PROJECT Design Documentation

Team Information

- Team name: Toasters
- Team members:
    - Ethan Hartman
    - Albert Abaykhanov
    - Ben Griffin
    - Bevan Neiberg

Executive Summary

A local organization that facilitates the funding of student projects. Volunteers can either provide monetary support or direct goods and services for projects of their choice.

Purpose

Project Statement: A web application which allows users to fund student projects in order to complete the projects. The most important user group is the Student, which adds needs to their project. Their goal is to fulfill their project needs and finish the project.

Glossary and Acronyms

Term	Definition
SPA	Single Page
MVP	Minimum Viable Product

Requirements

This section describes the features of the application.

Definition of MVP

The MVP allows for users to sign into the application, view student projects, add needs from those projects to their fund basket, and view the fund basket, as well as checking out their basket to fund the student projects. For students, it allows the creation and managing of needs for project funding.

MVP Features

The MVP is comprised of two top-level epics: the Admin Abilities and the Supporter Abilities, each having their respective user stories

Admin Abilities:

- Managers can view who supports their needs: As an admin I want to view all the past receipts so that I can keep track of past funding.

- Funded Need Leaderboard: As an Admin I want to see who has funded the most expensive sum of needs so that there is competition in the application.
- Create New Need: AS an Admin I want to submit a request to create a new need (name [has to be unique], cost, quantity, type) SO THAT it is added to the cupboard.
- Get a single Need: As an Admin I want to submit a request to get a single need so that I can access the cost, quantity and type.
- Update a Need: As an Admin I want to submit a request to modify an item in the cupboard so that I can update needs in the cupboard.
- Delete a single Need: As an Admin I want to submit a request to delete a single need SO THAT it is no longer in the cupboard.
- Get entire cupboard: As an Admin I want to submit a request to get the cupboard SO THAT I can update needs in the cupboard.
- Search for a Need: AS a Developer I WANT to submit a request to get the the needs in the cupboard whose name contains the given text, SO THAT I have access to only that subset of needs.

Supporter Abilities:

- Add needs to basket: As a Supporter I want to add need to my funding basket so that I can potentially fund them later.
- Remove Needs from basket: As a Supporter I want to remove needs from my funding basket so that if I change my mind, I don’t have to fund that need.
- Fund Needs: As a Supporter I want to fund needs so that I can see a Student be successful with their project.
- Get a Single Need: AS a Supporter I WANT to submit a request to get a single need SO THAT I can access the cost, quantity and type.
- Search for Needs: AS a Supporter I WANT to submit a request to get the the needs in the cupboard whose name contains the given text, SO THAT I have access to only that subset of needs.
- Funded Need Leaderboard: As a USER I want to see who has funded the most expensive sum of needs so that there is competition in the application.
- Save Funded Needs: As a User I want funded needs to be saved so that they can be tracked.
- Support Partial Amount: As a SUPPORTER I want to fund partial needs so that I don’t need to supply whole need quantities.
- Display username: As a supporter, I want my screen to display my username so that I can tell that I’m logged in on the right account.
- User Sign Up: As a potential supporter I want to create an account so that I can become a supporter.

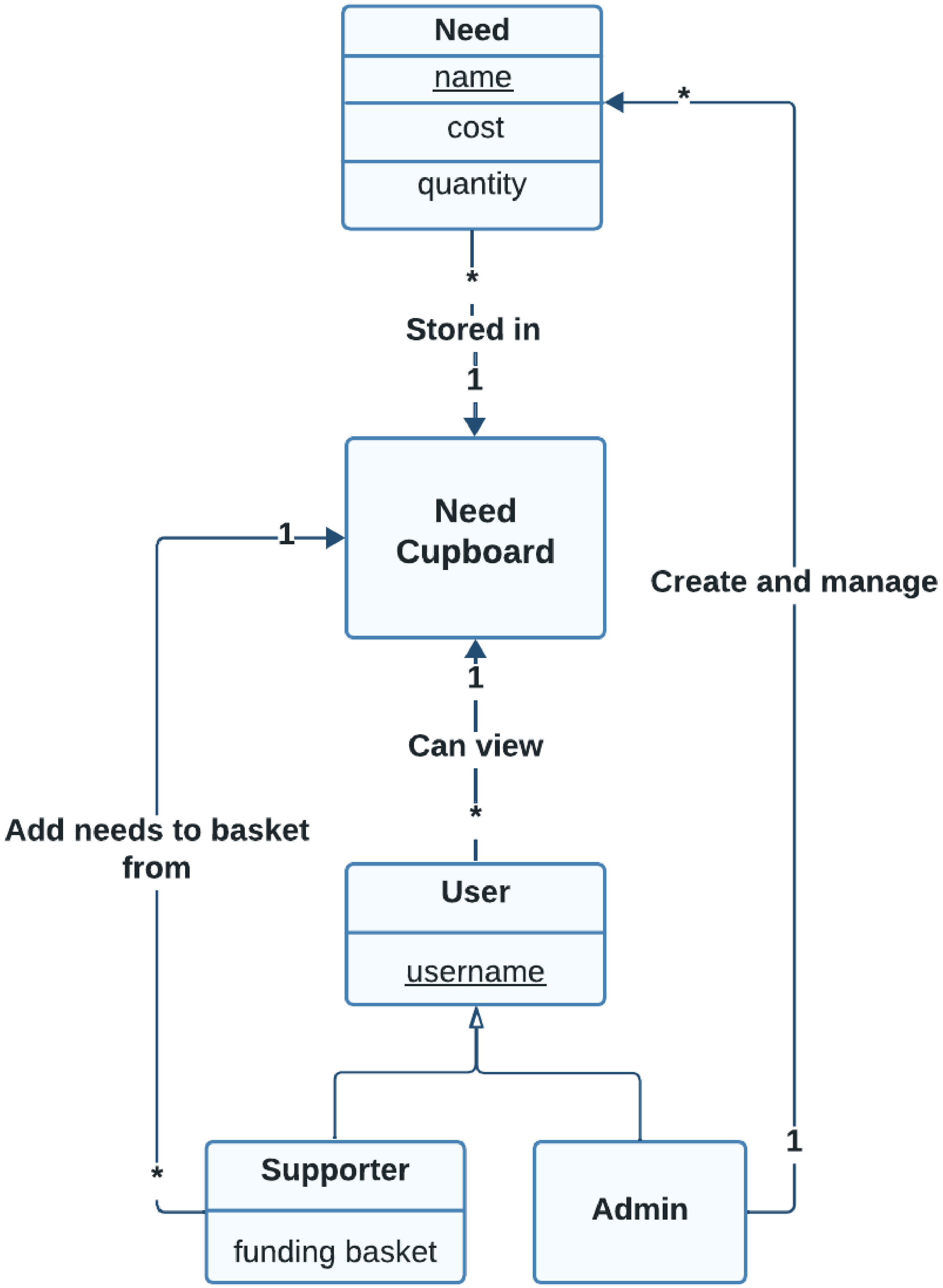
Enhancements

Enahncement 1: Supporter leaderboard After each user checks out needs, the monetary value of their support is saved and kept updated. A leaderboard is diplayed on the home page to illustrate the ranking of users based on the greatest total contributions.

Enhancement 2: Thanking message Admins have access to checkouts receipts, that also contain the contact information of the user who checked out, allowing the admin to send the user a personal message expressing their thanks for the user's support.

Application Domain

This section describes the application domain.



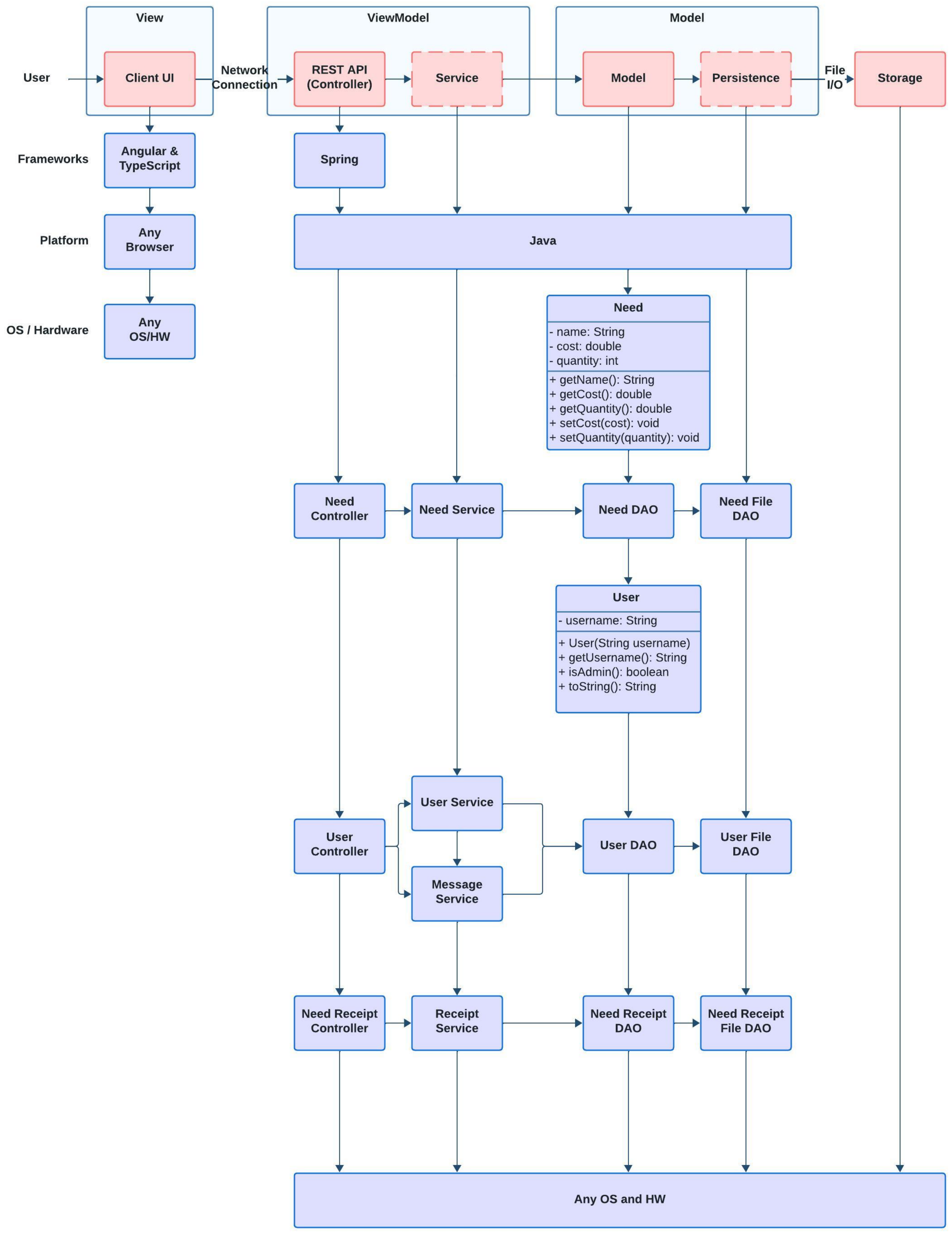
Students can create and manage needs. Students can log in as admins. Supporters can view needs in student projects and they can also add and remove needs from their funding basket. Supporters can check out their fund basket.

Architecture and Design

This section describes the application architecture.

Summary

The following Tiers/Layers model shows a high-level view of the webapp's architecture. **NOTE**: detailed diagrams are required in later sections of this document. (*When requested, replace this diagram with your **own** rendition and representations of sample classes of your system*.)



The web application, is built using the Model–View–ViewModel (MVVM) architecture pattern.

The Model stores the application data objects including any functionality to provide persistance.

The View is the client-side SPA built with Angular utilizing HTML, CSS and TypeScript. The ViewModel provides RESTful APIs to the client (View) as well as any logic required to manipulate the data objects from the Model.

Both the ViewModel and Model are built using Java and Spring Framework. Details of the components within these tiers are supplied below.

Overview of User Interface

The user flow begins at the home page, where users can either log in or explore the projects. Logged-in admin users, have access to a wide range of functionalities like project creation, management, and detailed views. Regular users can browse projects, support them, and engage through the messaging system. The checkout and basket pages facilitate the funding process, while the leaderboard adds a competitive or community-focused element. The application is designed to guide users smoothly from discovering projects to supporting them, with clear navigation and user-friendly interfaces.

View Tier

The application is a system designed for creating and managing Needs. It offers a platform for students to seek support and for kind supporters to contribute, bringing projects to life. Navigating the Application:

Upon launching the application, users are greeted with a login page. This is the gateway to the application's ecosystem, where users can access their roles either as students seeking support or as supporters willing to fund needs.

The Admin Experience:

Logging in as an admin, users find themselves in the "Project Need Cupboard", a central repository where all student needs are listed. Here, an admin can edit existing needs by simply clicking on them. Once valid inputs are entered, the admin can successfully update the need, with the total cost dynamically reflected in the cupboard.

Creating and Managing Needs:

Transitioning to the perspective of a regular user, the application allows for the creation of new needs. A user can add any need they require as long as it doesn't contain conflicting names or other invalid input. In the funded needs section, the application showcases a comprehensive view of all needs that have been funded, including details about the supporters. Users can interact with this section to send messages to supporters.

The Supporter Experience:

From a supporter's perspective, the application provides a seamless experience in browsing and funding needs. A supporter logs in and navigates the list of needs, updating their basket with chosen items. The application dynamically updates the needs list and the stock availability as supporters fund items, ensuring real-time accuracy in the display of information. A search functionality is also available, allowing supporters to filter needs based on specific criteria.

Checkout and Leaderboard:

Supporters can proceed to checkout with their selected items in the funding basket. Post-checkout, the application updates the leaderboard, reflecting the contributions made by each supporter. The leaderboard is an engaging feature that ranks supporters based on their funding amounts, fostering a sense of community and competition.

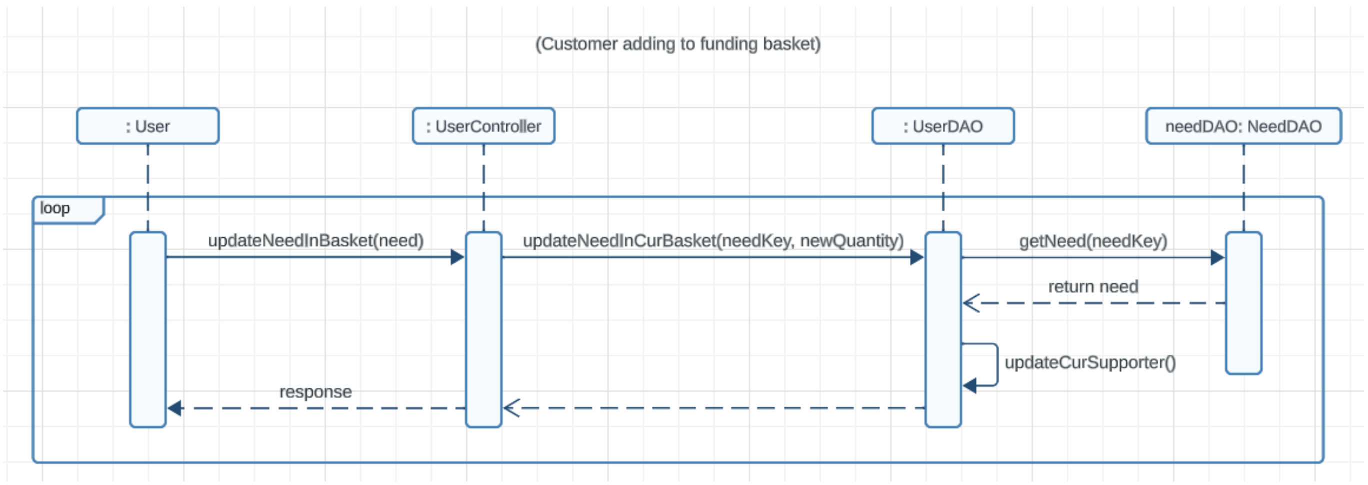
Messages and Interaction:

Another key feature is the messaging system. Supporters receive messages for their contributions, which they can view and manage in their inboxes. This feature adds a layer of personal interaction between students and supporters, enhancing the sense of community within the application.

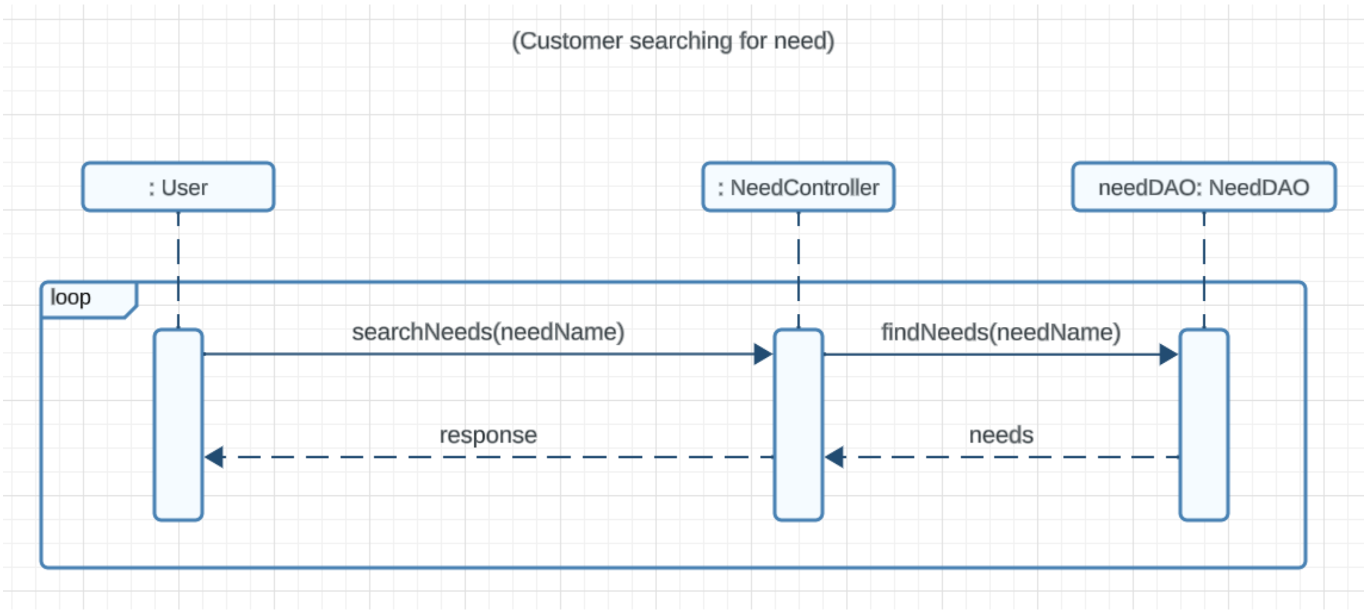
Conclusion and User Cycle:

As users log out and new users sign in, the application demonstrates its capacity to handle multiple user sessions and maintain the integrity of each user's actions. For example, when new user logs in, they see a fresh perspective of the application, with their funding basket and potential contributions reflected distinctly from previous users.

Sequence Diagram 1: Adding a Need to Funding Basket

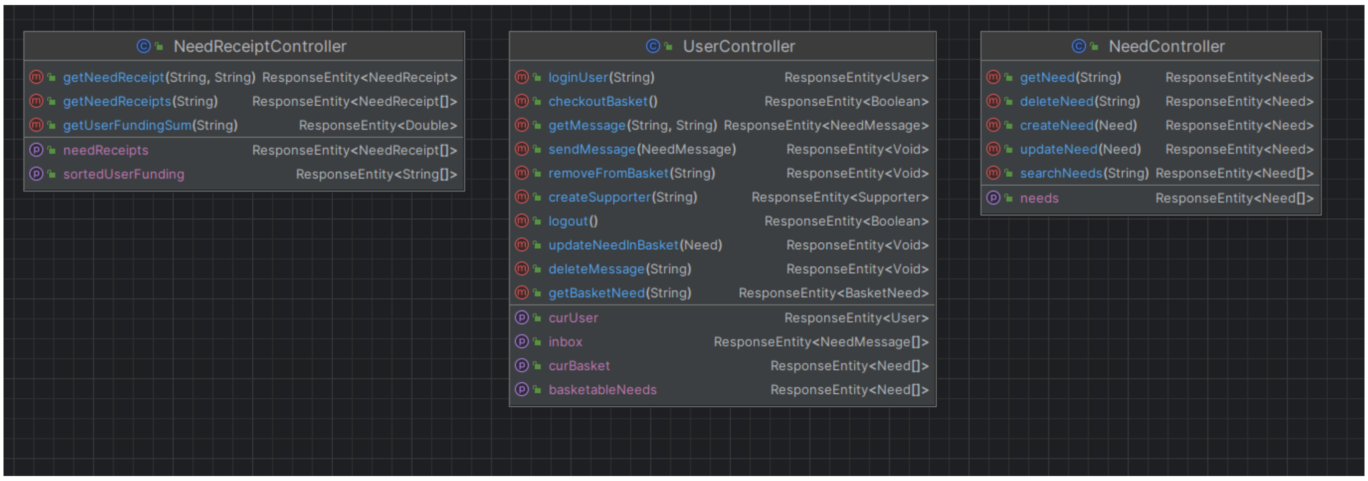


Sequence Diagram 2: Searching for a Need



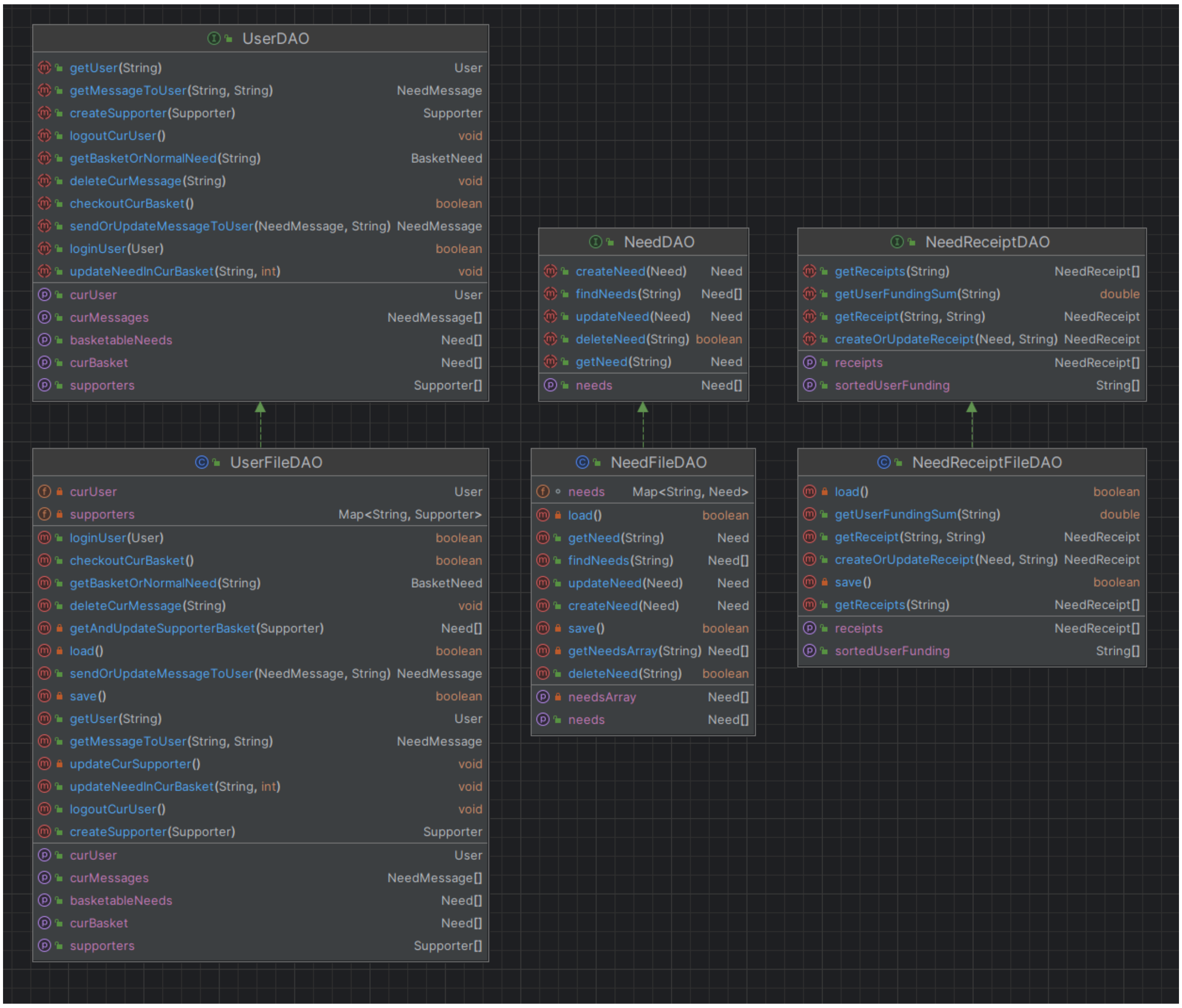
ViewModel Tier

The ViewModel tier sits between the Model and the View. It acts as an intermediary that converts the data from the Model into a format that can be easily presented by the View. It also handles user input from the View and communicates with the Model to update data. In our project, the ViewModel acts as the intermediary between the Angular UI and the Spring Boot backend. Our UI implements services for each object that communicate with its specific backend controller. For instance, the front end can call a service method to delete a need in the cupboard. When this is called, it sends an http delete request to the backend where a mapped method is called to delete the need. Similarly, receipt service and user service also use http requests to communicate with their backend controllers.


NeedReceiptController
getNeedReceipt(String, String) ResponseEntity<NeedReceipt>
getNeedReceipts(String) ResponseEntity<NeedReceipt[]>
getUserFundingSum(String) ResponseEntity<Double>
needReceipts ResponseEntity<NeedReceipt[]>
sortedUserFunding ResponseEntity<String[]>
UserController
loginUser(String) ResponseEntity<User>
checkoutBasket() ResponseEntity<Boolean>
getMessage(String, String) ResponseEntity<NeedMessage>
sendMessage(NeedMessage) ResponseEntity<Void>
removeFromBasket(String) ResponseEntity<Void>
createSupporter(String) ResponseEntity<Supporter>
logout() ResponseEntity<Boolean>
updateNeedInBasket(Need) ResponseEntity<Void>
deleteMessage(String) ResponseEntity<Void>
getBasketNeed(String) ResponseEntity<BasketNeed>
curUser ResponseEntity<User>
inbox ResponseEntity<NeedMessage[]>
curBasket ResponseEntity<Need[]>
basketableNeeds ResponseEntity<Need[]>
NeedController
getNeed(String) ResponseEntity<Need>
deleteNeed(String) ResponseEntity<Need>
createNeed(Need) ResponseEntity<Need>
updateNeed(Need) ResponseEntity<Need>
searchNeeds(String) ResponseEntity<Need[]>
needs ResponseEntity<Need[]>

Model Tier

For the model tier, we can see we have DAOS, which inherit from their own interfaces. These DAOS are used by the controllers. Each DAO has its own functionality, which is represented by the names of them. For example, the NeedFileDAO handles all persistence of Need objects. We can see it has public methods, createNeed, findNeeds, updateNeed, deleteNeed, and getNeed. This DAO additionally stores a list of the currently active needs in the system. The NeedReceiptDAO handles all persistence of NeedReceipt objects. It has public methods, createOrUpdateReceipt, getReceipt, getUserFundingSSum, and getReceipts. This DAO additionally stores a list of the currently active need receipts in the system. Lastly, the UserFileDAO handles all persistence of User objects. It has public methods, getUser, getMessageToUser, createSupporter, logoutCurUser, getBasketOrNormalNeed, deleteCurMessage, checkoutCurBasket, sendOrUpdateMessageToUser, loginUser, updateNeedInCurBasket. This DAO additionally stores a list of the currently active users in the system, and the logged in user information.


UserDAO
getUser(String) User
getMessageToUser(String, String) NeedMessage
createSupporter(Supporter) Supporter
logoutCurUser() void
getBasketOrNormalNeed(String) BasketNeed
deleteCurMessage(String) void
checkoutCurBasket() boolean
sendOrUpdateMessageToUser(NeedMessage, String) NeedMessage
loginUser(User) boolean
updateNeedInCurBasket(String, int) void
curUser User
curMessages NeedMessage[]
basketableNeeds Need[]
curBasket Need[]
supporters Supporter[]
NeedDAO
createNeed(Need) Need
findNeeds(String) Need[]
updateNeed(Need) Need
deleteNeed(String) boolean
getNeed(String) Need
needs Need[]
NeedReceiptDAO
getReceipts(String) NeedReceipt[]
getUserFundingSum(String) double
getReceipt(String, String) NeedReceipt
createOrUpdateReceipt(Need, String) NeedReceipt
receipts NeedReceipt[]
sortedUserFunding String[]
UserFileDAO
curUser User
supporters Map<String, Supporter>
loginUser(User) boolean
checkoutCurBasket() boolean
getBasketOrNormalNeed(String) BasketNeed
deleteCurMessage(String) void
getAndUpdateSupporterBasket(Supporter) Need[]
load() boolean
sendOrUpdateMessageToUser(NeedMessage, String) NeedMessage
save() boolean
getUser(String) User
getMessageToUser(String, String) NeedMessage
updateCurSupporter() void
updateNeedInCurBasket(String, int) void
logoutCurUser() void
createSupporter(Supporter) Supporter
curUser User
curMessages NeedMessage[]
basketableNeeds Need[]
curBasket Need[]
supporters Supporter[]
NeedFileDAO
needs Map<String, Need>
load() boolean
getNeed(String) Need
findNeeds(String) Need[]
updateNeed(Need) Need
createNeed(Need) Need
save() boolean
getNeedsArray(String) Need[]
deleteNeed(String) boolean
needsArray Need[]
needs Need[]
NeedReceiptFileDAO
load() boolean
getUserFundingSum(String) double
getReceipt(String, String) NeedReceipt
createOrUpdateReceipt(Need, String) NeedReceipt
save() boolean
getReceipts(String) NeedReceipt[]
receipts NeedReceipt[]
sortedUserFunding String[]

OO Design Principles

Principle 1: Single Responsibility We can see single responsibility, as we have our controllers, persistence, and model classes all separated. Moving even deeper, we have a User class, which holds usernames for users, and a Supporter class specifically for supporters, which inherits from the user class and includes a funding basket. We separated the two because an admin is a user but is not a supporter.

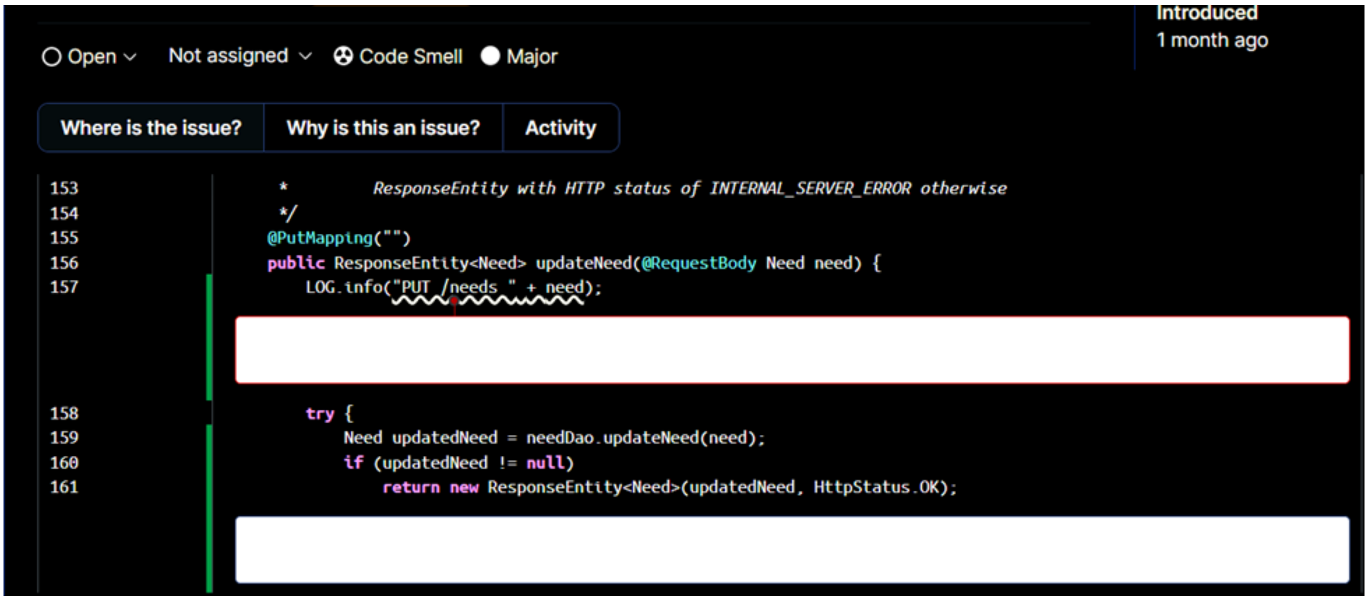
Principle 2: Law of Demeter We see the law of Demeter when analyzing the controllers. The two controllers, UserController and NeedController only have access to their specific DAO class, UserDAO or NeedDAO. These controllers are then accessed by their specific Angular service, user service and need service.

Principle 3: High Cohesion All of our classes are named according to their functionality, similarly, they use other classes which have related functionality. For example, if we look at the NeedFileDAO, we can see that this handles all persistence of Need objects

Principle 4: Open/Closed The Open/Closed Principle is used in our project such that classes are open for extension but closed for modification. A specific example of this was our use of extending error classes, instead of having to rewrite our files. We wrote our code so that when we need more error handling, we don't need to break other code to add more handling.

Static Code Analysis/Future Design Improvements

SonarQube recommends using fstrings instead of concatenation. For example, we should use LOG.info("PUT /needs {need}"); Direct concatenation is apparently inefficient.



Introduced
1 month ago

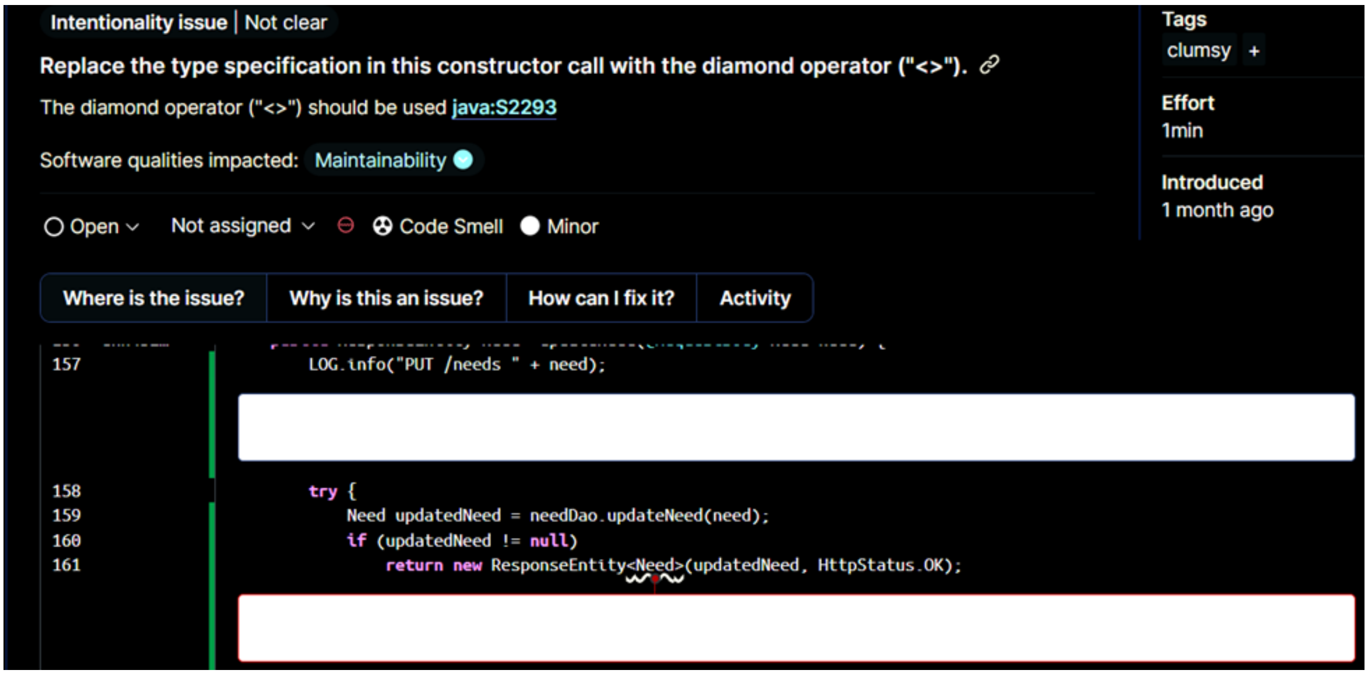
Open · Not assigned · Code Smell · Major

Where is the issue? | Why is this an issue? | Activity

```
     *         ResponseEntity with HTTP status of INTERNAL_SERVER_ERROR otherwise
     */
    @PutMapping("")
    public ResponseEntity<Need> updateNeed(@RequestBody Need need) {
        LOG.info("PUT /needs " + need);

        try {
            Need updatedNeed = needDao.updateNeed(need);
            if (updatedNeed != null)
                return new ResponseEntity<Need>(updatedNeed, HttpStatus.OK);
```

Many of our controller returns include the item type while creating a new ResponseEntity. However, this is no longer used. Instead, we should just use empty diamond operator.



Intentionality issue | Not clear

Replace the type specification in this constructor call with the diamond operator ("<>").

The diamond operator ("<>") should be used java:S2293

Software qualities impacted: Maintainability

Open · Not assigned · Code Smell · Minor

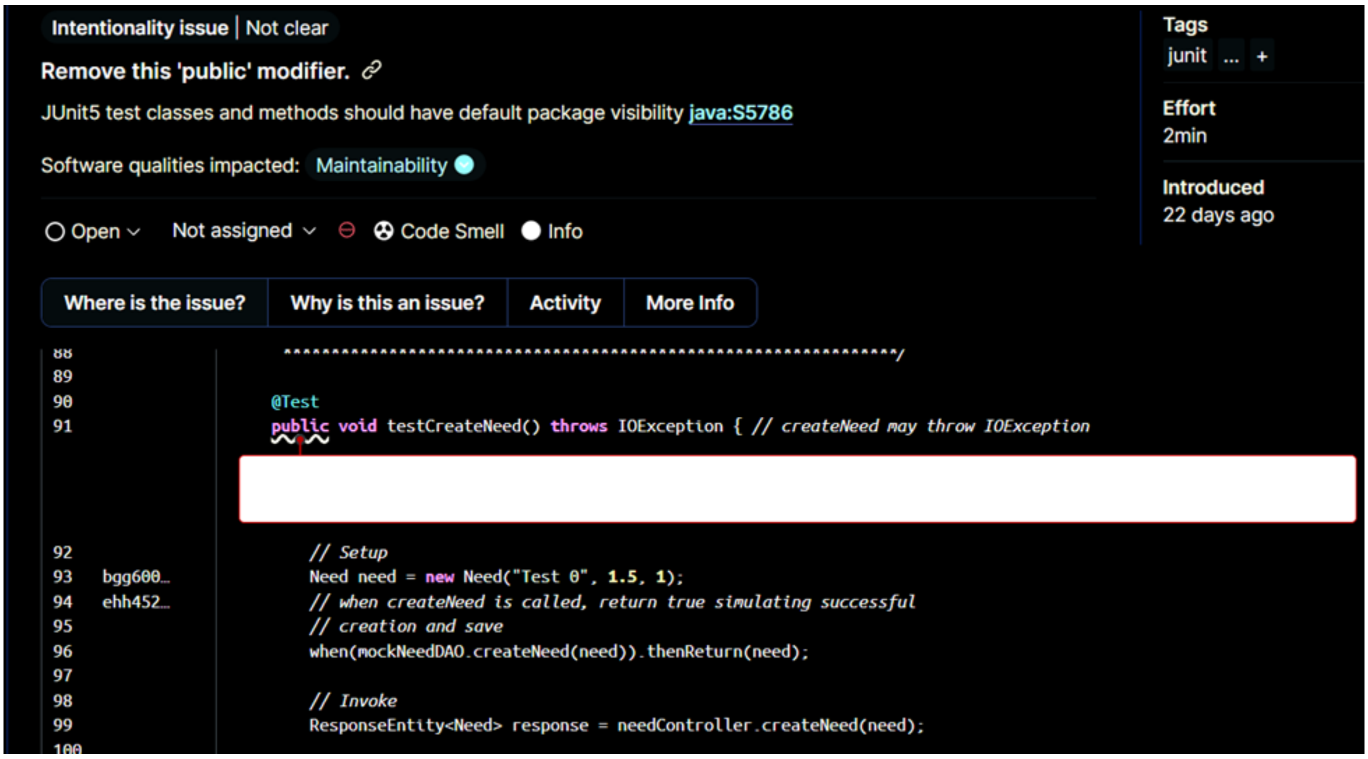
Tags: clumsy
Effort: 1min
Introduced: 1 month ago

Where is the issue? | Why is this an issue? | How can I fix it? | Activity

```
        LOG.info("PUT /needs " + need);

        try {
            Need updatedNeed = needDao.updateNeed(need);
            if (updatedNeed != null)
                return new ResponseEntity<Need>(updatedNeed, HttpStatus.OK);
```

All tests are public functions. These should be changed to private since they aren't used elsewhere.



Intentionality issue | Not clear

Remove this 'public' modifier.

JUnit5 test classes and methods should have default package visibility java:S5786

Software qualities impacted: Maintainability

Open | Not assigned | Code Smell | Info

Tags: junit ... +
Effort: 2min
Introduced: 22 days ago

Where is the issue? | Why is this an issue? | Activity | More Info

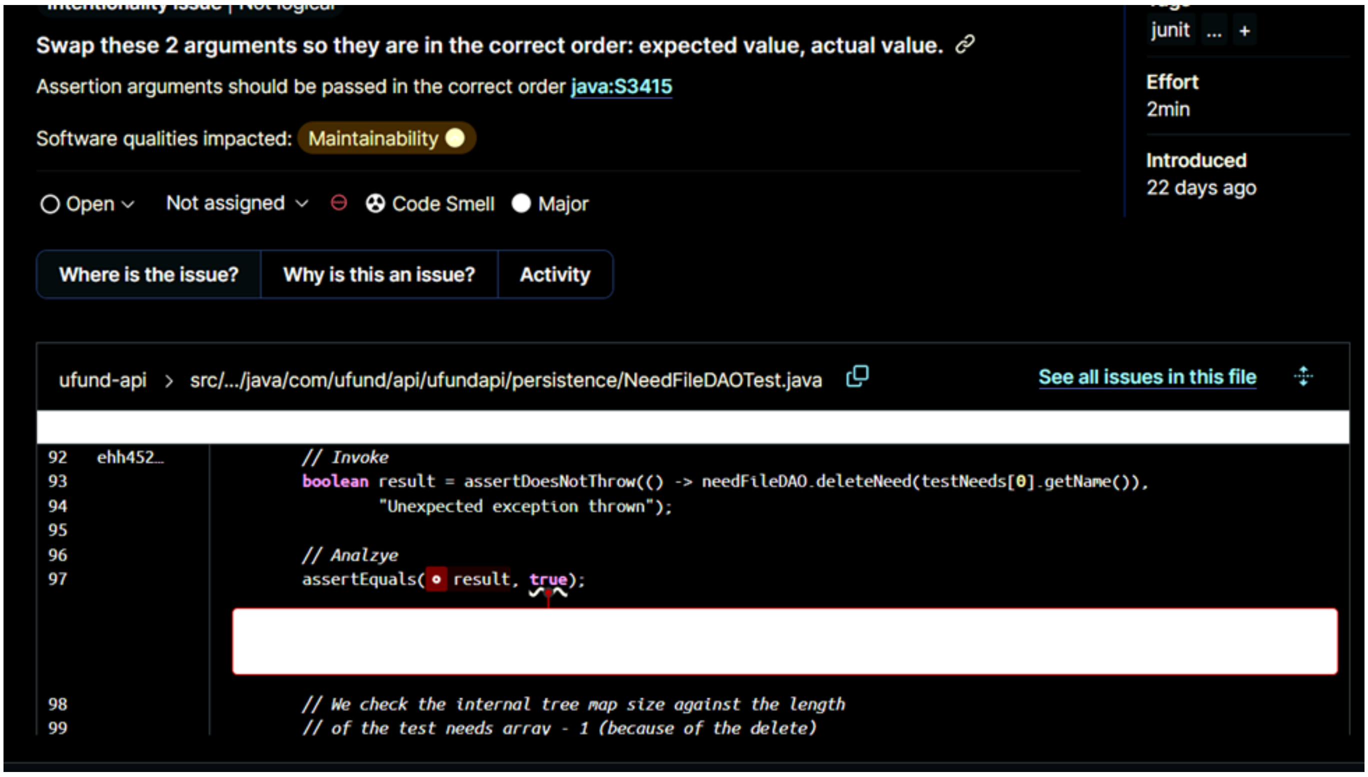
```
     *****************************************************************/

    @Test
    public void testCreateNeed() throws IOException { // createNeed may throw IOException

        // Setup
        Need need = new Need("Test 0", 1.5, 1);
        // when createNeed is called, return true simulating successful
        // creation and save
        when(mockNeedDAO.createNeed(need)).thenReturn(need);

        // Invoke
        ResponseEntity<Need> response = needController.createNeed(need);
```

When doing test cases, assertEquals should use expected value, then actual as arguments. For some testing methods, we were doing the opposite.



Swap these 2 arguments so they are in the correct order: expected value, actual value.

Assertion arguments should be passed in the correct order java:S3415

Software qualities impacted: Maintainability

Open | Not assigned | Code Smell | Major

junit ... +
Effort: 2min
Introduced: 22 days ago

Where is the issue? | Why is this an issue? | Activity

ufund-api > src/.../java/com/ufund/api/ufundapi/persistence/NeedFileDAOTest.java   See all issues in this file

```
        // Invoke
        boolean result = assertDoesNotThrow(() -> needFileDAO.deleteNeed(testNeeds[0].getName()),
                "Unexpected exception thrown");

        // Analzye
        assertEquals( result, true);

        // We check the internal tree map size against the length
        // of the test needs array - 1 (because of the delete)
```

In the future, if we had more time, there would be numerous things we'd like to accomplish. Firstly, we would have liked making it so that the admin wasn't the only student who could have a project, and add needs to their project. It would have been nice if users could sign up as students, create projects for themselves, post needs, and have those needs be funded. Additionally, our UI was quite simple. It would have been nice to have a more complex UI, with more features, such as better need filtering, and a more complex leaderboard. Lastly, it would have been nice to track the progress of projects and needs so users can ssee how close a project is to being completed.

Testing

Acceptance Testing

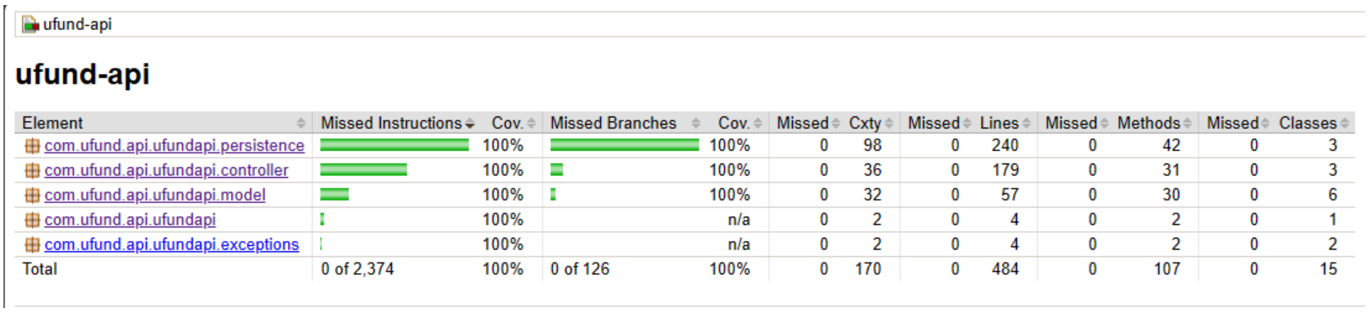
Sprint 2: 9/9 user stories passed all acceptance criteria tests. No issues found during acceptance testing.

Sprint 3: 30/34 acceptance criteria passing tests, 2 of which were old acceptance criteria which should have been updated, but they weren't, resulting in failed tests. For the last 2, the testing team couldn't figure out how to use the search bar. However, if used correctly, the acceptance criteria do pass.

Unit Testing and Code Coverage

Our unit testing strategy is a fundamental aspect of our development process, aimed at ensuring the reliability, functionality, and maintainability of our codebase. Our strategy is to develop a comprehensive suite of unit tests covering critical components, functionalities, and edge cases within our codebase. As of the latest assessment, our code coverage stands at 100%. This aligns well with our target coverage of 100%. We set this target to mitigate risks and ensure that all exceptions are handled.

Our code coverage



ufund-api

ufund-api

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.ufund.api.ufundapi.persistence	🟩	100%	🟩	100%	0	98	0	240	0	42	0	3
com.ufund.api.ufundapi.controller	🟩	100%	🟩	100%	0	36	0	179	0	31	0	3
com.ufund.api.ufundapi.model	🟩	100%	🟩	100%	0	32	0	57	0	30	0	6
com.ufund.api.ufundapi	🟩	100%		n/a	0	2	0	4	0	2	0	1
com.ufund.api.ufundapi.exceptions	🟩	100%		n/a	0	2	0	4	0	2	0	2
Total	0 of 2,374	100%	0 of 126	100%	0	170	0	484	0	107	0	15