Milestone 2: Detailed Design

Authored By: Evan Roubekas, Muhammad Asim Ali, Shaun Lyne

Introduction:

This report was written with the intent of defining and designing the architecture for our web app. The main purpose of the web app is for users to have the ability to track changes in prices for various shoes they have purchased and how that price compares to their current resale price (the concept is very similar to stocks). In this report you will find two UML diagrams for both class and behavior as well as rationale for their design. The end of the report contains a quick summary on our plan for continuous integration, testing, and deployment.

Figure 1 UML class diagram

UML Class Diagram - Status: Proposed

Context:

User classes

The system needed to have a login module, which required the system to have users. These users needed to have a username, email, and password. The users also needed to have functionality to change their username, password, and email address.

There was also a need to seperate user types into an admin user type and a customer user type. The customer user type needed to have their personal product list, as well as a number of other attributes like their net gain/loss, number of shoes, total sunk cost, current net worth, average net gain/loss, ranking, and user id. The customer also required ways to get info on one of their shoes, add a shoe to track, and untrack a shoe. The admin required functionality to add/remove users, and add or remove products to and from the master product list.

Additional Functionality

A function that was needed for the customer was the ability to view the master product list. The customer needed the functionality to get info on shoes, search for shoes, and view all available shoes. There also needed to be functionality for sorting the currently available shoes by price, brand, or model. This would allow customers to look at shoes they were interested in to see the current market value, or decide whether or not they wanted to invest in a certain shoe.

We wanted customers to be able to add notifications so that when a certain shoe went below (or above) a certain threshold, they would get a notification about it. This would allow customers to buy or sell a shoe at a certain price.

Another function that we wanted was the ability for customers to view their ranking on a leaderboard, against other users. There also needed to be a way to sort the rows in the leaderboard table by net gain/loss or average net gain/loss. This would allow customers to gauge where they sit vs. all other users if they were inclined to do so.

Decision:

User classes

Admin and customer classes were made to be subclasses of the “user” class. The decision to have a seperate “user” class from which all other user types would inherit was based on the benefit of increased modifiability. In the future if there was a requirement to add another user type we could easily make another subclass of user.

Admin

For the admin class we added a method “add_user” which allows the admin to make users with admin privileges. We also gave the admin ability to remove users (customers and admins). This implies that an admin can remove another admin. Furthermore, we also added methods to add, make inactive and update products, these functions make admin essentially a “controller” for the product model. Only the admin is able to add and edit products to ensure data reliability which is one of the quality attributes of the system. Only the admin has access to the “admin view” and “product factory view”. The admin view and product factory view enable the admin to easily add or edit products which supports the usability quality attribute of the system for adding products efficiently.

Customer

The customer class was made to represent a lot of the core functionality of the system for the users who are the main stakeholders of the system. The customer class has attributes such as product list which is a list of all the products the user is tracking and analytical attributes such net income, average income, sunk cost and rank. The customer view was made to provide the ability to add new products to track, remove products from the users product list, request new products to be added to the product model view, and also allows the customer to see their products and sort them according to different attributes of the product. The customer_controller executes all of these functions that are requested by the customer through the view.The ability to sort the products the user has in their product list was an important functionality that was needed to support the usability quality attribute for the customer.

Notification Centre

For the notification centre, we used a model called notification_centre, which stores the customer’s notifications (in a Notification array), and has functionality to query the customer’s notifications and send a notification if one or more shoes has broken the threshold set by the customer. The notification_centre_view would allow the customer to view their notifications and add a new notification. The notification_controller executes the commands to get info on a specific notification, add/remove notifications, and get all notifications.

Leaderboard

For the leaderboard, the model used is stored in the database. This model stores a variety of attributes such as the users rank and the number of shoes they own, their average net gain/loss, and their net gain/loss. The leaderboard view would show the top 100 users, as well as the users rank and their statistics. The leaderboard_controller allows for users to sort the leaderboard based on avg/net gain/loss, and also allows the user to display a specified number of leaderboard entries.

Master Product View

The product_model_view allows for customers to view all available shoes, search for a certain shoe, get info on a certain shoe, and sort the list by price, brand, or model (ascending/descending). The product_view_controller executes these commands that the user calls from the view.

Data Factory

Something not shown on our class diagram is the data factory, which is responsible for updating the market values of shoes in the database. Since the data factory is created by us, this ensures data reliability since all the data produced in-house. This also ensures that the data is updated quickly, which means that the data is as close to real-time as possible.

Additional Notes

The portability of the system will be dependant on the implementation, which is not specified in this class diagram. Because of this, the design does not address the portability QAR mentioned in Milestone 1, which is left up to the implementation.

Consequences:

One of the main consequences of this design is the amount of interconnectivity that is present between each of the modules, as well as the number of different views and controllers in the design. Though this helps with understanding the design as a whole, having such a large amount of modules may create challenges during implementation. There is also a lot of duplicate data spread out between the different modules, which will all need to be updated whenever a shoe is updated in the master product list by the data factory. This has the potential to affect system speed and data reliability.

A consequence arising from having a leaderboard is that when any shoes cost is updated, the user's leaderboard ranking must be updated too. This could again potentially affect how close the leaderboard's data is to real time, and could also affect the speed of the system as a whole depending on how often a shoe's cost is updated.

A consequence of having attributes linked to the customer such as net/average gain/loss is that whenever the customer adds a new shoe, these values must be updated. In addition to the customer's attributes, their ranking on the leaderboard must also be updated. Again, this may potentially affect the speed of the system and the reliability of data if many customers are adding shoes at the same time.

A consequence of allowing customers to request products is that there may be a customer who gets bored and decides to write a program to spam the admins with product

requests. Though there is currently no functionality to prevent this, the admin can ban the user to stop the torrent of requests. Having this product request functionality may also cause bogus products to be requested, which ends up wasting admin time and delays actual products being added to the master product list. This may affect the speed at which data is updated.

A consequence of the admin user being able to add new users with admin privileges and also being able to remove other admin users from the system is that this functionality has the potential to be misused. However since an admin account would be governed by company policy this would not be a likely scenario and the benefits of easily being able to add new admins or remove existing admins outweigh the risk of this functionality being misused.

UML Behavior Diagram - Status: Proposed

Context:

The context of the behavior diagram is one of our user stories. “As a sneakerhead I want to be able to add more pairs of shoes to my portfolio in order to track my profits as I acquire new shoes.” The below sequence diagram goes through the entire process from a user, requesting a new shoe to be added to the product list, to tracking the shoe.

Decision:

The main design decision that can be seen from this diagram is the distinction between the user requesting the product to be added, and it being added to the database (via the admin user). The reason for not allowing the user to directly add a product to the database stems mostly from security concerns. The process of adding a product would be much more involved if there were to be no admin moderating the content being added to the database, as we would need to develop methods to prevent cross site scripting attacks and the injection of malicious code from the user input.

The secondary design decision we would like to draw attention to is the ability for the admin to directly add products to the database. This allows for increased usability for the admin since they can directly add products the database without much downtime. This also increases the speed at which new products are added to the database, which helps keep data as close to real time as possible.

Consequences:

The main consequence of having admin intervention in adding a product is the time it takes. If 10 users submit a request, and there only exists 2 admins, it will obviously take more time for the admins to add all the products versus having the users do it themselves. However, with this in mind, the prevention of malicious code injection is worth the extra time it will take. Also, thus far we have not designed a way to notify users that their request has been put

through and they will now be able to add their product from the product list. As it exists right now users will just have to periodically check the list to see if their requested product has been created.

A consequence stemming from allowing the admin to directly modify the database is that there may be a case where an admin uses this function maliciously. However, since the number of people who will be admins in this project are very few, the ease of adding products to the database outweighs the risk that they function may be used maliciously.

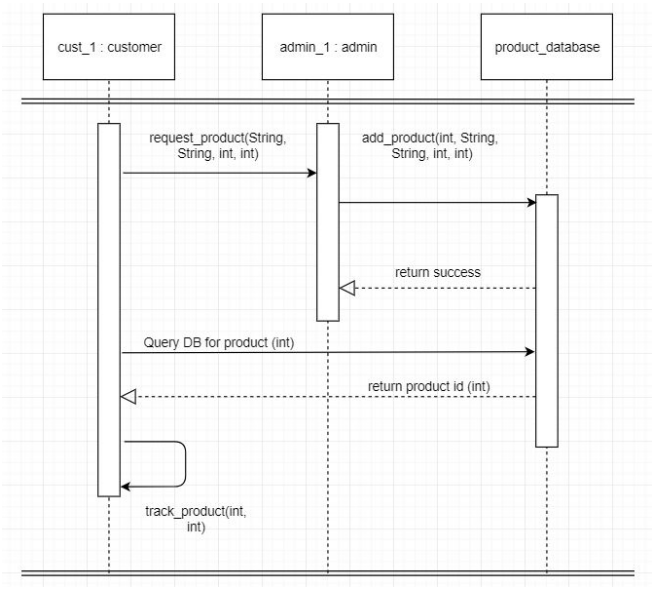

Figure 2: UML Behavior Sequence Diagram

Continuous Integration

The continuous integration (CI) for our web application will be done through the Travis CI tool. We are choosing Travis over another CI tool such as Jenkins because of its thorough documentation, ease of use, and the fact that it is listed as a deliverable in milestone three. The pipeline will be standardized across the three team members as the web app development and deployment will be happening with a docker container. Therefore, the CI pipeline will not need any extra requirements or setup to make cross platform use more accessible. The pipeline will work in a similar fashion to what is displayed on Travis' homepage (see figure 1.).

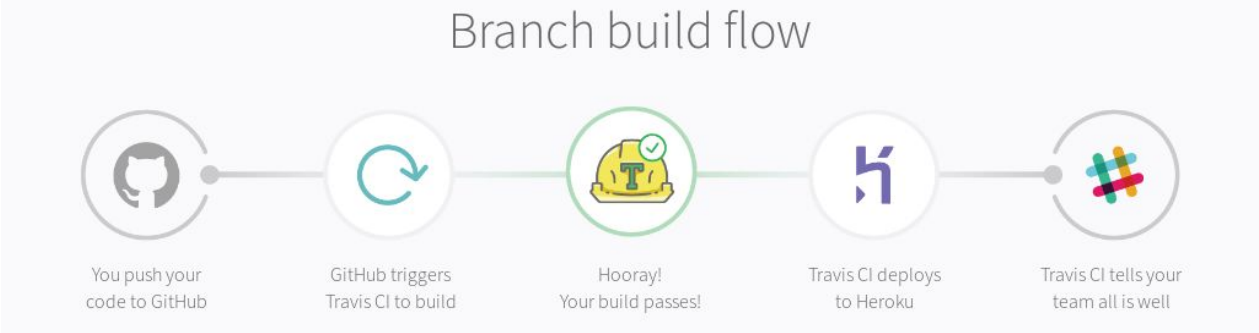

Figure 3: Typical Pipeline Flows for a Project (source: www.travis-ci.org)

The differences in our approach will be that there will be a more in depth testing sequence during step 2 as well as deployment not to heroku, but rather to a digital ocean droplet.

Testing

Testing is a crucial step in all development, without it software projects are quick to fail. As such, we have chosen the automatic software tool Selenium, to assist us in making coherent, effective tests for our web app. Emphasis must be added on the *automatic* part of selenium, as it will make our development cycle much more efficient. Testing that otherwise cannot be done automatically, such as usability testing (ex. ease of use) will be done manually. We will also make use of unit tests to ensure that our typescript is working properly.

Deployment

Deployment will most likely be the easiest part of this pipeline to set-up as our group member Evan already has a digital ocean droplet with a NGINX web server (as well as reverse proxy) configured and running. This will also be where the live database is stored and deployed using MongoDB. The web app will be running on a sub-domain of roubekas.com. The app should be running as long as the server is up (excluding fatal bugs), this also includes the database. This relates to the systems quality attribute for availability as the app will be up as long as the server and database are running. For this deployment to be configured there needs to be a bare git project repository with a post-receive hook that runs a script to update and rebuild the docker container that is hosting/running the web app. Within the travis.yml config file that sets up the pipeline there will be an option to build, test, and deploy the projectm or to merely build, test, and push to github. The latter option is preferred in cases when the team members are actively collaborating on a new feature that is not yet ready for production.