# [WeTok] Report

The following is a report template to help your team successfully provide all the details necessary for your report in a structured and organised manner. Please give a straightforward and concise report that best demonstrates your project. Note that a good report will give a better impression of your project to the reviewers.

*Here are some tips to write a good report:*

- *Try to summarise and list the* `bullet points` *of your project as much as possible rather than give long, tedious paragraphs that mix up everything together.*
- *Try to create* `diagrams` *instead of text descriptions, which are more straightforward and explanatory.*
- *Try to make your report* `well structured` *, which is easier for the reviewers to capture the necessary information.*

*We give instructions enclosed in square brackets [...] and examples for each sections to demonstrate what are expected for your project report.*

*Please remove the instructions or examples in* `italic` *in your final report.*

## Table of Contents

## Team Members and Roles

| UID | Name | Role |
|---|---|---|
| u6120911 | Xinyu Kang | Integration engineer, Data structure designer, Ranking algorithm designer |
| u7151386 | Xinyue Hu | Database Designer, Data structure designer, Integration engineer |
| u6684233 | Yuxin Hong | Tokenizer and parser designer and tester, Ranking algorithm tester |
| u6111569 | Zhaoting Jiang | UI Designer, Fragment designer |

## Conflict Resolution Protocol

1. Conflictors organize their views, list the pros and cons of their choice.
2. Initiate a zoom meeting with all team member.
3. Conflictors presenting the confliction and why disagree with the others.
4. Team member do a vote and share the reason.
5. As a result, merge conflictors's solutions or take more agreed solutions.

## Application Description

**Wetok is a social media app specifically aimed at person with a strong personality. You can share your status and mood with your friends, people in the same city, and even strangers anytime, anywhere. In here, you do not have to worry about cyber-violence and personal**

**abuse, we only provide you with a pure sharing platform. As our logo shows, we talk, but we don't comment. Furthermore, you can search for tags you're interested in and subscribe people that you're interested in. You can understand what is happening in this world through wetok. Welcome to Wetok.**

## Application Use Cases and or Examples

Targets Users: Everyone who is willing to use and learn about this app

Jack wants to learn about skateboarding skills that interest him most

1. Jack searched the tag about skateboarding and found that Kevin shared skateboarding skills every day
2. He followed Kevin
3. He learned new moves from Kevin's skateboard skills

Andy wants to know about the good restaurants around him

1. Andy turns on the City feature and finds that Claire shared a delicious restaurant nearby
2. He followed Claire and hope Claire can share more delicious restaurants in the future

*Here is a map navigation application example*

*Targets Users: Drivers*

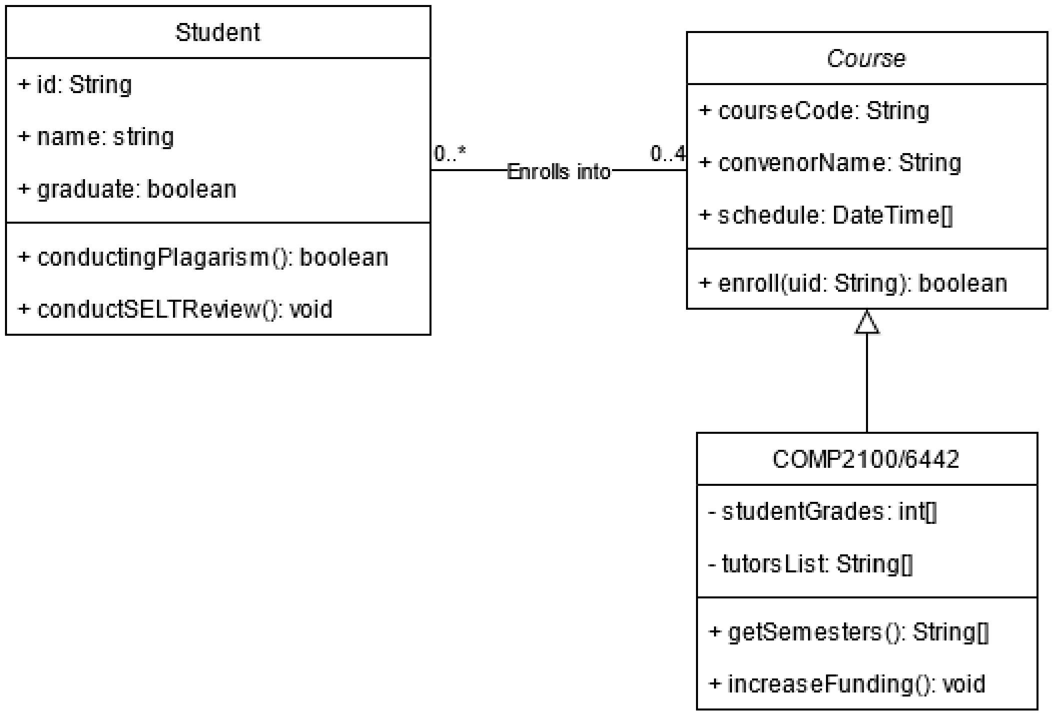
- *Users can use it to navigate in order to reach the destinations.*
- *Users can learn the traffic conditions*
- ...

*Target Users: Those who want to find some good restaurants*

- *Users can find nearby restaurants and the application can give recommendations*
- ...

*List all the use cases in text descriptions or create use case diagrams. Please refer to https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-use-case-diagram/ for use case diagram.*

## Application UML



*[Replace the above with a class diagram. You can look at how we have linked an image here as an example of how you can do it too.]*

## Application Design and Decisions

*Please give clear and concise descriptions for each subsections of this part. It would be better to list all the concrete items for each subsection and give no more than* `5` *concise, crucial reasons of your design. Here is an example for the subsection* `Data Structures` *:*

*I used the following data structures in my project:*

**Data Structures**

1. AVL Tree
    - *Objective: It is used for storing xxxx for xxx feature.*
    - *Locations: line xxx in XXX.java, ..., etc.*
    - *Reasons:*
        - *It is more efficient than Arraylist for insertion with a time complexity O(1)*
        - *We don't need to access the item by index for this feature*

**Design Patterns**

1. Singleton
    - Objective: It is used for making sure of that there is exactly one instance of the current user
    - Locations: *CurrentUser.java*

   - Reasons:
     - It can effectively avoid the case of multiple-current-users error
     - We need a class storing the current user instance that can be access from the whole project
2. Template
   - Objective: It is used for computing the ranking scores of the retrieved posts from different dimensions.
   - Locations: *ScoreTemplate.java* (abstract class), *RelevanceScore.java* (concrete class), *ImportanceScore.java* (concrete class), *UserSimilarity.java* (concrete class)
   - Reasons:
     - We want to sort the posts based on three criteria. There is a common process (i.e., calculate scores, normalize scores) to scoring the posts but each criterion has its own scoring logic.
     - The template method can clearly define the structure and make the code more readable and reuseable.
3. DAO
   - Objective: It is used for storing the users and posts data read from persistent files.
   - Locations: *UserDao.java*, *PostDao.java*
   - Reasons:
     - We want to decouple domain logic from persistence mechanisms and avoid exposing details of the data storage.
     - The DAO method allows JUnit test to run faster as it allows to create Mock and avoid connecting to database to run tests.

#### Grammars

- Objective:
- Locations:
- Reasons:

#### Tokenizer and Parsers

- Objective:
- Locations:
- Reasons:

#### Surpise Item

1. Ranking algorithm
   - Objective: It is used for ranking the retrieved posts
   - Locations: Package *ranking* : *Rank.java*, *ScoreTemplate.java*, *RelevanceScore.java*, *ImportanceScore.java*, *UserSimilarityScore.java*

    - Reasons:
        - This is an interesting feature and closely related to society
        - We want to rank the retrieved posts based on more than the post time
        - There are members in our group who have learned information retrieval
2. Simple personalisation
    - Objective: It is used for logging users' addresses to improve the timeline creation
    - Locations: Package *view* : Package *fragment* : "CityFragment.java*
    - Reasons:
        - This is the simplest way to personalise the timeline creation.
        - We have thought about maximising the chances of accurately providing search results or information in a timeline by using user posts, user interactions and so on.
        - However, in the ranking algorithm we considered the similarity between the current user of the sender of the posts as a negative factor of search ranking to break *Filter Bubbles*. We do not want to "offset" the effect of the ranking algorithm.
        - So, in our app the users can choose to enable (by click on *city* in the navigation bar) or unable the simple location personalisation according to their preferences .

## Data Structures

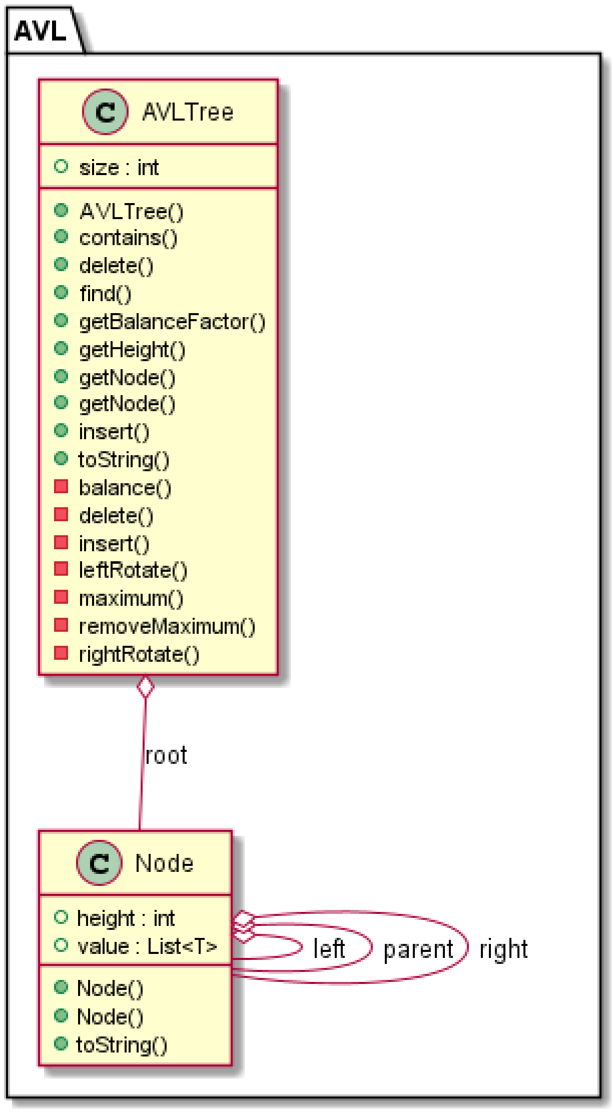
#### AVL Tree

We selected AVL tree as the data structure of posts operations. AVL tree is a self-balancing binary search tree. It controls the height of the tree and prevents it from becoming skewed. For operations considered they have time complexity:

- Rotations to achieve balanced tree *O(1)*
- Insertion *O(log(n))*
- Deletion *O(log(n))*
- Search *O(log(n))*
- Max/Min *O(log(n))*

It is an efficient data structure and here is the UML of our implementation:

AVL's Class Diagram



PlantUML diagram generated by SketchIt! (https://bitbucket.org/pmesmeur/sketch.it)
For more information about this tool, please contact philippe.mesmeur@gmail.com

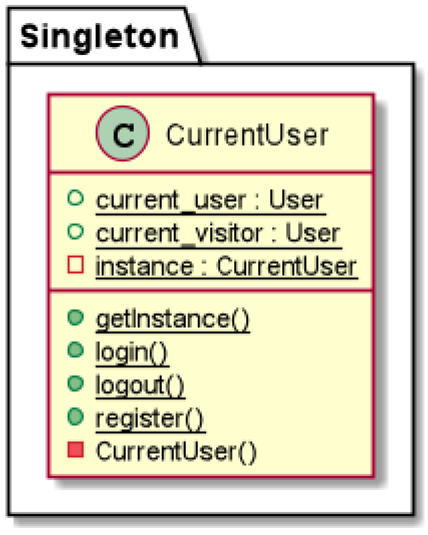
## Design Patterns

We used three design patterns: Singleton, Template, DAO

### Singleton

- Location: *CurrentUser.java*
- We used Singleton design pattern in CurrentUser to make sure there must be exactly one instance of the current user.

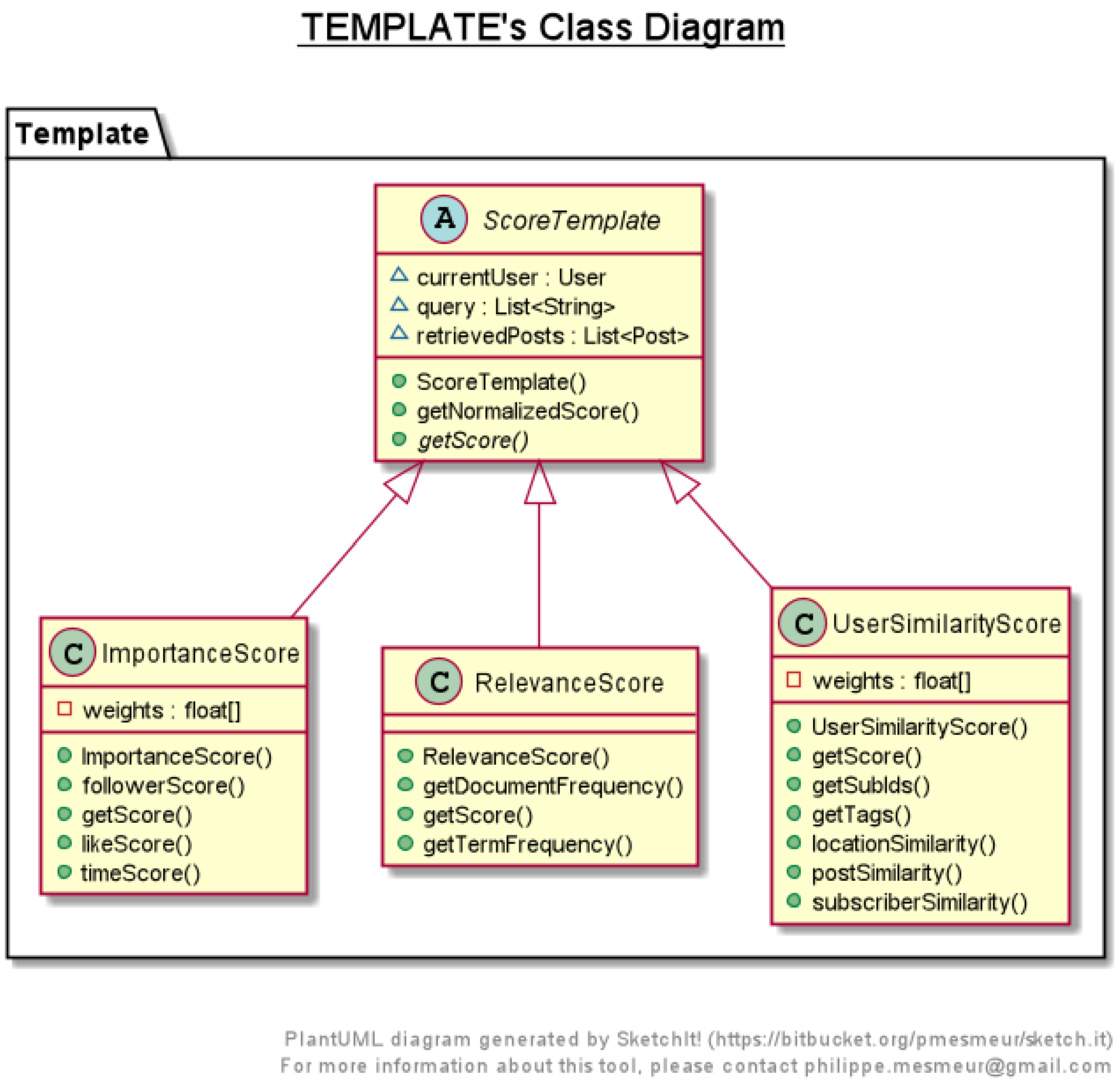
## SINGLETON's Class Diagram



PlantUML diagram generated by SketchIt! (https://bitbucket.org/pmesmeur/sketch.it)
For more information about this tool, please contact philippe.mesmeur@gmail.com

**Template**

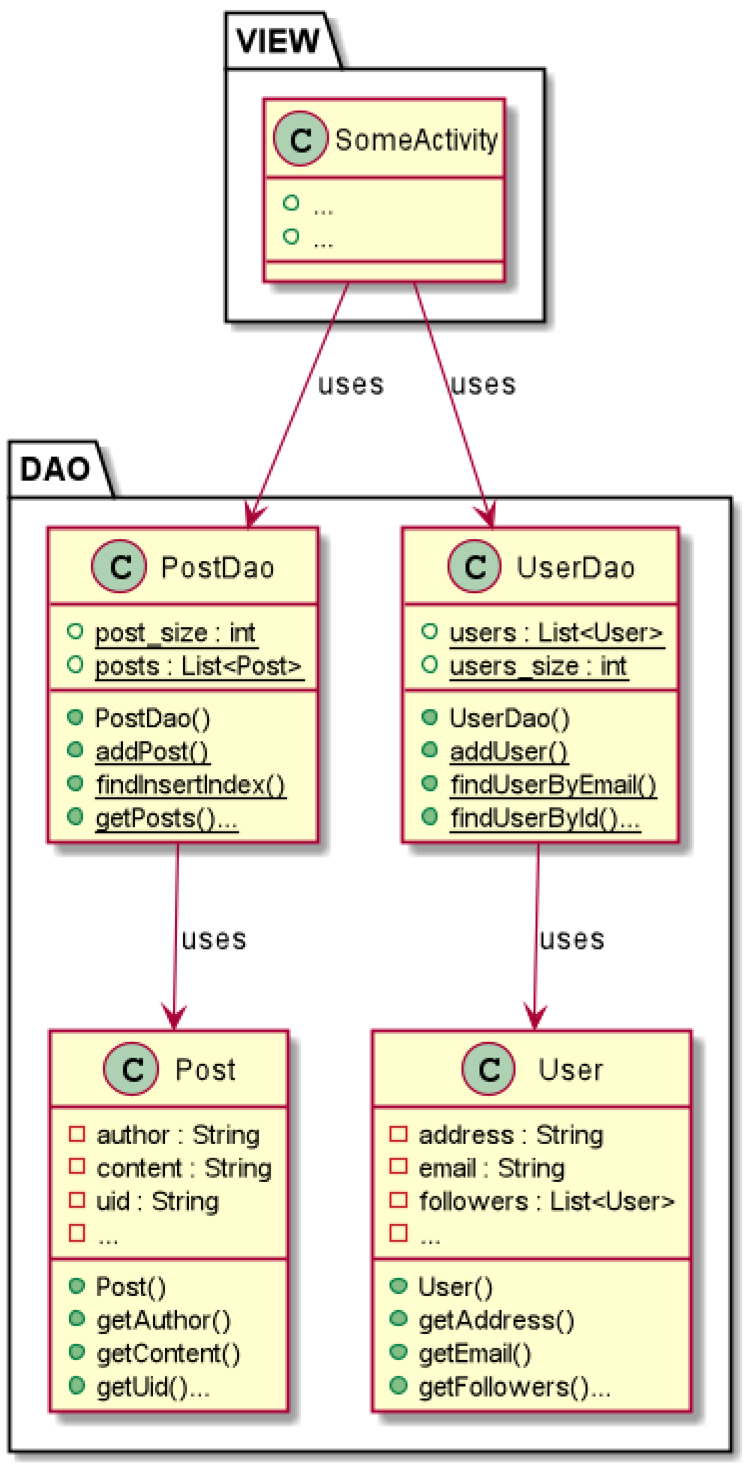
- Location: *ScoreTemplate.java* (abstract class), *RelevanceScore.java* (concrete class), *ImportanceScore.java* (concrete class), *UserSimilarity.java* (concrete class)
- We want to sort the posts based on three criteria. There is a common process (i.e., calculate scores, normalize scores) to scoring the posts but each criterion has its own scoring logic.
- So, we used a score template to define the skeleton of the scoring algotirthm, and let subclasses redefine certain steps of the algotirthm without changing the main structure.



**DAO**

- Location: *UserDao.java*, *PostDao.java*
- We want to decouple domain logic from persistence mechanisms and avoid exposing details of the data storage. The two DAO classes provided convenient access to user data in whole project.

DAO's Class Diagram



PlantUML diagram generated by SketchIt! (https://bitbucket.org/pmesmeur/sketch.it)
For more information about this tool, please contact philippe.mesmeur@gmail.com

## Grammars

*Production Rules*

<exp> ::= <term> | <term> '|' <exp>

<term> ::= <factor> | <factor> '&' <term>

<factor> ::= <tag> | '(' <exp> ')'

According to this gramma, we can parse *AND* and *OR* operations. For example, suppose we have the expression **condition1 & condition2 | condition3**, human will process it as **(condition1 & condition2) | condition3**, so does our gramma will do. Another example is **condition1 & (condition2 | condition3 & condition4)**, human will process it as **condition1 & (condition2 | (condition3 & condition4))**, so does our gramma will do. In application, the advantages of our gramma is to filter tag of post with *AND* and *OR* operation. It takes tag as its condition and only return post that satified given condition in expression.

## Tokenizer and Parsers

Generally speaking, the advantages of our design is we to search multiple tags at once and this is also the design approach of our gramma. Operations *AND* and *OR* can be aplied on multiple tags search, that is, the intersection and union of single search result. By search *#tag1&#tag2*, the intersection of individual search result of *#tag1* and *#tag2* will be returned. By search *#tag1|#tag2*, the union of of individual search result of *#tag1* and *#tag2* will be returned. And of course the operators can be freely used to create more expressions. By default, the precedence of *AND* operation is greater then *OR* operation. And precedence can be changed parentheses *LBRA* and *RBRA*.

## Surpise Item

### Ranking algorithm

Based on the *Pariser talk*, we decided to sort the posts by three criteria: relecvance; importance; user similarity.

- **Relevance**

  The class *RelevanceScore* evaluates the relecance of the query and the retrieved posts by calculating tf-idf scores. The score of post d given query q = (t_1, t_2, ..., t_m) is:

$$score_{tfidf}(d,q) = \sum_{i=1}^{m} tfidf_{t_i,d}$$

$$tfidf_{t,d} = tf_{t,d} \times idf_t$$

$$idf_t = log\frac{N}{1+df_t}$$

$df_t$ = the number of posts in all posts that contain a term $t$

$tf_{t,d}$ = the number of occurrences of tag $t$ in post $d$

$N$ = the total number of posts

We expect the posts containing more "rare words" that appeared in the query to be more relevant. The rare word here refers to the word that does not exist in many posts.

- **Importance**

  The class *ImportanceScore* evaluates the importance of the retrieved posts by calculating the post time, post likes, and the user's followers:

$$score_{\text{importance}} = w_{\text{time}} \times score_{\text{time}} + w_{\text{follower}} \times score_{\text{follower}} + w_{\text{like}} \times score_{\text{like}}$$

$$score_{\text{time}} = \frac{1}{1 + \text{\#days between query and post}}$$

$$score_{\text{follower}} = min(\frac{1}{4}log_{10}(\text{\#followers of post}), 0)$$

$$score_{\text{like}} = min(\frac{1}{4}log_{10}(\text{\#likes of post}), 0)$$

$$w_{\text{time}}, w_{\text{follower}}, w_{\text{like}} \text{are initially set to be } 0.7, 0.2, 0.1$$

  We expect the posts with more likes, the posts sent by users with more followers, and the latest posts to be more important to society, and thus have higher ranking scores.

- **User Similarity**

  The class *UserSimilarityScore* evaluates the similarity of the current user and the senders of the posts by evaluating their subscribers, posts, and addresses:

$$score_{\text{similarity}} = w_{\text{subscriber}} \times score_{\text{subscriber}} + w_{\text{post}} \times score_{\text{post}} + w_{\text{address}} \times score_{\text{address}}$$

$$score_{\text{subscriber}} = \frac{1}{1 + \text{\#common subscribers}}$$

$$score_{\text{post}} = \frac{1}{1 + \text{\#common tags from past posts}}$$

$$score_{\text{address}} = 1 \text{ if current user and post sender live in same area, } = 0 \text{ otherwise}$$

$$w_{\text{subscriber}}, w_{\text{post}}, w_{\text{address}} \text{are initially set to be } 0.5, 0.4, 0.1$$

The post sent by the user who is considered as more 'similar' with the current user will have a lower ranking score.We expect to use this algotithm to offset some negative effects of the *'Filter Bubbles'*.

- **Overall Score**

$$score = w_{\text{tfidf}} \times score_{\text{tfidf}} + w_{\text{importance}} \times score_{\text{importance}} + w_{\text{similarity}} \times score_{\text{similarity}}$$

$$w_{\text{tfidf}}, w_{\text{importance}}, w_{\text{similarity}} \text{are initially set to be } 0.6, 0.2, 0.2$$

All of the weights introduced in this algorithm were intuitively chose by our group members. In practice we should use machine learning techniques to train the best weights.

We expect that the posts having strong relevance with the query, the posts which are important to society, and the posts sent by the users who are "different" with the user, to have better ranking in the search results presenting.

More and more intelligent recommendation systems create the *'Filter Bubbles'* for users. We intend to "break" the bubble by implementing the ranking algorithm from users similarity dimension. The score of the posts created by the users who are very "similar" with you will be scaled down. People will have more chances to receive different viewpoints from social media.

### Simple personalisation

*In order to create unique personalization features, we recorded the location of the user while recording the user login data. Based on the user's location, we added a search function, which is the city search function. The local search function searches for all posts in the same city based on the current user's location. This will make it easier for users to see what's going on around them.*

## Summary of Known Errors and Bugs

*[Where are the known errors and bugs? What consequences might they lead to?]*

*Here is an example:*

1. *Bug 1:*

- *A space bar (' ') in the sign in email will crash the application.*
- ...

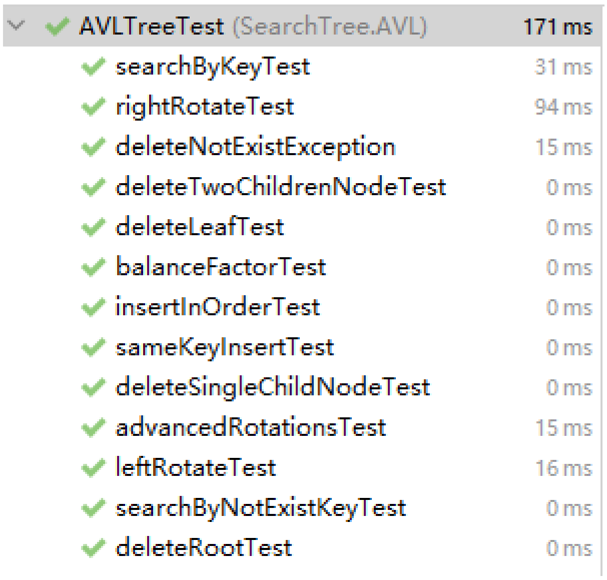
2. *Bug 2:*
3. ...

*List all the known errors and bugs here. If we find bugs/errors that your team do not know of, it shows that your testing is not through.*

## Testing Summary

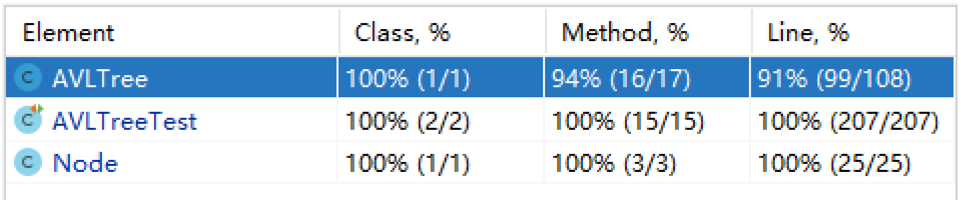
### AVL Tree Testings

Number of test cases: 13



```
AVLTreeTest (SearchTree.AVL)     171 ms
    searchByKeyTest               31 ms
    rightRotateTest               94 ms
    deleteNotExistException       15 ms
    deleteTwoChildrenNodeTest      0 ms
    deleteLeafTest                 0 ms
    balanceFactorTest              0 ms
    insertInOrderTest              0 ms
    sameKeyInsertTest              0 ms
    deleteSingleChildNodeTest      0 ms
    advancedRotationsTest         15 ms
    leftRotateTest                16 ms
    searchByNotExistKeyTest        0 ms
    deleteRootTest                 0 ms
```

Code coverage:



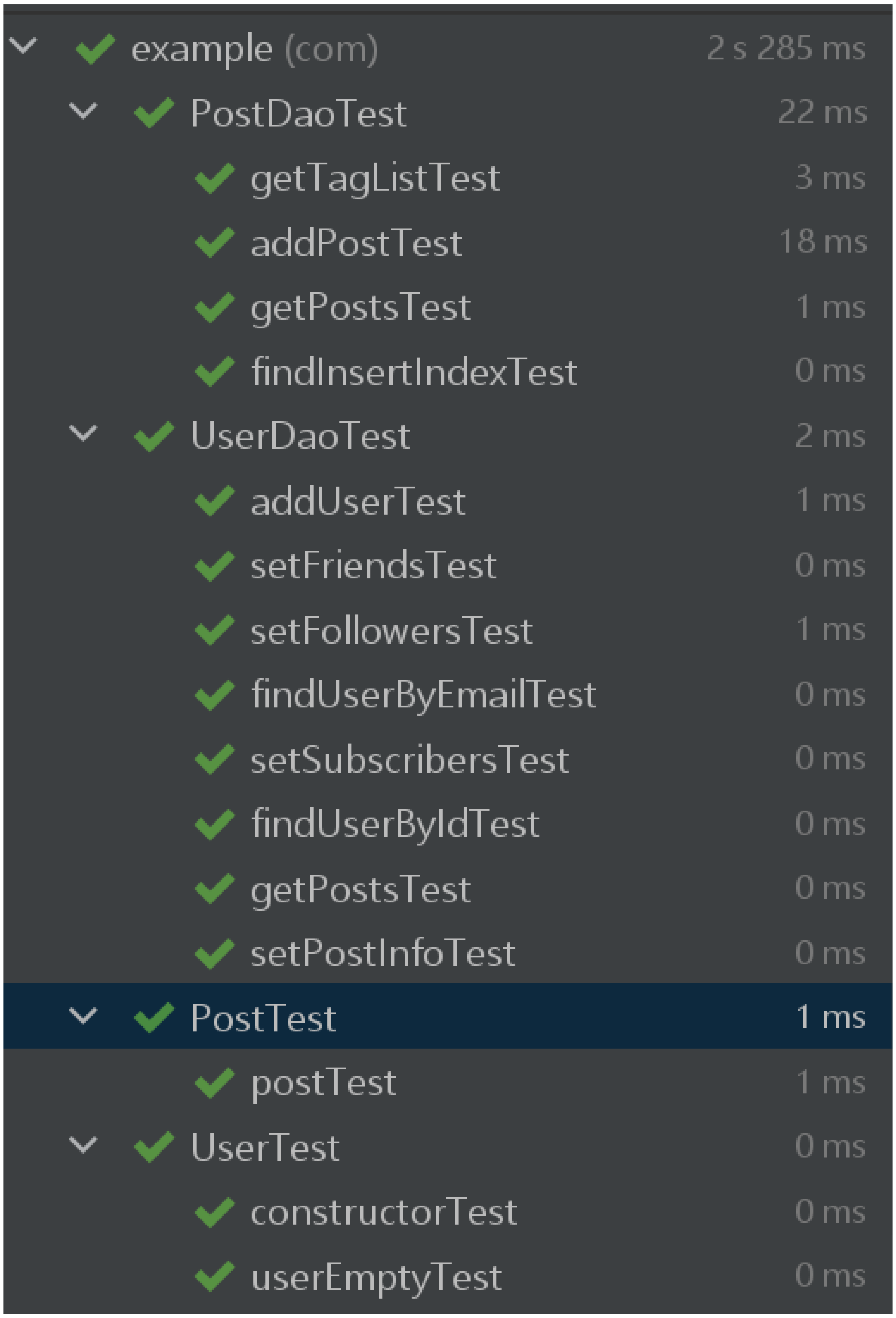
| Element | Class, % | Method, % | Line, % |
|---|---|---|---|
| AVLTree | 100% (1/1) | 94% (16/17) | 91% (99/108) |
| AVLTreeTest | 100% (2/2) | 100% (15/15) | 100% (207/207) |
| Node | 100% (1/1) | 100% (3/3) | 100% (25/25) |

Types of tests created:

1. *sameKeyInsertTest* : test whether the AVL tree can insert multiple values into one nodes
2. *insertInOrderTest* : test whether the AVL tree can correctly insert nodes in order
3. *leftRotateTest* : test whether the AVL tree can correctly left rotate
4. *rightRotateTest* : test whether the AVL tree can correctly right rotate
5. *balanceFactorTest* : test whether the class can generate correct balance factors
6. *advancedRotationsTest* : test whether the AVL tree can handle complex rotations
7. *deleteNotExistException* : test whether the AVL tree can throw an exception when deleting a node that not exists
8. *deleteLeafTest* : test whether the AVL tree can delete a leaf node
9. *deleteSingleChildNodeTest* : test whether the AVL tree can correctly delete a node with one child
10. *deleteTwoChildrenNodeTest* : test whether the AVL tree can correctly delete a node with two children
11. *deleteRootTest* : test whether the AVL tree can correctly delete the root node
12. *searchByKeyTest* : test whether the AVL tree can correctly find the node based on the key
13. *searchByNotExistKeyTest* : test whether the AVL tree returns null when searching a key that not exists

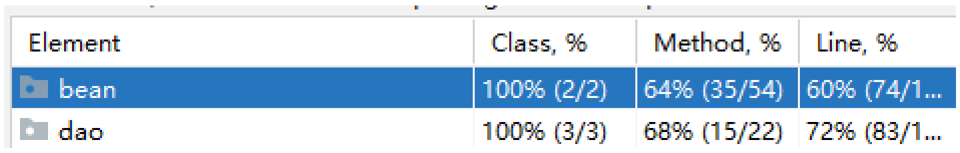
## Dao and Bean Testings

Number of test cases: 14



```
✓ example (com)              2 s 285 ms
  ✓ PostDaoTest                   22 ms
      ✓ getTagListTest             3 ms
      ✓ addPostTest               18 ms
      ✓ getPostsTest               1 ms
      ✓ findInsertIndexTest        0 ms
  ✓ UserDaoTest                    2 ms
      ✓ addUserTest                1 ms
      ✓ setFriendsTest             0 ms
      ✓ setFollowersTest           1 ms
      ✓ findUserByEmailTest        0 ms
      ✓ setSubscribersTest         0 ms
      ✓ findUserByIdTest           0 ms
      ✓ getPostsTest               0 ms
      ✓ setPostInfoTest            0 ms
  ✓ PostTest                       1 ms
      ✓ postTest                   1 ms
  ✓ UserTest                       0 ms
      ✓ constructorTest            0 ms
      ✓ userEmptyTest              0 ms
```

Code coverage:



| Element | Class, % | Method, % | Line, % |
|---|---|---|---|
| bean | 100% (2/2) | 64% (35/54) | 60% (74/1... |
| dao | 100% (3/3) | 68% (15/22) | 72% (83/1... |

Tests created for Bean:

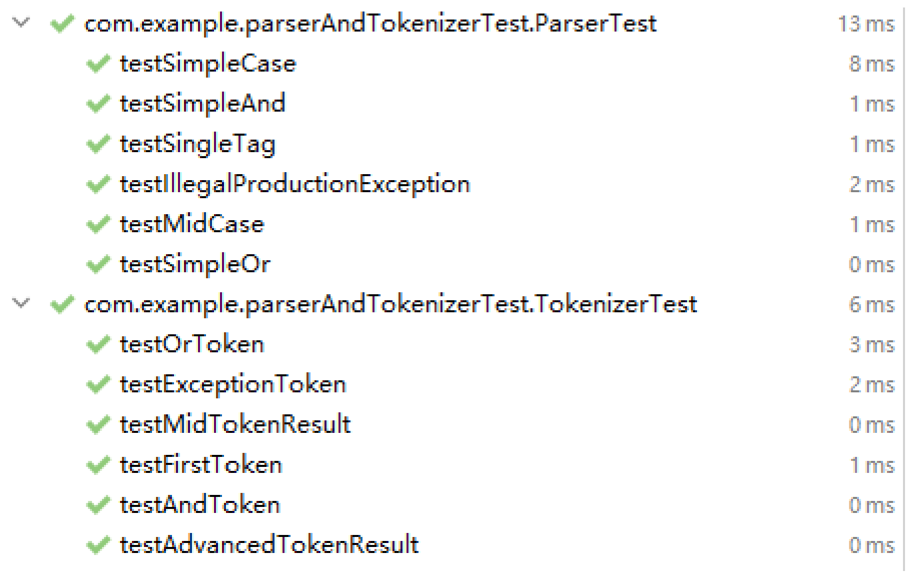
1. *userEmptyTest* : test whether all the setter and getter in User class are valid.
2. *constructorTest* : test whether the constructor in User class is valid.

Tests created for Dao:

1. *getTagListTest* : test whether the PostDao can get all the tags in posts.
2. *findInsertIndexTest* : test whether the PostDao can find the right place to insert post.
3. *addPostTest* : test whether the PostDao can add a post to database.
4. *getPostsTest* : test whether the PostDao can get all the posts in database.
5. *findUserByEmailTest* : test whether the UserDao can find the right user by it's email.
6. *findUserByIdTest* : test whether the UserDao can find the right user by it's user id.
7. *setFriendsTest* : test whether the UserDao can set friends for corresponding user.
8. *setFollowersTest* : test whether the UserDao can set followers for corresponding user.
9. *setSubscribersTest* test whether the UserDao can set subscribers for corresponding user.
10. *getPostsTest* : test whether the UserDao can get all the posts of each user.
11. *setPostInfoTest* : test whether the UserDao can instantiate each user's post.
12. *addUserTest* : test whether the UserDao can add user to database.

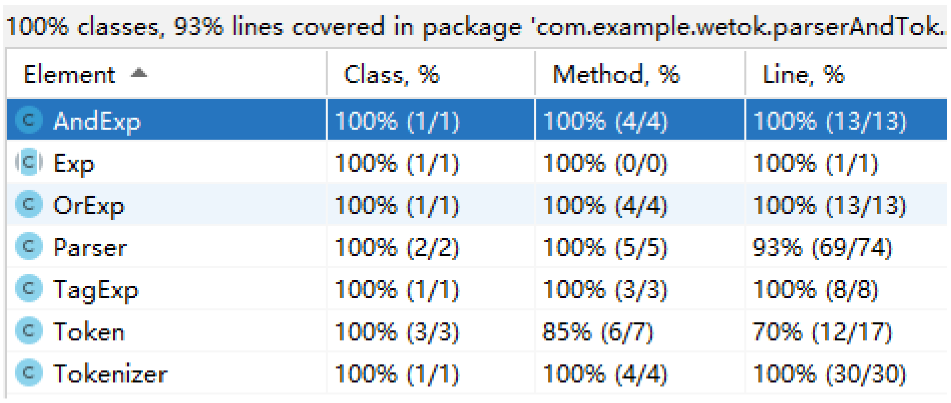
## Parser and Tokenizer Tests

Number of test cases: 12



```
✓ com.example.parserAndTokenizerTest.ParserTest          13 ms
    ✓ testSimpleCase                                      8 ms
    ✓ testSimpleAnd                                       1 ms
    ✓ testSingleTag                                       1 ms
    ✓ testIllegalProductionException                      2 ms
    ✓ testMidCase                                         1 ms
    ✓ testSimpleOr                                        0 ms
✓ com.example.parserAndTokenizerTest.TokenizerTest        6 ms
    ✓ testOrToken                                         3 ms
    ✓ testExceptionToken                                  2 ms
    ✓ testMidTokenResult                                  0 ms
    ✓ testFirstToken                                      1 ms
    ✓ testAndToken                                        0 ms
    ✓ testAdvancedTokenResult                             0 ms
```

Code coverage:



100% classes, 93% lines covered in package 'com.example.wetok.parserAndTok.

| Element | Class, % | Method, % | Line, % |
|---|---|---|---|
| AndExp | 100% (1/1) | 100% (4/4) | 100% (13/13) |
| Exp | 100% (1/1) | 100% (0/0) | 100% (1/1) |
| OrExp | 100% (1/1) | 100% (4/4) | 100% (13/13) |
| Parser | 100% (2/2) | 100% (5/5) | 93% (69/74) |
| TagExp | 100% (1/1) | 100% (3/3) | 100% (8/8) |
| Token | 100% (3/3) | 85% (6/7) | 70% (12/17) |
| Tokenizer | 100% (1/1) | 100% (4/4) | 100% (30/30) |

Tests created for Tokenizer:

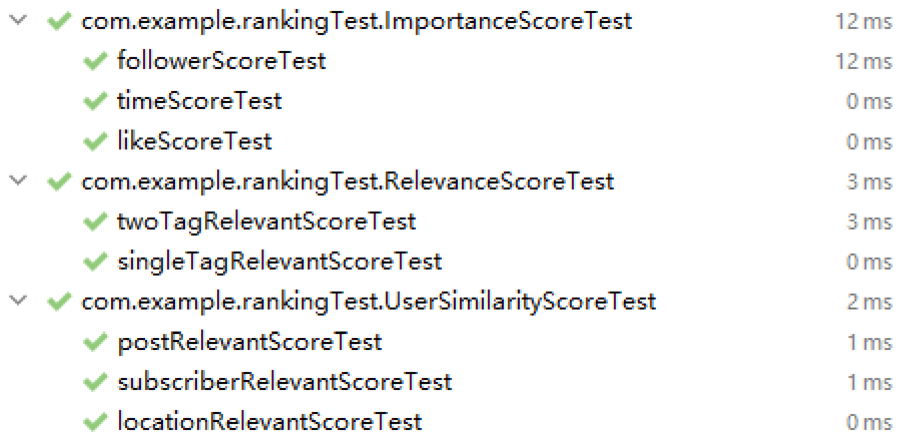
1. *testAndToken* : test whether *AND* token can be recogize.
2. *testOrToken* : test whether *OR* token can be recogize.
3. *testFirstToken* : test whether first *LBRA* token can be recogize.
4. *testMidTokenResult* : test whether *TAG* and *OR* token at middle can be recogize.
5. *testAdvancedTokenResult* : test whether tokens from *(#weekend* | *#mood) & #time* can be tokenized.
6. *testExceptionToken* : test whether tokenizer throw exceptions as expected.

Tests created for Parser:

1. *testSingleTag* : test whether single-tag-search is correct.
2. *testSimpleAnd* : test whether search tag with *AND* operation is correct.
3. *testSimpleOr* : test whether search tag with *OR* operation is correct.
4. *testSimpleCase* : test whether search tag with multiple *AND* operations is correct.
5. *testMidCase* : test whether search tag with *AND* and *OR* operations is correct.
6. *testIllegalProductionException* : test whether parser throw exceptions as expected.

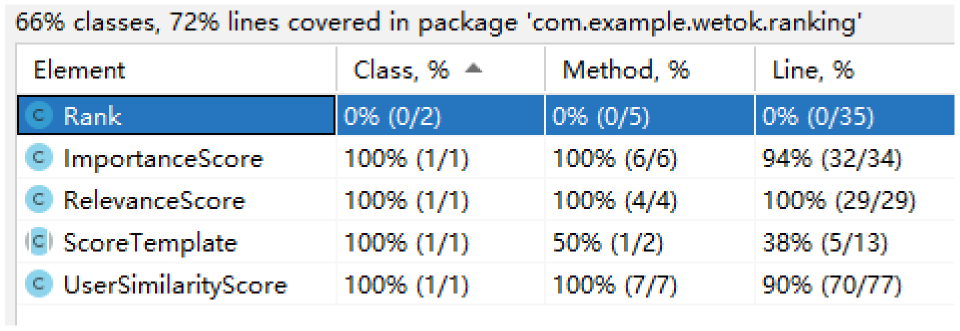
## Ranking Tests

Number of test cases: 8



```
✓ com.example.rankingTest.ImportanceScoreTest        12 ms
    ✓ followerScoreTest                               12 ms
    ✓ timeScoreTest                                    0 ms
    ✓ likeScoreTest                                    0 ms
✓ com.example.rankingTest.RelevanceScoreTest          3 ms
    ✓ twoTagRelevantScoreTest                          3 ms
    ✓ singleTagRelevantScoreTest                       0 ms
✓ com.example.rankingTest.UserSimilarityScoreTest     2 ms
    ✓ postRelevantScoreTest                            1 ms
    ✓ subscriberRelevantScoreTest                      1 ms
    ✓ locationRelevantScoreTest                        0 ms
```

Code coverage:



66% classes, 72% lines covered in package 'com.example.wetok.ranking'

| Element | Class, % | Method, % | Line, % |
|---|---|---|---|
| Rank | 0% (0/2) | 0% (0/5) | 0% (0/35) |
| ImportanceScore | 100% (1/1) | 100% (6/6) | 94% (32/34) |
| RelevanceScore | 100% (1/1) | 100% (4/4) | 100% (29/29) |
| ScoreTemplate | 100% (1/1) | 50% (1/2) | 38% (5/13) |
| UserSimilarityScore | 100% (1/1) | 100% (7/7) | 90% (70/77) |

Tests created for Ranking Tests:

1. ImportanceScoreTest: 3 tests for 3 component of calculating importance score: time, like, follower.
2. RelevanceScoreTest: 2 tests for 1 tag relevent and 2 tags relevant.
3. UserSimilarityScoreTest: 3 tests for 3 component of calculating user similarity score: location, subscriber, post.

## Implemented Features

*Improved Search*

1. *Search functionality can handle partially valid and invalid search queries. (medium)*

*UI Design and Testing*

1. *UI must have portrait and landscape layout variants as well as support for different screen sizes. Simply using Android studio's automated support for orientation and screen sizes and or creating support without effort to make them look reasonable will net you zero marks. (easy)*

*Greater Data Usage, Handling and Sophistication*

1. *User profile activity containing a media file (image, animation (e.g. gif), video). (easy)*
2. *Deletion method of either a Red-Black Tree and or AVL tree data structure. The deletion of nodes must serve a purpose within your application (e.g. deleting posts). (hard)*

*User Interactivity*

1. *The ability to micro-interact with 'posts' (e.g. like, report, etc.) [stored in-memory]. (easy)*
2. *The ability for users to ‘follow’ other users. There must be an adjustment to either the user’s timeline in relation to their following users or a section specifically dedicated to posts by followed users. [stored in-memory] (medium)*

*User Interactivity*

1. *Use Firebase to implement user Authentication/Authorisation. (easy)*

## Team Meetings

- *Team Meeting 1*
- *Team Meeting 2*
- *Team Meeting 3*
- *Team Meeting 4*