GHENT UNIVERSITY

FACULTY OF ENGINEERING AND ARCHITECTURE

Testing and Quality Assurance

Design Project - Runamic Ghent

Group 16

Depauw Hendrik

van Herwaarden Lorenz

Aelterman Nick

Cammaert Olivier

Deweirdt Maxim

Dox Gerwin

Neuville Simon

Uyttersprot Stiaan

1 Introduction

This part of the final deliverable describes how the test plan has evolved during the project and how the prototype has been tested. Quality assurance is also described within this document, specifying how the quality of the product has been maintained and which changes concerning quality assurance have been made during the project.

2 Initial test plan

2.1 Introduction

In this section, the test strategy for *Runamic Ghent* is specified. This document is to be used by all developers in the project as a textual aid for performing tests. During this project, *agile development* will be used. Sprints will take around 2 weeks and in each sprint tests will have to be performed. In agile development, an iterative and incremental approach to software development is taken. The software is developed in small increments using the principles of continuous design and testing.
Here, the strategy for testing will be discussed. This is the plan that describes how testing objectives will be met effectively. Our test strategy will envelop three areas that have to be tested. Firstly we have the *server side* program, which contains all server-side methods. The second one is the *client-side application*, the application used in the project. The third area is *the system as a whole*.
Responsibility for testing is shared, meaning everyone is responsible for tests in the necessary places in their code, and takes the quality of this code into account. Test should be automated as much as possible and should be handled with the same importance as production code.

2.2 Quality and test objectives

- Correctness: Features and functions work as intended.
  - Measure and target:
    - 100% completion of agreed features
    - SonarQube reliability rating A
  - Priority: Must have
- Integrity: Ability to prevent unauthorized access, prevent information loss, protect from virus infection, protect privacy of data entered.
  - Measure and target:
    - SonarQube security rating A
    - All access from client to server via HTTPS
    - User login is managed by third party (firebase)
  - Priority: Must have
- Maintainability: How easy is it to add features, correct defects
  - Measure and target:
    - Code duplication $< 5\%$
    - Clean code (e.g. no cycles)

        * Modularity
    - Priority: Should have

- Availability: Percentage of planned up-time that the system is required to be operational
    - Measure and target:
        * 99% uptime
    - Priority: Should have

- Interoperability: Ease with which the system can exchange information with other systems.
    - Measure and target:
        * User application renders and functions properly on Android 6.0 and later mobile versions
        * Android wear application supports 1.4 devices
    - Priority: Should have

- Performance: Measures the responsiveness of the system under a given load and the ability to scale to meet growing demand.
    - Measure and target:
        * Accuracy of navigation: 50 meters
        * Response time route calculation: 5s
        * Dynamic adjustment time: 2s
    - Priority: Should have

2.3 Test Types

- Line Coverage: aim for 100%

- Unit tests:
    - Unit tests should be performed for most methods. Methods that are excluded from unit testing are pure visual methods in android. It is especially important to test all methods with user inputs and methods that perform calculations.
    - Execution/timing:
        * at commit
        * automatically
    - Test tools:
        * Java JUnit
        * Robolectric
        * Android Testing support Library
        * Python PyUnit
        * Django built-in test tool

- Component tests:
    - Where a component is defined as several classes and/or methods that work closely together but do not communicate with other parts of the system (otherwise they are integration tests).

  - Execution/timing:
    * For important components in the system
    * Performed at commit
    * automatically
  - Test tools: see unit testing

- Integration tests:
  - Execution/timing:
    * Important interactions between different components of the system
    * at commit
    * automatically
  - Test tools: see unit testing

- UI tests:
  - Execution/timing:
    * Both predefined scenarios and monkey testing
    * automatically at nighttime
  - Test tools:
    * Espresso Test Recorder
    * UIAutomator
    * UI/Application Exerciser Monkey

- Code analysis:
  - Execution/timing:
    * at commit
  - Test tools:
    * SonarQube
    * Pylint

- Performance tests:
  - Server should be able to handle around 50 requests/second
  - server calculations for dynamic route generation must take less than 2 seconds
  - Execution/timing:
    * At night
  - Test tools:
    * Taurus

- User tests:
  - Starting from sprint 4
  - Execution/timing:
    * Continuous by test group
  - Test tools:
    * Fabric

2.4 What to test, what not to test

- Server-side
  - Routing will be thoroughly tested as this is the backbone of the application
  - All other server components will also be tested to ensure performance, correctness and robustness.
- Client-side
  - All data providers and other methods that perform calculations should be tested individually
  - Activities will only be covered by UI testing

2.5 User testing crash handling

When user testing is in place, bugs and defects will be recorded by the Fabric platform. This platform is checked by designated developers. When a bug or defect appears the developer of the code containing that bug will be notified to test and correct that bug.

3 Test plan changes

A first change in the test plan was the 100% line test coverage. This has been changed to 90% as we felt that 100% would be an infeasible goal. This 90% also refers only to the code that will be tested, e.g. activities are not counted. At the end of the project we reached a test coverage of 72.8% client side (SonarQube percentage), which means there we still did not reach our goal of 90%. This is mainly due to the fact that instrumentation/UI tests are not counted towards test coverage and a couple of our components do contain elements that cannot be tested without instrumentation test (like classes using the Google API for localisation)
A second change was the decision to not test the Android Wear device. Our Android wear application has only two functions, the first one showing data sent from the smartphone, the second one reading heart beat data. Therefore there are no classes to unit or component test. UITesting or integration would also not be worth the extra trouble of finding a framework, if there is any (no framework was found in the scope of this project). The android wear application is only tested manually but thoroughly.
In the first plan it was stated that UI tests would be run automatically at night, however we had trouble getting an android emulator running on the server and could not do this. It was then decided that the UI tests would be run by the developers preferably every time when a new feature was added but at least once every three days on the latest committed version by the test expert, to make sure the tests are executed regularly. This entails both monkey tests and test scenarios.
UI tests did show some instability, as in some versions of the application a crash occurred when running the UI tests (the scenario where a run is started). The problem with this crash is that it could not be reproduced whatsoever on a device (both emulated and real) even if the exact same instructions were followed as were executed in the UI test. We suspect it is thus the UI test causing the crash and did not yet find a fix for this.
Even though the original test plan stated all classes in the android application except for activities would be unit/component tested, this was more difficult than expected. The AndroidWearProvider class serves as a bluetooth interface to the android wear application. This bluetooth connection for example is not covered by tests. Other examples are the component

that regulates the audio directions.

Server side no changes worth mentioning were made to the test strategy.

4 User Testing

4.1 Tools

Fabric To handle the distribution, crash reporting, logging, metrics analysis and more we use Fabric. Fabric exists out of 3 main kits, namely:

- **Beta**: For creating groups of test users, distribution of different versions of the app and get an overview of who has installed and launched our app. A notification can also be sent to the user to tell him an update is available.
- **Crashlytics**: Crash reports bundled together by crash type, OS, device with the stack trace and information about all the threads on the user's smartphone.
- **Answers**: Create personalised metrics that can be added programatically in the app.

Aside from these kits, different sections concerning retention, growth, key performance indicator, latest release and engagement offer much more information.

Slack Crashlytics is integrated with our slack channel #crashlytics. Every time a crash occurs, a summary is sent to that channel and the link can be clicked to further investigate the crash.

Mail Mails are sent with daily crash digests, new testers downloaded the app, Answers anomalies and more.

4.2 Initial Strategy

4.2.1 Phases

Phase 1 Version 0.3 will only be distributed to our so called 'friendly' users who aren't too critical. These users won't quickly write off the app if the features aren't very refined or the app isn't super stable. Nevertheless, there feedback will be very important for us, not only for the development, but also for the later distribution in the 2nd phase.

Phase 2 Version 0.4 or 0.5 will be distributed to a bigger group of people that don't solely exist out of these 'friendly' users. These could be people who don't know very well or don't even know. These could be also be people that often run with a running app and have certain requirements for such an app. With the lessons learned from the 1st phase, we hope to make the distribution even smoother.
It is possible that we will send subreleases to our test users as well to get even faster feedback. When we update the app on Fabric, users will automatically receive a notification to update their app.

4.2.2 Release and distribution

Each time after our release, the branch *masterUserTest* will pull from the master branch and the debug settings will be removed from the code. The apk of this branch can then be uploaded in the Fabric plugin in Android Studio and release notes can be added.
Before distributing the app with Beta from Fabric, we will send the users an email with extra instructions and guidelines concerning what the purpose is of the app, the features it has, what they should test, how the wearable can be installed and used and more. We do this because the release notes that can be added to fabric distribution aren't shown very clearly in their mail, because release notes might still be too technical for some users and because this extra email beforehand results in a more personal connection with the test user.

4.2.3 Crash Reports

When a crash occurs we will get a short summary of it in the slack channel #crashlytics as mentioned before. In this summary the link points to the Fabric crash report with gives us more information like stack trace, information about all the threads, the class where it occurred, on which line, how many users have been affected and how many crashes have happened.
These crash reports will be investigated firstly by the the person who primarily wrote that class. We will try to find the root cause of the crash and we will investigate why this bug wasn't caught by our test plan. If possible, a test will be written for that specific situation and the bug will be fixed. The priority of which crashes will be handled first depends on how many users are affected, how many times this crash has occurs and how vital the class is to our product (e.g. settings dialog crash might not be as important as the running screen crashing for instance).

4.2.4 Metrics

Fabric offers many metrics like real time usage, crash-free sessions, time per session and much more. We've added some customised events to get more insight into the usage of users.

- Sign up event: When users signs up.
- Log in event: When users logs in.
- History view event: When the users opens the history screen, the number of runs contained in there is sent.
- End run event: When the user ends a run, info about the run like distance, duration, avg speed, avg heart rate are sent.
- Save run event: When the user saves his run in the end run screen, the same info as above, plus their rating is sent.
- Discard run event: When the user discards his run in the end run screen, this is sent.

Analysing these metrics could be very interesting to get an insight into how the user interacts with the app. More custom events could be added for the next release to get more insight.

4.2.5 Feedback

After a week, a questionnaire will be sent to the users that installed the app to get elaborate feedback. The feedback will be analysed and depending on what the feedback is about different steps will be taken. We will try to keep the questionnaire short but still concise.

4.3 Changes to Strategy

2 main things occurred differently than according to the strategy. First, Phase 2 only started from release 0.5/0.6 and even then we found it difficult to find non-friendly users to test our app. It is really difficult to get people running. Second, we only sent out a questionnaire in the last week. People hate filling in a questionnaire and even in the last week we had to constantly ask to people personally to fill it in. We also didn't really find it necessary to send out multiple questionnaires as we spontaneously got a lot of personal feedback via e-mail and Facebook.

4.4 Crashes

Fabric's crash reporting works really well and it helped us in some rare occasions. Rare, because in the releases we distributed not a lot of crashes occurred. E.g. in the first release to friendly users (release 0.3) not a single crash occurred. More metrics about crashes that occurred can be found in Section 5.2.
When a crash occurred, a summary would come in the #crashlytics channel on Slack. I would then check the crash and copy its stack trace to a new Jira issue. I would then contact the person who was the most responsible for the code where the crash happened and give him more information about the crash.

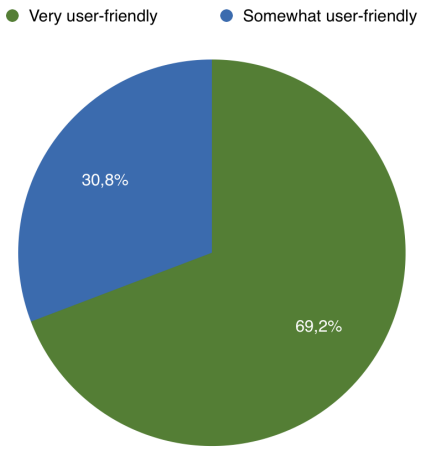
4.5 Evaluation of Tools Utilised

Fabric is still very new and many features aren't yet completely worked out yet. It is very easy to integrate into a project and it can generate a lot of statistics, but sometimes these statistics aren't very meaningful because they are only about the last week. Also some metrics are weirdly defined. E.g. daily new users also counts a user updating to a new version as a new user. As it is not possible to filter per version for this metric, new daily users can not be retrieved in the way we desire. This is something we can see however on the Firebase platform. As some metrics can only be accurately retrieved per week or day, Fabric might be more suited towards apps that have many users per day, which is not the case for us.

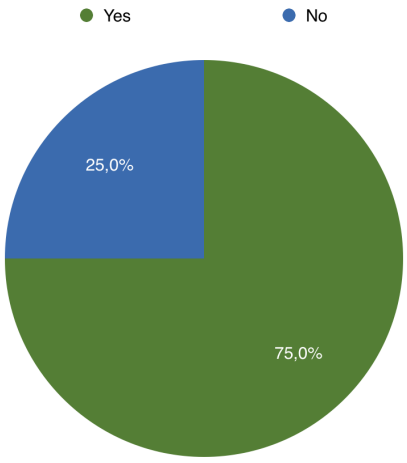
4.6 Feedback

As mentioned before, we decided that sending 1 questionnaire would be sufficient as people don't like filling them in and we trusted more on getting personal feedback. However, questioning our users about how they experienced our app at the end provided some nice insights. Most of the test users that effectively tested our app and we had contact information from were (semi) friendly users. The questionnaire was therefore not divided between friendly and non-friendly test users as the number of replies would not be statistically relevant. However, the questionnaire was anonymous so we do expect a quite honest answer from all the test users.

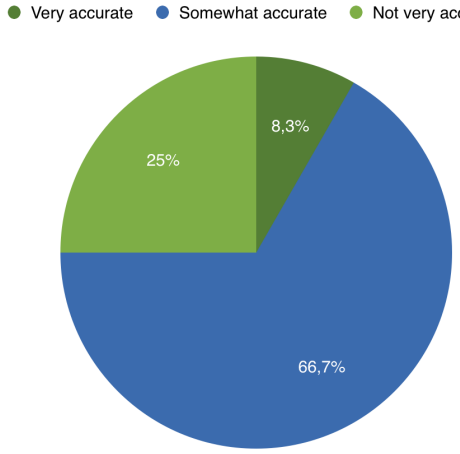
Results In Figure 1 the most interesting results from the questionnaire can be found. The caption also denotes which question was asked and how many people responded. The answers in the legend are only the possibilities that people answered. For the question in Figure 1a, 2 other answers that nobody marked were 'not very user-friendly' and 'totally not user-friendly'. For the question in Figure 1c, 1 other answer that nobody marked was 'totally not accurate'. For the question in Figure 1d, 1 other answer that nobody marked was 'totally not pleasant'.
It is clear that the test users find our app to be user-friendly. This is an important result as one of our system quality attributes was usability. The fact that it motivates so much users and they generally find the routes pleasant are also satisfying results. The test users also found



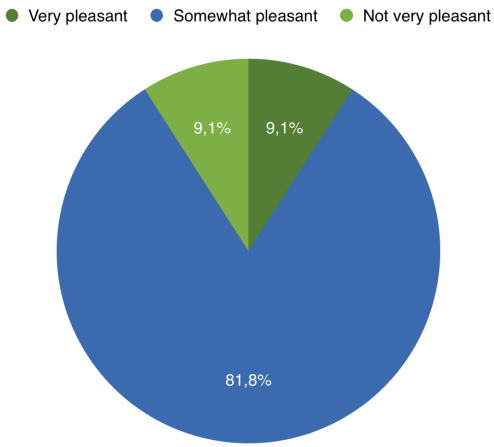
(a) Did you perceive our app to be user-friendly? (13 people)



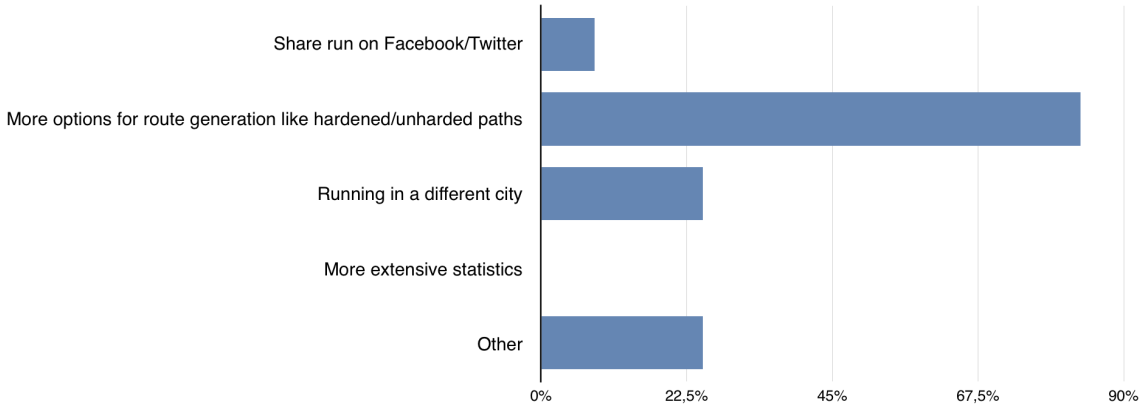
(b) Did the generated routes motivate you to go running more? (12 people)



(c) Did you find the statistics of your run accurate? (12 people)



(d) Were the generated routes that you ran pleasant? (11 people)



(e) Which other features really need to be added? (12 people)

Figure 1: Plots for the most interesting responses to questions asked in the questionnaire

the statistics to be generally accurate, although we know that we could still improve this a lot. There was no time left, but there were some things we could implement a bit differently and fix to make the statistics more accurate. Finally, it is clear that the test users would prefer to have more options for the route generation. We added the feature that the user can input how much he prefers going through parks and past water. Unfortunately, this feature was added after the questionnaire was given.

5 Quality

This section provides some metric of the project. These metrics are in accordance with the test strategy described above and were recorded on 21 May 2017.

5.1 Code

This section describes the results from SonarQube, our static code analysis tool.

Client The client consists of 8580 lines Java code. It has 72.8% line coverage, with the core components `EventBroker` and `GuiController` having over 95% coverage. The `Reliability` and `Security Rating` are both `A`, meaning the static code analysis has found no more bugs. As most of us made it a practice to thoroughly check SonarQube when we pushed a new piece of code, we have a small `Technical Debt` of 0.6%. We managed to get `line duplication` down to 2.2%, with most of the remaining duplication being Android activity life cycle code.

Server The Python part of the server has 1555 lines of code and 84.5% line coverage. The Rust part has 70% coverage (300/436, note that boilerplate like structure definitions and trait implementations are not covered). Note that the Rust version contains a lot of boilerplate. As the relation Python-Rust is 20%- 80%, we can say the server has about 81.5% line coverage. Once again, we have removed all bugs the static code analysis could find, resulting in an `A` for `Reliability` and `Security Rating`. The `Technical Debt` is a bit higher at 2.5%, but still acceptable. `Code duplication` is non existent at 0.0%.

5.2 User experience

This section contains some metrics derived from Fabric, our user testing platform.

First some general statistics to provide a context for the metrics. From the first release on 27 March 2017 to now (21 May 2017), there were 457 user sessions for a total of 33 unique users. The average number of daily active users was 2.67.

There were 4 issues discovered by the users that resulted in 7 crashes. This means there was only a crash in 1.5% of the sessions. Apart from the crashes, 6 bugs were reported by our users. Four of these were considered major and fixed immediately. The remaining 2 were rather cosmetic and have been fixed eventually. In general we have 0.3 bugs found per user. An non exhaustive list can be found in table 1. Next to these bugs, we also received dozens of recommendations for improvement and feature requests. We don't have an exact number for these as a lot of them were reported informally by friendly users and implemented immediately. Once again, a non exhaustive list can be found in table 1.

Bugs	Feature requests
Total distance not correct	Auto-rotate map while running
Log-in with facebook not working correctly	Show length of generated route
Crash when 'peeking' a run of 0 km	Error message for route generation
Average speed and heart rate not correct	Snap to location
Numberpicker invisible on some phones	Back button exits app
Spaces not stripped from input fields	Generate alternative route button
High battery usage	

Table 1: Non exhaustive list of user bugs and feature requests