should be avoided, rather than used, inherited or extended. Deprecation is a warning that the class or interface has been superseded, and will eventually be removed. The deprecation period allows you to make a smooth transition away from the aging, soon-to-be-retired technology.
Noncompliant Code Example

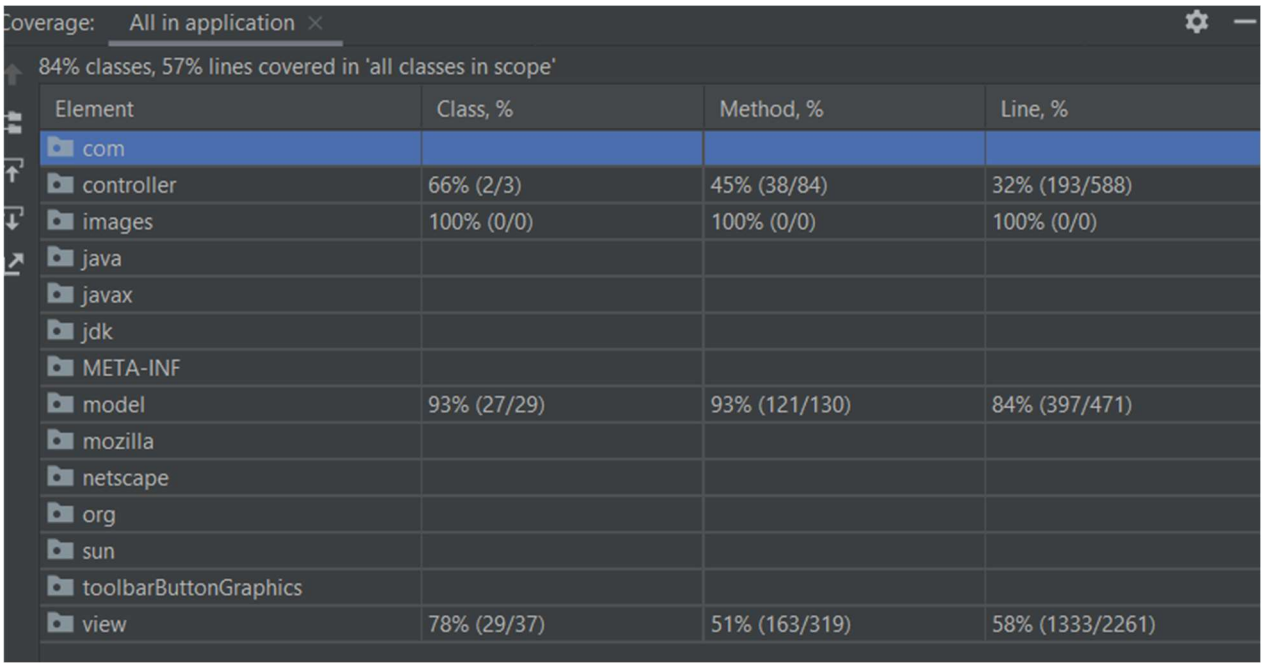

Rule
Locations
"InterruptedException" should not be ignored
Bug
Major
java:S2142
InterruptedExceptions should never be ignored in the code, and simply logging the exception counts in this case as "ignoring". The throwing of the InterruptedException clears the interrupted state of the Thread, so if the exception is not handled properly the fact that the thread was interrupted will be lost. Instead, InterruptedExceptions should either be rethrown - immediately or after cleaning up the method's state - or the thread should be re-interrupted by calling Thread.interrupt() even if this is supposed to be a single-threaded application. Any other course of action risks delaying thread shutdown and loses the information that the thread was interrupted - probably without finishing its task.
Similarly, the ThreadDeath exception should also be propagated. According to its JavaDoc:
If ThreadDeath is caught by a method, it is important that it be rethrown so that the thread actually dies.

4.4. Testing

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

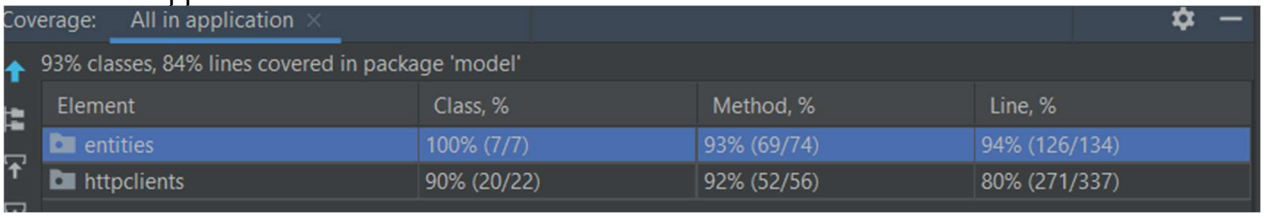


Our tests where mostly centered around the Http-Clients that are used to interact with the backend/ the database and connect all components. With this approached we achieved the following overall coverage:



Element	Class, %	Method, %	Line, %
com			
controller	66% (2/3)	45% (38/84)	32% (193/588)
images	100% (0/0)	100% (0/0)	100% (0/0)
java			
javax			
jdk			
META-INF			
model	93% (27/29)	93% (121/130)	84% (397/471)
mozilla			
netscape			
org			
sun			
toolbarButtonGraphics			
view	78% (29/37)	51% (163/319)	58% (1333/2261)

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



Element	Class, %	Method, %	Line, %
entities	100% (7/7)	93% (69/74)	94% (126/134)
httpclients	90% (20/22)	92% (52/56)	80% (271/337)

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

66% classes, 32% lines covered in package 'controller'

Element	Class, %	Method, %	Line, %
LoginController	100% (1/1)	81% (9/11)	72% (43/59)
MainController	50% (1/2)	39% (29/73)	28% (150/529)

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:

Coverage: All in application

78% classes, 58% lines covered in package 'view'

Element	Class, %	Method, %	Line, %
events	100% (1/1)	55% (5/9)	55% (10/18)
frames	93% (15/16)	59% (66/111)	81% (540/664)
listeners	100% (0/0)	100% (0/0)	100% (0/0)
panels	65% (13/20)	46% (92/199)	49% (783/1579)

Circle CI:

```
[INFO] Results:
[INFO]
[INFO] Tests run: 59, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] --- maven-jar-plugin:3.1.0:jar (default-jar) @ application ---
[INFO] Building jar: /home/circleci/project/target/application-1.0.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  01:06 min
[INFO] Finished at: 2021-07-12T17:13:20Z
[INFO] ------------------------------------------------------------------------
CircleCI received exit code 0
```

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	

5. Installation instruction

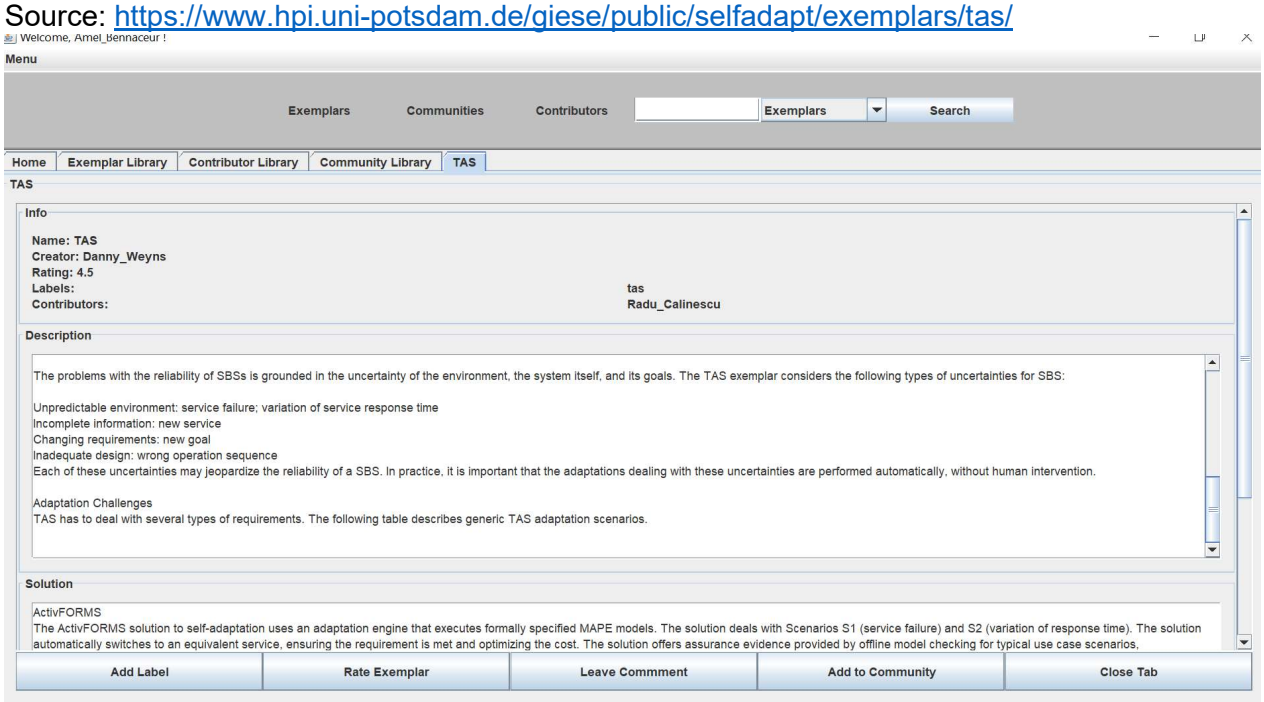
For a description on how to install and start the application we refer to our github repository, more specifically the README.md

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

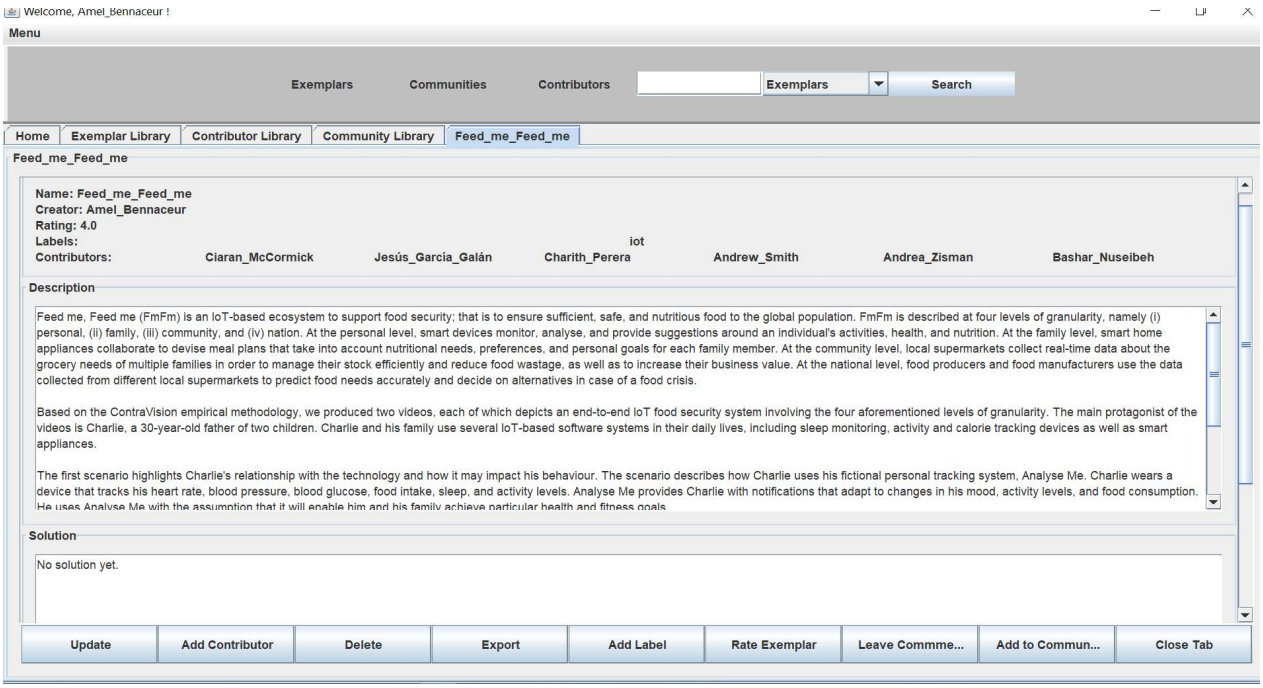
6. Screenshots of the provided common exemplars

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/