ResubmissionChanges

Sections added:

1 . Test Design.

Added More text to:

Reflections on RFID event firing double.

Code Added:

Added more BVA tests:

testMinimumChargingCurrent() testAboveMinimumChargingCurrent() testBelowMinimumChargingCurrent() testChargingMaxAllowed() testChargingBelowMaxAllowed() testChargingAboveMaxAllowed()

Added ZOMBIE MANY tests:

$\text{Station}_{O}nce_{R}FID_{C}orrect_{O}nce_{R}FID_{W}rong_{D}oor_{C}losed()Station_{D}oubleRFID_{C}orrect_{D}oor_{C}losed()$

Charging Station

Software Test

Aarhus Institue of Technology

Author:
Sune Dyrbye,
Morten Høgsberg
Yevhen Parolia

Date: April 6, 2022

Contents

Introduction

This journal has been writen by the SWT group **SWTF22 3**, consisting of

- Morten Høgsberg Futtrup Kristensen (stud. nr.: 201704542)
- Sune Andreas Dyrbye (stud. nr.: 201205948)
- Yevhen Parolia (stud. nr.: 20112870)

It is the result of the mandatory assignment **Obligatorisk Handin 2** in the course **ST4SWT-02 Software test**. Design and reflections is presented here, while the sourcecode is available in the github repository at:
https://github.com/UniquesKernel/ChargingStation

The system has been tested through a Jenkins CI job at
http://ci3.ase.au.dk:8080/job/SWTF22_3_handin2/

Charging Station Design

Our design is based on the design suggested in the suplied material, and relies heavily on interfaces to allow testing the system through hardware simulations. It is thus easy to create a piece of code that sends the same signals to the system as the physical hardware implementation would, allowing for testing of the individual units before implementation. This is taken further, as the internal components of the system (*stationControl* and *ChargeController*) interacts through an interface, allowing the *chargeControlle* to be replaced for upgrades or testing of *stationControl*.

The design of the charging station is displayed in Figure 1. Central to the system is the *stationControl* class, which handles all interaction with the user, through various interfaces (display, door and rfid detector). It also interacts with the *chargeController* through another interface. The *chargeController* meanwhile interfaces with the USB charger, controlling it, and also displays charging status on the display through the display interface.

The charging station is first activated by the door opening, causing the station to ask the user to connect the phone. The user then connects the phone and closes the door, which causes the station to ask for an RFID. If the phone has not been correctly connected (as cheked by the chargeController) the station reports this to the user through the display. If the phone has been properly connected when the RFID is detected the door locks and the station starts charging the phone. This continues until the charging is complete or the same RFID as was used to start the charging is detected. When the RFID of the user is detected after charging has begun, the system stops the charging and unlocks the door, allowing the user to disconnect their phone. This sequence is illustrated in Figure 2.

Test Design

To test the charge controller we used BVA and EP to test the boundary testcases above, on and below the maximum and minimum current values, as well as testing the connection threshold value. We used the EP principle to equate the allowed values of the charger current. To test the event of RFID we used the ZOMBIE Many principle to simulate two correct RFID and one correct and one wrong RFID tag events. For the door event we did not use any additional test principle as the door operations and lock and unlock functions are expected to operate mechanically and fault in this systems will introduce testcases that are not a scope of this assignment. Display simulator test is a simple whitebox test, any attempt to test the consolewriteline method will move the focus of the assignment to begin testing Windows System Calls which we do not want to. This will reduce the coverage of the display simulator class but is acceptable.

Design Reflection

The major component in design philosophy for the project have been to allow for a low coupled and testable design. In extension of that the use of events in the observer pattern means that it has been unnessesary to use threads to handle multiple events.

SOLID

Take cognizance of the SOLID principles. We have divvied up the respondsible of each part of the ChargingStation system into seperate classes.

This seperation of responsibility between the class makes it effortless to change one part of the system without impacting or breaking the other functionalities. It also ensures that each class can be tested seperately.

All components of the system is assembled with the use of Interfaces to ophold the open and close principles. It makes it painless to extend the functionalities of the system without having to change existing implementaions.

The principles of Interface segregation and dependency inversion are used extensively to ensure that Station controller (a high level module) in no way depends on specific implementations of its components but rather interface abstractions.

Events

To handle event, we implemented the observer pattern to handle any changes in values either from user actions or through time based events such a regular variable updates with regular time intervals.

While such things could have been handled with the use of threads, we chose to use event driven programming instead as implementing threaded system can be more complex and harder to test.

It was noted upon further reflection that premature charging interruption could occur if the RFID tag event was fired very fast one after another, and if this was not the intention of the user. One solution to this problem could be a timer for the event, disabling the event for some time or simply ignoring the event.

Conclusion

While the SOLID principles makes everything loosely coupled, it does mean a marginally more complex codebase as the solution increases in scope. It is however, the better solution for the reasons mentioned above, making it lossely coupled and easilier testable.

The use of event driven programming rather than threaded programming follow the same argument where testability and simplicity was chosen of complexity.

In conclusion, due to these design considerations it means that mock classes could be used to ensure a high test coverage where each aspect of a class is regiously tested.

Work Distribution Reflection

At the start of the project the group worked together to make and understand the overall design of the system, based on the assignment handout. We then distributed the modules evenly between us and implemented and tested them individually. The *stationControl* module was left out, as this module is at the centre of the system and we felt that it would be best if all group members understood this module well. We therefore designed and wrote this part in colser colaboration than was pressent during the development of the other modules.

This approach of divide and conquer, with an occational "concentration of force" has worked well for the group, allowing us to work as suited the individual best, but also taking advantage of the small group size to be able to work "all hands on deck" on the key elements.

In our testing approach we tried different test techniques to see what worked for us. First when using mocks were not feasable we used tradition unit testing to test components that didnt rely on other components. when using mocks we first wrote mock classes but found this cumbersome and soon switched to using NSubstitute; a framework for doing mock testing.

One component the Log class lent itself poorly for testing as the system did not require a way of retrieving the log. This made excruciatingly painful to test and so the test was left out. After all there is no reason to test something that can't be retrieved anyway.

The jenkins server felt cumbersom, as threr was very little integreation to test, as most, if not all, tests were tests that we could do locally. It did, however, catch an error that prevented building the project almost imidiately. In other words, it worked as it should, but the limmited scope of the assignment didnt allow jenkins to shine as mutch as it could have.

Using a shared repository is by now standard opperating procedure for the group, and it works quite well. We all instinctively know not to work on files that other group members are using, and have enought knowledge of git to handle the merge conflicts that occationaly arise anyway. It works well with the divide and conquer approach that we used (and that was encouraged, not to say demanded, by the assignment), but does make it slightly more troublesome to quickly hand your work over to someone else.

References

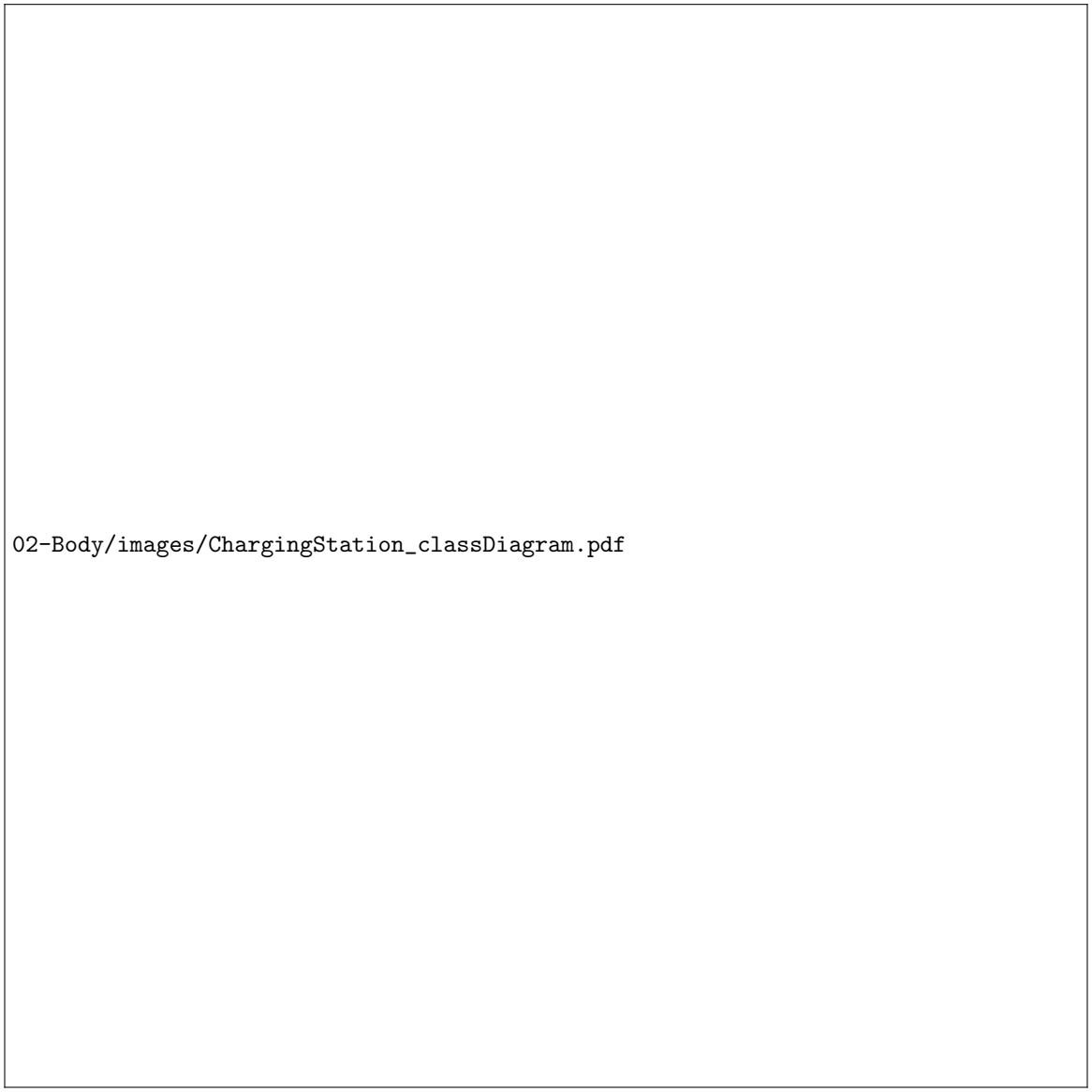

Figure 1: Class diagram for the charging station system

02-Body/images/SEQpdf.pdf

Figure 2: Sequence diagram describing normal opperation of the charging station