Authors

Johan Andersson
David Fogelberg
Sam Halali
Gunnar Gunnarsson
Nandha Gopal Elangovan
Miguel Angel Sanchez Cifo
Jonathan Granström

Git repository: https://github.com/lol2kpe/EDA397_Team3
Issue tracker: https://github.com/lol2kpe/EDA397_Team3/projects/3

Project Description

Project Title

H4U

Platform

Android

Minimum and target SDK

5.0

Description

The H4U app helps a user search for hospitals, pharmacies and doctors including; doctor details, doctor availability and doctor reputation. It will also show the directions to each hospital by showing the route map. The route map will be set from a user's current location and will show both a visual and descriptive way of the route, which will make it easy for a user to find the hospital location.

Sprint Log: Sprint 3

Commitment

List the features/stories the team commits to finish during the sprint.

User Stories

ID	User story
76	As a user, I want to change the language of the application, so that I can better understand it
77	As a user, I want to be able to search for a hospital directly, so that I can more quickly find a specific hospital
79	As a user, I want to be able to update my profile, so that I can change or update my information
80	As a user, I want to be able to set my favorite hospital, so that I don't have to search/look for that particular hospital all the time

Tasks

ID	Tasks

65	Add functionality to user profile
75	Fix continuous integration builds with Travis CI
78	Add emergency call button
81	report file
82	Connect add favorite place and delete favorite place - profile
84	Add search feature
87	Translate app into Spanish
88	Add app icon
89	Refactor the filter to support language change
90	Update filter with proper symptom filter functionality
95	Add option to change the language manually
96	Add swedish language option
85	Refactor place model to support symptoms

Work Done

Feature	Commits	Group members	Effort	Practices
Fix continuous integration builds with Travis CI	a026c38 9e8be76 e65f194 25a2109 5a8440a a30d2dc 7ad32fd a452eb0 c62dfd8 679d99a 4f7da31 05f9c24 097a852 ca691a0 45fdbfe 8c44e2d ce6c732 0805288 0933317 05af999 c08ed89	Johan Andersson	27hr	Refactoring
Map does not fill entire screen	e434e07	Johan Andersson Miguel Angel Sanchez Cifo	1hr	Pair Programming, Collective Codeownership, Coding Standards, Small Releases
Add app icon #88	3301033	Johan Andersson	2hr	Coding Standards, Small Releases
Add swedish language option #96	27e6264	Johan Andersson	2hr	Coding Standards, Small Releases, Simple Design
Refactor the filter to support language change	38e125a	Jonathan Granström	12hr	Sustainable Pace, Simple Design, Code Refactoring, Collective Code Ownership, Small Releases
Update filter with proper symptom filter functionality	20565b7	Jonathan Granström Sam Halali	4hr	Pair Programming, Code Refactoring, Simple Design, Sustainable Pace, Collective Code Ownership, Small Releases

Feature	Commits	Group members	Effort	Practices
Change the Services class into a Symptom class	d6c8e9d	Sam Halali	1hr	Simple Design, Refactoring
Add search feature	2fd7048	Sam Halali Miguel Angel Sanchez Cifo	3hr	Simple Design, Refactoring, Pair Programming
Add functionality to user profile	2cfc325	David Fogelberg	4hr	code refactoring, sustainable pace, collective code ownership
Connect add favorite place and delete place profile	f60d04b	David Fogelberg	4hr	code refactoring, sustainable pace, collective code ownership
create report file	3acddab	Nandha Gopal Elangovan	2hr	Small release
Refactor model	56f3a79	Nandha Gopal Elangovan	2hr	code Refactoring
Add option to change the language manually	0d8d2ef	Miguel Angel Sanchez Cifo	3hr	Simple design, Code refactoring
Translate app into Spanish	f114bbf	Miguel Angel Sanchez Cifo	1hr	Simple design
Add emergency call button	a453db5	Gunnar Örn Gunnarsson	2hr	Simple design

Reflections

In this section we describe our experiences of implementing agile methods in our project during sprint 3.

For the third acceptance test, we felt more comfortable with what we had created.Since we had clear list of needs from the customer side and worked on accordingly to the sprint planning , we completed the prioritized tasks and user stories.

After the second acceptance test, there were a few priority requirements requested from the customer's side. We had a discussion about them and listed out the prioritized users stories and tasks that covers the customers requirements.

Implementation of agile principles:

In order to manage requirements and organize the activities, we had to make the most out of the time and resources available. We focused on implementing agile methodologies.

The methodology

Test-first:

Test-first has been the harder and difficult to utilize properly. As the project goes on and the code is refactored, added, and combined so new dependencies occur, creating proper tests becomes more difficult and time consuming. Once you also factor in that the old tests needs to be updated, and you have other work to complete before the deadline, even more time needs to be spent on tests and more stress can be felt.

Also, at certain points throughout the project, it felt as if tests were written just for the sake of writing tests. You shouldn't find yourself in a situation where you ask yourself "Wait, why am I writing this test?". Some tests were pretty basic and didn't really need to be written and implemented, for example, testing if adding a value to a HashMap and the value really is added. Testing should really be implemented for code that have conditionals, loops,

transformations, etc, but this kind of boilerplate code. Of co urse, you could argue that it is good to implement tests to make sure another developer doesn't break the code, but you have to got some faith that other developers aren't clueless or has the intent to break your code. These kinds of tests are just pointless and a waste of time, and leaves you or other developers with the task to maintain those tests. If a practice becomes too "mainstream" we might very well forget what its real purpose is and the value of the practice.

So to make sure the practice of "Test-first" isn't misused and too time consuming, understanding which tests are useful and serves a good purpose and which tests are counterproductive and unnecessary is important. Black-box testing is great for this purpose. The written tests make sure that the functionality is working properly, without too much care going into how the functionality gets the results. This helps with the problem when you don't know exactly how the code for some functionality will be written, but you do know what input and results you expect. Tests also doesn't have to be too complicated. They could consist of print or log statements to make sure the basic functionality is consistent. The importance of simplicity, generalization, and automatization is especially important as code refactoring and test-first almost always goes hand-in-hand.

- Coding standards:

Coding standards hasn't really changed since the last sprint. So We still follow the chosen standard.

- Pair programming:

It is an intense experience, and it requires a bit of time to get used to. Where some pair programming was done, we had to connect the work with other members' work. which leads to increased code quality.

- Sustainable pace:

Keeping a sustainable pace has been both easy and hard to do properly. Other course work has a lot of times kept focus away from the project and taken away the development time, leading to the feeling that you really need to sit down and work on the project, especially as the deadline to the acceptance test draws closer. But getting your head away from the code and getting an opportunity to sit back, think, and plan has helped in coming back to the code and knowing what to do next, making sure the next hours spent on the code is efficient. A lot of people could look at a problem for hours, not knowing what to do to solve it, only to wake up the next morning or coming back to the problem after a few hours and instantly see the solution. This is the idea with the principle, to use the development time wisely and efficiently, and not putting emphasis on hours of work. Proper planning before each sprint, and knowing what the others are working on helps with keeping a sustainable pace. But other principles also affects the principle. Complex and time consuming tests and unnecessary refactoring (as mentioned in their respective sections) could take time away from development time, making it harder to properly follow the principle.

- Code Refactoring:

Code Refactoring has had great value throughout the project. At some times, dumb and ugly code has been implemented in order to get the application to work temporarily, or due to time constraints before an acceptance test. Going back and refactoring that code afterwards has been very important. Just because code works doesn't mean it will always work. Refactoring is also important for the collective code ownership principle, as it makes the code more readable, maintainable, and reliable. As long as the refactoring doesn't change the underlying functionality or logic of the code, you will get the improvement of overall better code.

But refactoring could also create problems. At certain points, every developer will look at their written code and think "I've could have done this better" and tries to reshape and fix the code (which already works), for example, using a switch statement instead of an if-else statement. Refactoring is of course necessary when "dumb code" has been used to make a feature temporarily work, or when you need to support a data model, etc. but at some point, a developer might refactor code that already works just because the code is "working-but-awful". This is especially bad if other tasks needs to be completed and time is wasted on making a certain part of the code better, having a negative effect on productivity. Some factors that lead a developer to do unnecessary refactoring could be the thought that "this code is mine" and it becomes a matter of reputation ("No one can see this badly written code!"), or the perfectionist inside us is telling us that this is wrong and needs to be changed right now. It is especially bad if you start working on other developers code. Sitting back and asking yourself "do I REALLY need to work on this code, or has someone asked me to work on this code?" before you start refactoring could help in limiting refactoring work, and making sure that whatever refactoring is done is actually necessary.

Throughout the project, code refactoring has somewhat gone hand-in-hand with test-first, as it is important to check that after refactoring is done that the overall functionality isn't broken. As mentioned in the test-first section, as code is refactored and updated, this changes how existing tests function as well. Knowing how to put tests in place is not easy, but you can make it easier for yourself with by having proper tests in place. As mentioned in test-first, this puts extra effort and value on making sure that existing tests are properly implemented in order to reduce the workload and make sure the code works properly. If tests haven't been implemented before or after the refactoring is done, simple tests like print statements is enough to make sure the functionality is consistent, just because some kind of tests is better than non.

- Collective code ownership:

It has been harder to have proper knowledge of what the entirety of the code in the application does. As the project moves on, more and more code is added, which makes it more difficult to understand everything in the project. Adding comments to methods, classes, etc. isn't always done, so time has to be spent looking at the entirety of the code in order to understand it. Also, as some users has spent a majority of their development time on specific tasks and code, they usually keep working with that code exclusively. This makes it so that certain members are considered "experts" on some parts of the code, and have full responsibility and knowledge for that code. Of course, at some times, other members will fix errors or make additions to code written by others, so it isn't that bad and most members understand how other people's' code works, but mainly on a basic level and not to in-depth.

- Simple Design:

This has been a somewhat easy principle to follow. With each task, first creating a stupid but easy solution to the problem, using whatever current knowledge you have to make it work, has not been a hard thing to do. However, this has lead to a lot of refactoring work, as the code works but might not be optimal, easy to understand, or reliable. At some point, it felt as if it would have been faster to first create a more complex but better solution, than an easy one that would need refactoring work to be complete. It is a matter of weighing in on pros and cons.

- Continuous-integration:

Continuous - integration is carried out to seek two important objective (a) Minimize the duration and effort required by each integration episode and (b) Be able to deliver a product version suitable for release at any moment, where it leads to involve in producing a clean build of the system several times with continuous-integration the process is productive. A good way to ensure working code for each release is to use a continuous integration service like Travis CI. Unfortunately, we did not realise this until after the second sprint. Hence for this sprint, a lot of work with getting travis to build everything without failing has been carried out. Even though this was resolved, it was done by excluding some tests and reverting code that had already been deployed. Therefore, the code that works with travis have not been deployed to the main branch at this point.

- Testing:

The testing methods mentioned above are useful where the raised defects are fixed within the same iteration and thereby keeping the code clean. Hand on tests have also been carried out in order to ensure that no faulty code is being deployed. This has been done either in the android emulator or an actual android phone. However no test protocols have been set up.