Priotool - An Inventory & Repair Management System for Blue City

Frederik Bode Thorbensen, Hans Erik Heje, Markus Emil Hye-Knudsen, Ming Hui Sun, Jakob Elias Gylling Saadbye, Sture Skar Svensson, David Doctor Heyde Rasmussen

December 23, 2021

AALBORG UNIVERSITY
DENMARK

Department of Computer Science

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Priotool - An inventory & repair management system for Blue City

Theme:
A well structured application that solves a real problem

Project Period:
3. SEMESTER

Project Group:
sw-k3301

Participant(s):
Frederik Bode Thorbensen
Hans Erik Heje
Jakob Elias Gylling Saadbye
Markus Emil Hye-Knudsen
Ming Hui Sun
Sture Skar Svensson
David Doctor Heyde Rasmussen

Supervisor:
Daniel Russo

Copies: 1

Page Numbers: 102

Date of Completion:
December 23, 2021

Abstract:

In this project we collaborate with the secondhand consumer electronics company Blue City, a company that has a deprecated system for managing its repairs. This project aims to analyse the flaws of their current system and evaluate what a better system requires. The entirety of the problem and application domain is analysed thoroughly through interviews which condense to the following problem statement: "*How do we make a system which increases the productivity of Blue City's technicians and improves the repair flow of their workshop, lowering repair times and potentially increasing profits*".

We followed principles from object oriented analysis & design and fundamental principles of designing a good user experience to create highly usable designs and solid system models. Implementing the design using technologies such as SpringBoot, Java, React.js and MongoDB, we create a system that largely satisfies user and system requirements. Testing ensures that the functionalities of the application fulfil the users' requirements. Unit testing allowed us to confirm the reliability of the application. After participating in the usability test, Blue City expressed great excitement towards our current solution, indicating that the user experience is impeccable and could be interested in adopting the system with just a few extra features. Even though some features are missing, the web application is still highly usable and reliable, potentially resulting in lower repair times and improved workflow of the Blue City repair shop.

Preface

We would like to thank Blue City for letting us collaborate with them to make a product and would like to thank the participants that were willing to help in performing the user test. Secondly we would like to thank our supervisor for his inspiration to the project and knowledge on system development.

Contents

1 Introduction

For this project, the group will be collaborating with a company called Blue City. Blue City is a company that primarily buys, sells and repairs used consumer electronics. When Blue City receive a product, either by buying it from a customer or receiving it from another company, their grade is based on certain physical appearance and functionality. If the product is not working, it receives the grade "D" for defective and will have to be repaired by a Blue City technician in order to be functional. A product is defective if the outside is physically broken or if one or more parts inside the product are faulty. In most cases, to repair a product, parts will need to be replaced. Once repaired, that product will then be sold in the store for profit.

Blue City's most essential employees in terms of repairing defective products are the technicians and the manager. They both perform different tasks within Blue City, with the technicians' most important tasks typically being; performing all the repairs on defective products and writing off products. In contrast, the manager's tasks consist of maintaining an overview of the technicians' performance, ordering spare parts and approving the writing off of products.
However, Blue City currently lacks a practical system to allow the technicians and manager to perform their tasks efficiently. Blue City's current system, called Priotool, is disorderly and has several integral usability issues. These issues include but are not limited to; taking over 5 minutes to load their inventory and only being accessibly by one user at a time as discussed in Appendix A. Therefore, this project will aim to create a system capable of effectively completing technician and manager tasks.

Throughout this report, we will analyse and discuss the current Blue City system and our system. As the system will be built for usability, we will also discuss the choices we make during the design of our user-interface. Subsequently, we will discuss the architectural design and implementation of our system. Afterwards, user-testing of our system and any potential further development will be discussed. Finally, we evaluate and conclude if the aim of the system has been met.

1.1 Limitations

A natural limitation, given by Aalborg University for the third-semester project, dictates that the solution must be a standalone system. This removes any option to integrate with an already existing system.

2 Background - Blue City

2.1 Current situation

Blue City Vesterbrogade's workshop consists of four technicians, a workshop manager and a workshop assistant. The technicians work simultaneously on individual repairs and each technician is specialised in repairing a specific product type. The backlog of defective products which are in need of repairs is at least 150 at any given time. Each technician is responsible for their own work schedule and prioritising the most pressing repair based on their specialisation.

The workshop handles three types of repairs which are reclamations, in-house repairs and street repairs. Reclamations and street repairs receive the highest priority from the technicians, as these products are owned by customers, who needs them back as soon as possible. These two categories of repairs are contained in a separate system to in-house repairs. The in-house category consists of repairs on products Blue City owns and wants to re-sell once they have been fixed. Their current system relies heavily on the technicians or the manager manually entering the products and spare parts into an excel sheet, which they then use to identify the most pressing repairs. Because of the flaws of the Priotool, a majority of technicians do not use the current system.

In order to appreciate the situation and obtain valuable insight into the current situation we arranged a meeting with one of Blue City's technicians, to conduct a semi-structured interview. The semi-structured interview technique is picked as there are a few key concepts which are crucial to obtain an understanding for, but a need for a broader understanding as well. In preparation for the interview we formulated several questions with the main purpose of ascertaining the struggles a technician faces when using their current repair system.

2.2 Rich picture

The semi-structured interview provided a general idea of their current system. This knowledge was used to visualise Blue City's workshop workflow in a Rich picture. The rich picture illustrates the process a product goes through from the time it enters the workshop, when it is repaired/serviced, until it is finally ready to be sold.

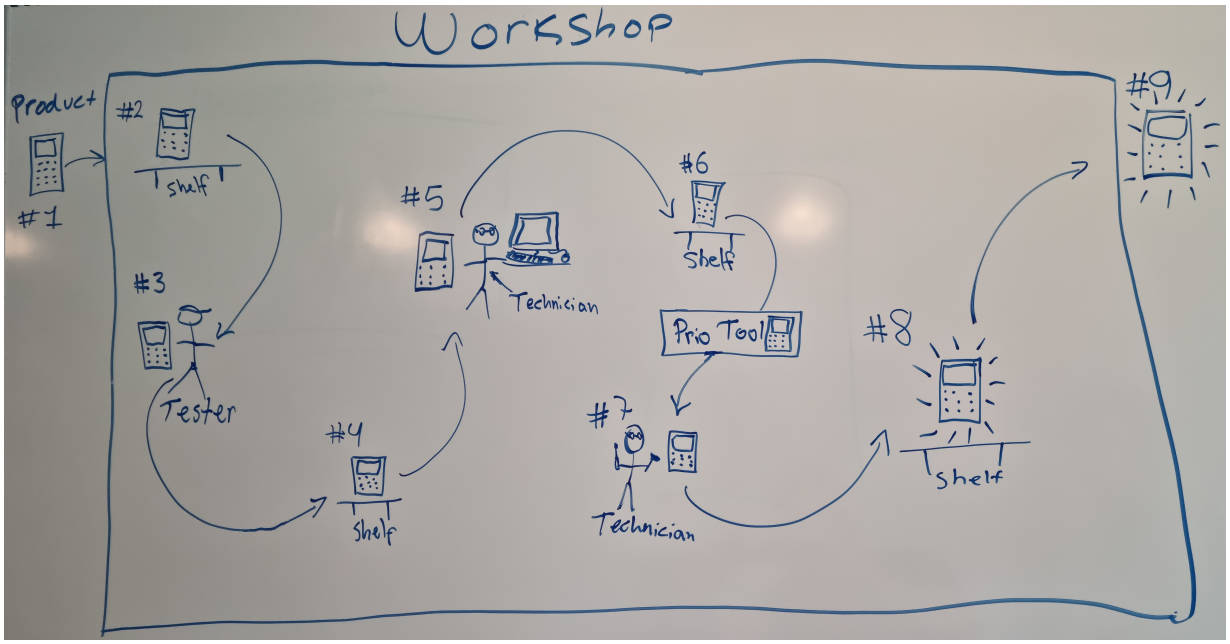


Figure 2.1: Rich picture of Blue City's current workflow

Based on the initial interview conducted with Blue City and the current situation described in section 2.1, we create a rich picture [1, p. 26] which can be seen on Fig. 2.1. The main purpose of this rich picture is to visualise the journey of a product, from the start where the product enters the workshop, to the end where the product is repaired and exits the workshop. This visualisation is instrumental to understand the workflow of Blue City's technicians and provides an indication of where the workflow can be improved. The workflow is comprised of several major tasks which including testing, repairing and registering products. Testing and repairing products are both physical and mandatory aspects of the process and as such can not be impacted heavily. With this insight into the current workflow we try to identify the root of their problem, which must exist somewhere after testing and before physically repairing the product. To determine exactly where in this repair process the problem lies, we made another rich picture detailing the registration aspect, where the inherent problems and struggles must reside, see figure 2.2 below.

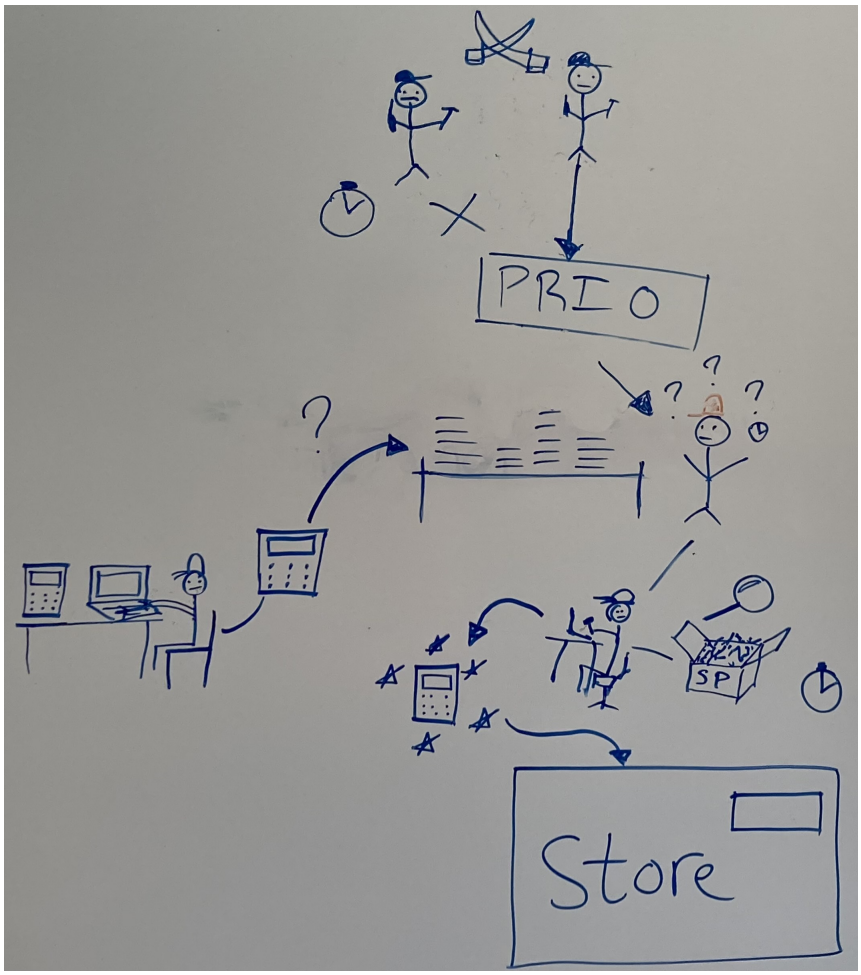


Figure 2.2: Rich picture to identify problems in the current workflow of Blue city

From this rich picture, we outlined our understanding of their current struggles:

- Where the spare parts can be found in the warehouse
- Where the broken and fixed products can be found in the warehouse
- The amount of time trying to get their "Priotool" to work
- That only one person can use their "Priotool" at once
- That the technicians do not know which product to start repairing first

From this we started to get a really good idea of their current problem and constructed the following problem statement.

2.3 Problem statement

How do we make a system which increases the productivity of Blue City's technicians and improves the repair flow of their workshop, lowering repair times and potentially increasing profits.

2.4 System definition

In this section the system is described through a system definition which is further expanded in a factor analysis [1, p. 40]. These two methods of defining the system are based on our first interview with Blue City as well as our initial research into the struggles of Blue City's workshop. The methods work in cohesion to provide an understanding of the problem and the different aspects of it.

The system is defined as; a computerised system primarily used by Blue City technicians to prioritise repairs, manage products and manage spare parts. Secondarily used by the management to visualise workshop productivity. The system should run through the browser on Blue City's PCs.

2.4.1 FACTOR Analysis

Functionality: The system can recommend repairs for technicians based on; defective products with the longest storage time, and the highest predicted profit. Furthermore, the system can manage an inventory of spare-parts and defective products, register technicians with credentials and allows technicians to add a comment to defective products. Additionally, it can update cost prices of products after a part is replaced and keep track of time spent on each repair. The system allows Blue City management to monitor technician and repair statistics. The system supports multiple employees accessing the system simultaneously through a web-based client-host architecture.

Application domain: The system is provided as a web application used by technicians across Blue City's stores. They use it to prioritise which products are more important to repair. The Blue City management will use the system to monitor the workshop's productivity.

Conditions: The web-application will be developed by third-semester Software students from AAU in collaboration with highly technically enabled technicians from Blue City and their corresponding manager.

Technology: The web-application will be accessed with credentials through a browser on any Blue City work computer. The system will require an account to have access.

Objects: Product, Technician, Spare part, Repair, Write-off ticket, Order list.

Responsibility: The responsibility of the system is to reliably recommend which product needs to be repaired in accordance with assignments from upper management. To do this it needs to keep track of what parts are in stock such that the needed spare parts are available. Additionally, the system helps the technicians find the product and the needed spare parts in the workshop such that the least amount of time is used on a repair. To further optimise for profit, the system will help management keep track of productivity to get an intuitive overview of what can be improved and what is going well.

3 Problem-Domain Analysis

The problem domain is defined as; "That part of a context that is administrated, monitored, or controlled by a system" [1, p.47]. In order to understand the problem domain, we start by interviewing and discussing with the prospective users at length. From these discussions and interviews we gain a deeper understanding, which we use to construct models of the problem domain. As we manifest our understanding through the construction of problem domain models, new questions and areas of lacking understanding arise as a bi-product. This leads us to partake in further interviews to gain a more complete understanding of the problem domain. This process repeats throughout the problem domain analysis until the system is completely or sufficiently understood and the problem domain models reflect our understanding of the area that is administrated, monitored, or controlled by a system. Problem domain analysis is divided into three activities: The class activity, structure activity and behaviour activity. In the following section we will discuss these three activities in reference to the problem domain.

3.1 Classes

The class activity describes how we approach the task of defining a system's information content. We define and represent the problem domain by selecting classes and events. Classes define groups of relevant objects found in the problem domain. Events are viewed as the basic element of object behaviour. "The result of the class activity is an event table that contains the selected classes and their related events"[1, p. 49].

Identifying Objects

To find out which classes would be relevant for our problem domain analysis, we started by identifying the objects and events from our background research and interviews on the current situation and problem that the client has. We found out that the client's problem primarily revolved around the use of their *Priotool*. Therefore the objects and events that would be relevant to our problem domain would consist only of those that interact with *Priotool*. This means that the store clerks, high level managers and product testers were excluded. Similarly, as their *Priotool* only handles products of the grade "D", meaning defective, are included, while grades "A", "B", and "C" represent functional products of varying conditions are disregarded. Through these considerations we found the following objects related to the problem domain:

Technician, Manager, Repair, defective products, spare parts

From our background research, we also found out that each type of product contains a finite range of different spare parts that could be reused in other products of the same type, for example, an iPhone would contain the spare part objects: back camera, front camera, battery, screen, home-button etc. These would all be single objects related only to that specific iPhone model. This could be done for every different product that the technicians repair.

Finding Classes

Classes are a way to describe a collection of objects found in the problem domain, which share similar structure, behaviour and attributes. In the problem domain, the following class candidates were identified:

Class candidates:	
Workshop	Employee
Technician	Manager
Product	Spare part
Repair	Write-off ticket
Spare part inventory	Product inventory
Inventory	Order

Workshop - Since Blue City has multiple Workshop locations where they perform repairs it would be a class representing the location and workshop number of a specific workshop. The class will not be included since we will only be working with a single workshop.

Employee - There are two types of employees involved with repairs we could model with this class, technicians and managers. It would contain account information such as employee id, name and performance metrics such as number of repairs, number of write-offs etc. In the end we only monitor technicians and there is no need for an employee super class.

Technician - The purpose of this class is to create a profile containing information about a technician. Objects of this class will contain user information as name, initials etc. and allow repairs to be assigned to a specific technician.

Manager - We choose not to include this manager class as we will not be tracking manager profiling information and therefore will not be monitoring the manager. As we'll see, the manager plays a role later during the application-domain analysis.

Product - As we found through the identification of the different objects, Blue City repairs a plethora of different products such as smartphones, computers, etc. But in regards to the function of the repair shop, they all share structure, behavioural patterns and attributes, therefore we found it would be logical to group these objects under this class.

Spare part - The spare part class represents the collection of individual spare parts, such as screens, cameras and all the other types of spare parts consumed during a product repair. Since all individual spare part objects contain the same fields, it would be logical to group them under a single class.

Repair - The repair class represents the collection of individual repairs. A repair is an abstract object representing the activity of repairing a product that will often occur in the Blue City workshop, and in nature, the repair objects share the same structure, behavioural pattern and attributes, as the class Repair would.

Write-off ticket - The purpose of the write-off ticket class is to capture information about the product that needs to be written-off along with the spare parts that are still functional in the product and needs to be created by the system.

Order - The order class contains information on spare parts that should be ordered through the parts vendor. An order can be confirmed and be modified when spare parts gets added or removed.

Inventory - This general class would represent the concept of an overall collection of products and spare parts in the workshop. Since the inventory class would only contain other classes, we found it to be redundant as we would store no further information than what was already contained within the classes it stores.

Product Inventory - This subclass would be the collection of all the products that needs repair. We remove this class for the same reason as to remove the inventory class.

Sparepart Inventory - This subclass would be the collection of spare parts available to the technicians. It would contain both new and used spare parts. As with the other inventory classes we chose not to include it in the final list.

There were many considerations during this phase, and many iterations was made picking classes. In the end, this is the classes we chose to include based on the considerations above.

Final classes:	
Repair	Product
Spare part	Technician
Write-off ticket	Order

Finding events

To find the events relevant to our problem domain we look at each of the identified objects and consider which work related actions affect them. We consider which events create the objects, affect them while they exist, and eventually remove them. In evaluating our candidate list of events we consider that they must be instantaneous, atomic and identifiable as they occur. Each event candidate was considered and evaluated systematically.

Recommend product for repair: This event would occur internally in the system when a technician wants to see the list of products to repair. We left this off as it was better suited for a compute function in the application-domain and not something that happened in the problem-domain.

Repair started: A central event involving objects from product and spare part. The event occurs when a technician begins his process of repairing a product. This event should be included.

Repair finished: This event occurs in conjunction with starting a repair since a repair must be started before it can finish. The finished repairs in the workshop will be recorded for statistics. The product that was under a repair is handed back to the store front.

Repair paused: An event that happens when a technician has started a repair but the required spare parts are unavailable. In the workshop, the technician puts the product on a separate shelf to indicate that it is missing a required part.

Repair resumed: This event occurs when a required spare part has been received for a repair that has been paused. The technician takes the product on the waiting shelf back to the repair desk and continues the repair.

Repair cancelled: This event occurs in the workshop if a technician starts a repair by accident or the product actually needed to be written-off. If spare parts have been applied, the technician removes them.

Write-off created: This event occurs when a technician decides a product is not worth repairing and needs to be written off. Instead of simply removing the product from the system, we register which technician did it and which spare parts are still functional for use in repairs.

Write-off approved: This event occurs when the manager agrees with the technician who initiated the write-off. The product is written off and shelved, and the functional parts it contains can be used for other repairs.

Write-off declined: This event occurs when a manager does not agree with the technician regarding a write-off. In the workshop this event rarely occurs due to the size of the team, but needs to be dealt with in case an error is made in the write-off process. The product is put back on its shelf awaiting repair.

Technician registered: This event occurs when a manager registers a new technician. It is needed for technicians to be able to log in and so that the system can register which technician is involved with a repair or write-off.

Technician removed: Occurs when a manager removes a technician who is no longer employed in the workshop.

Spare part added to order list: Happens when a technician or manager adds a spare part that is needed in the workshop to the order list.

Spare part removed from order list: If a spare part isn't needed after all it needs to be removed to avoid ordering unnecessary parts.

Order list confirmed: Happens when all spare parts on the order list have been ordered from their respective manufacturers and they should no longer appear on the order list.

Spare part salvaged: Occurs when a write-off ticket is approved and the functional spare parts of a product are made available for technicians to use in repairs.

Spare part added to shop: This happens when a new spare part is received in the workshop and the manager imports it to the system's spare part collection.

Spare part added to repair: Happens when a technician has found a fitting spare part and attaches it to a product during a repair.

Product imported: This event occurs when a manager adds a new product to the system and information about its state and defects is registered.

Product cost updated: Happens when a repair is finished and the product's cost price is updated based on the cost of the used spare parts. We chose not to include it as it is the byproduct of finishing a repair and not a standalone event.

Defective comment added: This event would occur when the manager adds a defective comment to a new product

entering the workshop. The defective comment is registered on a product so that the technician knows what to repair. We chose not to include this event as it is part of importing products and therefore not considered atomic.

Events:	**Technician**	**Product**	**Spare part**	**Repair**	**Order list**	**Write-off**
Repair started	*	*		+		
Repair finished		+	+	+		
Repair paused				*		
Repair resumed				*		
Repair cancelled		*	+	+		
Write-off created	*	*	+			+
Write-off approved		+	+			+
Write-off declined		*	+			+
Technician registered	+					
Technician removed	+					
Spare part added to order list			+		*	
Spare part removed from order list			+		*	
Order list confirmed			+		*	
Spare part added to shop			+			
Spare part added to repair			*	*		
Spare part removed from repair			*	*		
Product imported		+				

Table 1: Event table

3.2 Structure

When defining the structural relations between classes and objects we study the static relations between classes and the dynamic relations between objects in the problem domain. To determine the structure of our problem domain we look at how all of the classes are related to each other. The class diagram in figure in Figure 3.1 shows how a **write-off** is associated to the technician who creates it, aggregates the product undergoing a write-off process, and associates to the functional used spare parts that can be salvaged from the product. A **repair** is associated to the technician who starts it, while aggregating the product being repaired and associates to the spare parts consumed in the process. Both kinds of spare parts are generalised into the spare part super class through which they associate to the repair in which they are consumed. Individually, a **used spare part** is associated to zero or one write-off, but always its origin product, while a **new spare part** is associated to the order list while they are to be ordered or en-route to the workshop. A **technician** can be associated to zero or more repairs and write-off tickets at the same time, while a **product** is aggregated to zero or one repair or write-off ticket.

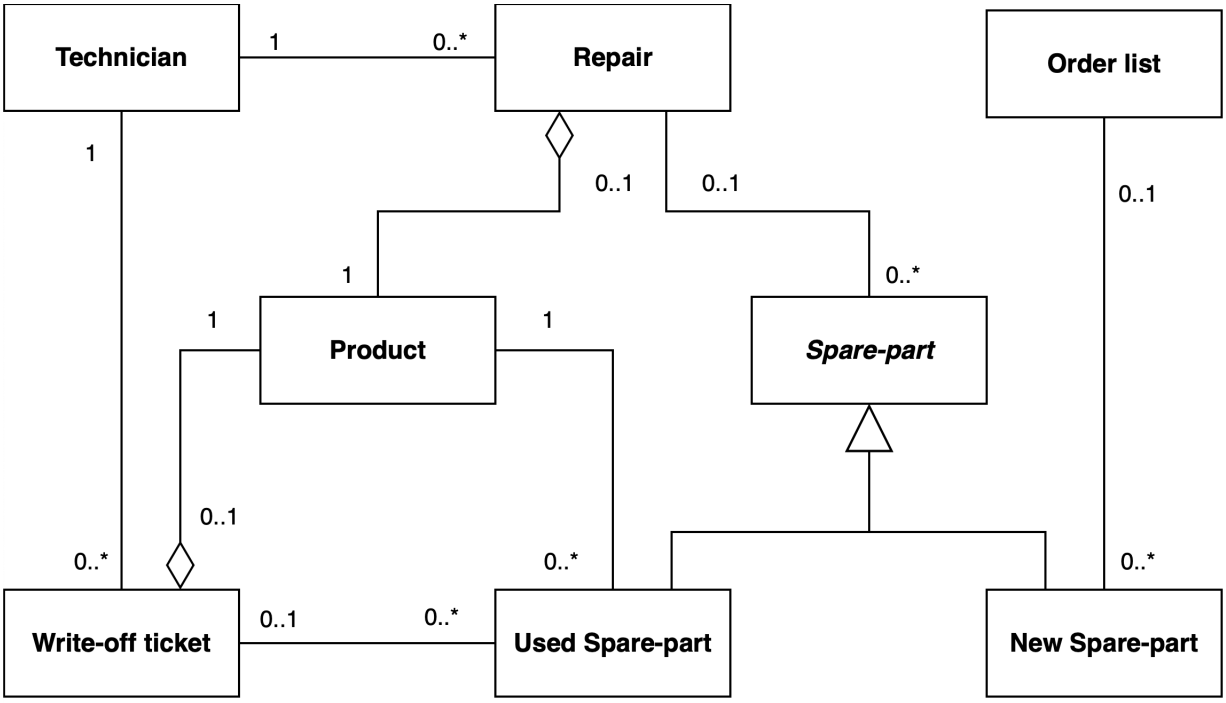


Figure 3.1: Class diagram

3.2.1 Aggregation structures

Product - Repair: Since a repair is always related to product, we have defined the relation between product and repair as an aggregation.
Product - Write-off: Similarly, since a write-off always concerns a product, we have denoted their relation as an aggregation.

3.2.2 Association structures

Technician - Repair: We have defined the relation between the technician and repair classes as an association structure in that each repair is performed by a technician, but the repair is not a part of the technician class. A technician can thus have relations to zero or more repairs, while each repair is related to exactly one technician.

3.3 Behaviour

In the behaviour activity we construct behaviour diagrams for each object's behaviour based upon the events they occur in. Events that happen zero or more times for a given object are denoted with (*) and events that happen zero or one time is denoted with a (+). These are reflected in the updated version of our event table as seen in Table 1. Additionally, the classes are updated with attributes relating to the events. By looking at the events, we get a better picture of which attributes the class needs to contain, so that the system model can keep track of changes that occur in the problem domain objects. We decided only to model the classes "Product", "Spare part" and "Repair" as these classes have the most events.

Product behaviour

Figure 3.2 shows an example of a state-chart diagram for the class "Product". The product class is mainly involved in the events that happen when a write-off takes place or during a repair. Since both the write-off process and the repair process can be cancelled the product can come back to its original defective state. The product exits the system when a write-off gets approved or a repair is finished. Ideally, the system should record every product that has been written-off or repaired for later use in statistics but at this point it's still unclear if we should save or delete these products.

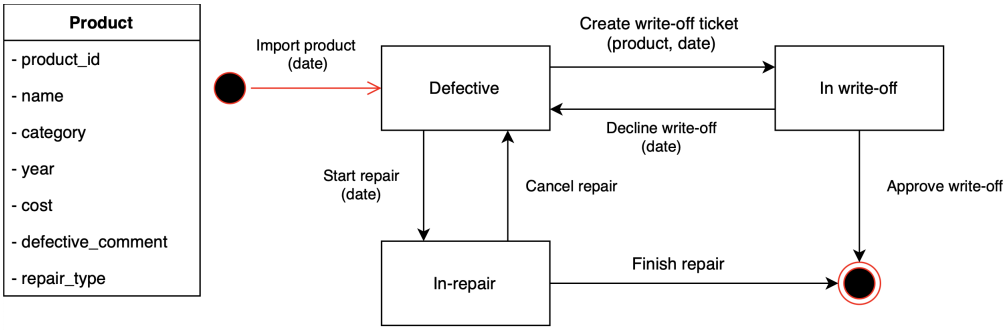


Figure 3.2: Statechart diagram for the class "Product"

Repair behaviour

The behaviour for the class "Repair" is shown in Figure 3.3. An object of the repair class represents a single repair process in which spare parts that are used are applied to the product being repaired. A repair is logged as a historical document intended to describe details on the technicians repairing process. The system should ideally remember all these changes in repair states for logging purposes. Similar as with deletion of products, we want to store finished repairs for statistics later on.

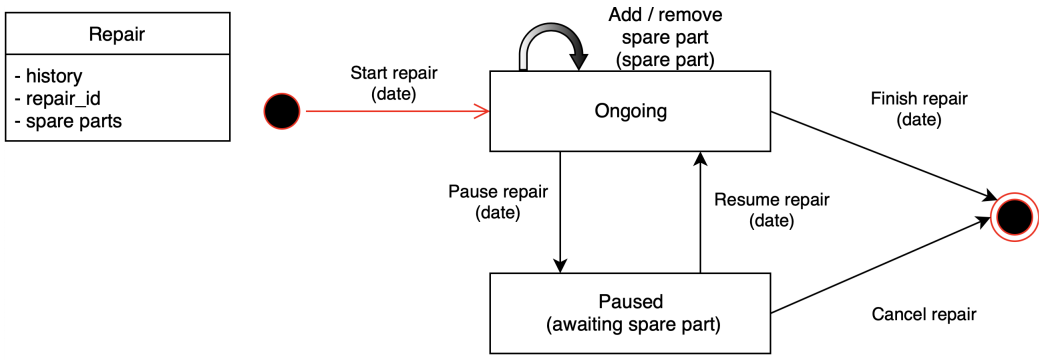


Figure 3.3: Statechart diagram for the class "Repair"

New Spare Part behaviour

In the problem domain we model both new and used spare parts which have somewhat different behaviours. Figure 3.4 shows the behaviour of the class "New spare part". An object of this class begins its life when it is added to an order list. From here it changes state when it is actually ordered from the supplier, and again when it arrives in the workshop and is registered / imported into in the system, after which it is available for use. A new spare part is reserved when it is added to a repair, disappears when the repair is finished, or returns to the available state if the spare part is removed again or the repair is cancelled. Specifically for a new spare part is the need for an attribute containing its SKU number which is used as an identifier when ordering new spare parts from the supplier.

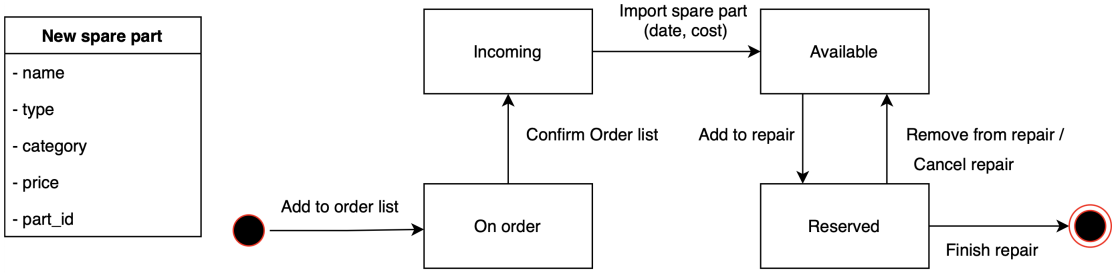


Figure 3.4: Statechart diagram for the class "New spare part"

Used Spare Part behaviour

An object of the class "Used spare part" begins its life when a write-off ticket is created. The part is now marked functional and can become available if the write-off is approved. When in the available state it behaves just like a new spare part. However, in case the write-off is declined the used spare part disappears again. To make it easy for a technician to locate a specific used spare part, the object needs an attribute to remember the location of the written off product which contains the spare part.
Both new and used spare parts also require attributes that specify their type e.g. "screen" or "battery" and which category of product they are compatible with, in addition to their cost.

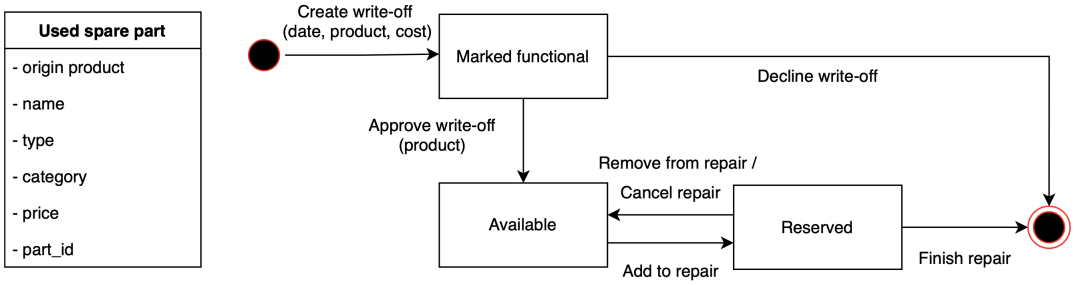


Figure 3.5: Statechart diagram for the class "Used spare part"

4 Application-Domain Analysis

In this part of the report, we focus our attention on analysing the system that will administrate, monitor, and control the problem domain. With the aim of determining the systems usage requirements. The analysis will go into detail with the actors and their uses of the system, leading to a need for supporting these use-cases with functions and interfaces. The actual design of these interfaces is analysed and discussed in Section 6, where we go more in depth with our interface design choices and the principles contained in the DEB course.

4.1 Usage

The system will be developed to support user tasks. Therefore it is fundamental to identify precisely who the users are and their system usage. The two central questions of the target system's usage are: Who will use the system? How will it be used? These two questions can be answered by discussing the system's actors and use-cases. An actor is defined in OOA&D [1, p. 121] as *An abstraction of users or other systems that interact with the target system*. While a use-case is defined as *A pattern for interaction between the system and actors in the application domain*. In this section, we will start with presenting our actor table and after that analyse the process leading to identifying the actors and use-cases outlined in the actor table, lastly evaluating the essential use-cases.
In the following actor table, we have identified eight essential use-cases and two actors that describe the target system.

	Technician	Manager
Register Technician		X
Register Spare Part		X
Register defective Product		X
See Spare Part status	X	X
Write off Product		X
Repair Product	X	
Monitor Workshop		X
Order Spare Parts	X	
Search Product	X	

Table 2: Actor table with actors and their use-cases

A different perspective of the relationship between actors and use-cases can be found in Appendix C.4, where we have made a use-case diagram.

4.1.1 Actors

As discussed in Section 4.1 an actor is defined as *An abstraction of users or other systems that interact with the target system.* To identify the target system, actors, and overall scope, we interviewed one of Blue City's technicians. From the interview in Appendix A it was clear that an integral actor in the target system was the technician. The importance of working closely with the user was highlighted, as a second interview, this time with the manager at Blue City, showed the manager's role as an actor in B. By working closely with different distinct users, we were able to identify the actors present and their role in the target system.

Manager

Goal: To report back statistics to people higher up in Blue City and make final or critical decisions in the workshop.

Characteristics: The manager is in charge of the workshop, with an almost equal level of experience in repairing as the technician. The manager often takes over the role of the technician to help make repairs.

Figure 4.1: Actor specification for "Manager"

Technician

Goal: To repair defective products allowing them to be sold by the store. Their basic need is to have the tools and spare parts necessary to carry out the repairs.

Characteristic: The systems primary users will be technicians that have a high need for information. Technicians will use the system to find the most important products to repair.

Figure 4.2: Actor specification for "Technician"

4.1.2 Use-Cases

The purpose of the application domain analysis is to define how the users will interact with the system. Use-cases help to describe the expected interaction between an actor and a particular part of the system. The complete set of use-cases determines all system uses within the application domain. Use-cases will assist in determining and developing the application domain. Furthermore, as use-cases describe the user's interaction with the system, establishing and analysing use-cases will let us better understand the user's interaction within the application domain, leading to a better system.

Determining use-cases

Determining use-cases is both a creative and analytical process. Use-cases describe the interaction between the user and the system. Therefore, determining use-cases requires an evaluation of the users and their work tasks. In our target system, the users are the following actors; technician and manager. An evaluation of the work tasks is derived from analysing the initial surveying/interviewing of the actors about their work tasks. After this information is analysed and organised, it is condensed to be listed in Section 5 as the requirements for a system supporting those actors' work tasks. Through this process, we determined a list of initial potential use-cases in Table 3:

Register Technician
View Shop Statistics
Register New Product Type
Add Product to Inventory
Examine Product
Write-off Product
Repair Product
Select Product for Repair
Search for Spare Part
Add Spare Part to Product
Add Repair Description
Register New Spare Part Type
Add Spare Part to Order List
Confirm and Clear Order List

Table 3: Initial use-case list

While many potential use-cases could be determined, through further analysing surveying/interviewing with Blue City we determined the most essential use-cases to our target system as the following:

- Register technician (Figure 4.3)
- Repair product (Figure 4.4)
- Write-off product (Figure 4.5)
- Monitor workshop (Appendix C.2)
- Register defective product (Appendix C.3)
- Register new spare-part type (Appendix C.11)
- Spare-part status (Appendix C.12)

Use-case statechart diagrams

In the following section, the three most essential use-cases will be discussed and analysed. While the other use-cases listed in Section 4.1.2 are important to the target system, the following use-cases are those we deem to be most essential to our product.

- Register new technician
- Repair product
- Write off product

Analysing these three use-cases will give an insight into the workings of the most essential parts of the target system.

Statechart for register technician use-case

Use-case: Register new technician can be accessed through an admin web page and can therefore only be accessed by the manager. The manager will then enter the credentials (name and password) of the new technician. When credentials have been entered, a validation will happen to see if the technician is already registered. If there is an error with registering a technician, the manager will have to register the technician again. If the credentials are verified and no error happens the technician is registered.

Objects: Manager, Technician

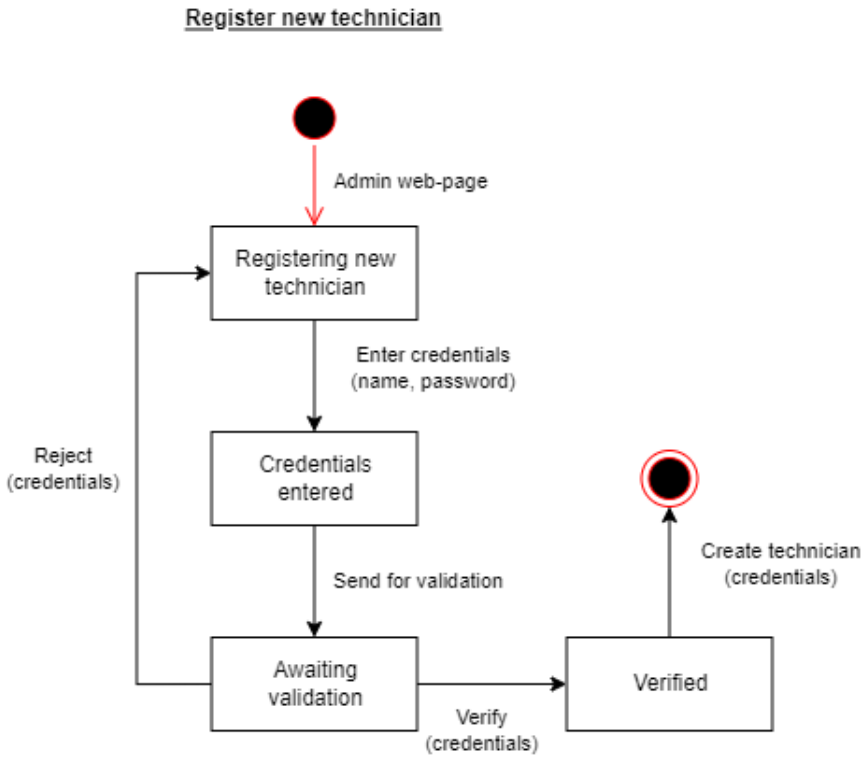


Figure 4.3: Register technician use-case

Statechart for repairing product use-case

Use-case: The process of repairing a product is initiated by selecting a product and starting a repair on it. The technician will then search for the spare parts required to repair the product. If no compatible spare parts are found the technician will put the missing spare parts on the order list, add a repair description and pause the repair.
If the spare parts are found in the system, the technician will retrieve them from the written off product they are stored in and add them to the product to be repaired. Then the technician will check that the spare parts work, and if so, add a repair description to the product, otherwise the spare part will be removed and the technician will search for other functional spare parts.
A repair process can be paused and resumed whenever necessary, i.e. if the technician needs to go on a break or does not have the required spare parts to finish the repair.

Objects: Product, Spare Part, Repair, Technician

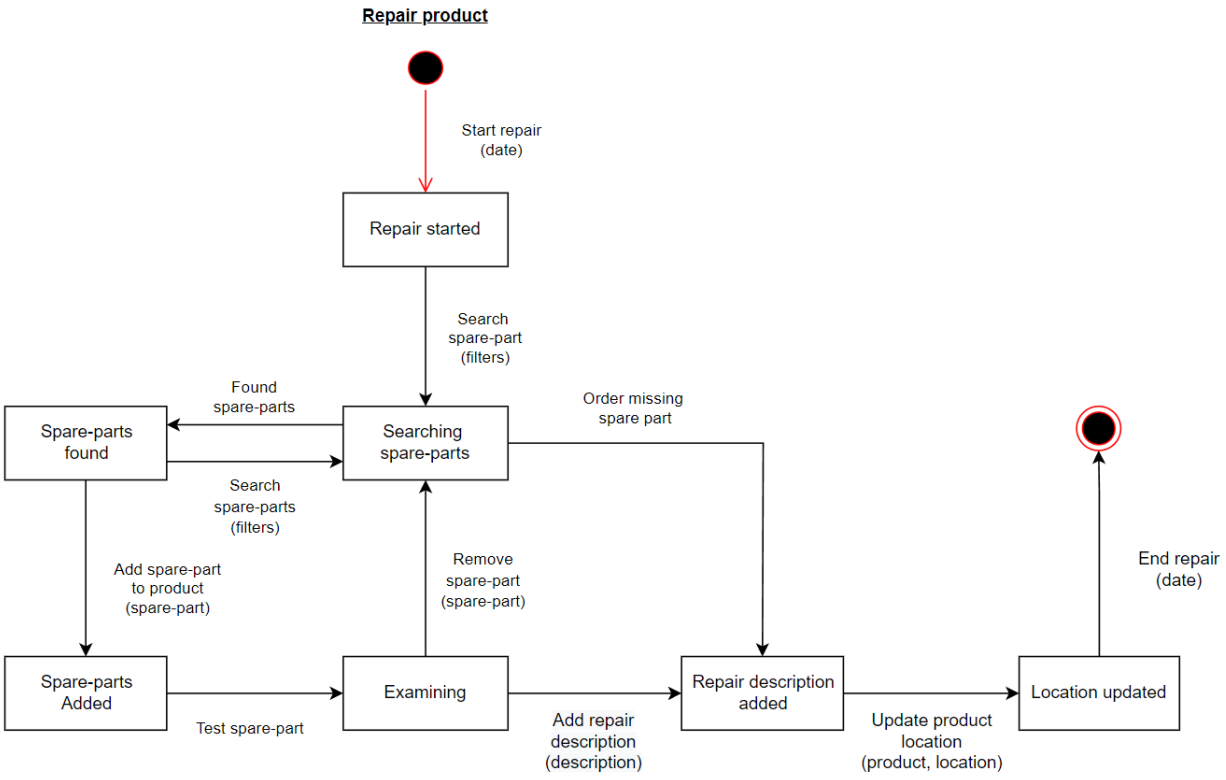


Figure 4.4: Repair Product use-case

Statechart for writing off product use-case

Use-case: write-off can be accessed when a product is sent to loss, and can be accessed both by the manager and the technicians. Writing off a product is initiated by choosing a specific product. When writing off, the functional spare parts will be marked, with the option of unmarking functional parts. Then a write-off ticket is created and the product will be marked as in write-off. The ticket will then wait for manager approval before the product is written off.

Objects: Manager, Technician, Product, Write-off Ticket, Spare Part

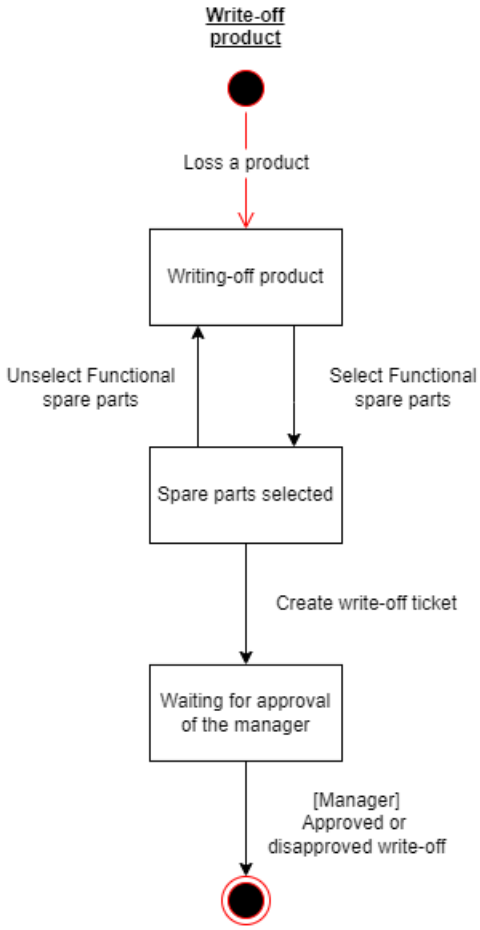


Figure 4.5: write-off product use-case

Evaluating use-cases

In Section 4.1.2 we determined that many use-cases exist in our application domain. However, evaluating the use-cases can lead to both a better understanding of the users' needs and a better overall system.

A truly critical evaluation of the use-cases will have to include the users. To truly involve the user, we will present our use-cases as prototypes and examine how the users interact with the them. In our system's case the users would be the technician and manager.

Examining how these actors interact with the prototypes will allow us to evaluate our use-cases and improve upon them. In order to further evaluate the use-cases, it is also important to examine Blue City's current computerised system and organisation. As our system is constructed to replace Blue City's already established Priotool, it should be evaluated how the target system will affect Blue City's current organisation and way of working.
In one of the interviews, we presented our use-cases as prototypes to the Blue City manager. However, since we presented the manager with prototypes representing use-cases for both technicians and managers it could have made the technician use-case feedback partially inaccurate. The prototypes were made with the prototyping tool Figma [2] and can be seen in Section 6.4. The manager was guided through the prototypes, and how the actors would interact with the system was explained. We examined the manager's interaction and took notes of his feedback on the prototypes. The manager mainly gave positive feedback, easily being able to follow the prototype's "simulated system" interaction and workflow. The only issues with the prototypes were usability issues. These usability issues were primarily on the user-interface design, but were also noted and considered. The manager mentioned that the prototypes illustrated the interaction between the system and users quite well. The use-cases were revisited following the interview and minimal changes were made.
The interviews we had with Blue City provided us insight that their current system lacks a lot of functionality. These functionalities included monitoring repairs, performing write-offs, and ordering spare parts.

Our system aims to replace their current system, as well as providing additional support for:

- Ordering spare parts
- Registering new technicians
- Selecting products to repair
- Writing off products

As we are aiming to replace one system with another, our system will not have any major organisational changes, while providing improved functionality.
However, the potentially biggest impact of our target system is the lack of full integration with the "Blue-system". Blue City's current Priotool is directly integrated into the Blue-system and would therefore automatically list defective products. As the target system is a stand-alone system and does not have this integration, any before automated information from the Blue-system will therefore have to be manually entered into the target system. This will change the way that Blue City actors work as this "information entering" will have to be an additional work function.

4.2 Functions

This chapter is dedicated to the functions that make the system functional. The purpose is to figure out *how* the system is going to support the different use-cases.

4.2.1 Identifying functions

Our strategy for identifying functions is to start systematically from use-cases and extract the necessary functions.Functions can also be determined by going back to the system definition to give us an abstract version of interpreting our solution. The following types are update, signal, read and compute functions.

The function list contains the types of each function. Each function represents how an action from the user in a use-case is supported through a functionality in the system. The context and characteristics of each function is what the system will do when executing it. Below is the full list of functions from each category of use-cases.

4.2.2 Function List

Repairing:	**Complexity**	**Type**
Start repair	Complex	Update
Cancel repair	Complex	Update
Finish repair	Complex	Update
Pause repair	Medium	Update
Resume repair	Medium	Update
Add spare part to repair	Medium	Update
Remove spare part from repair	Medium	Update
Update product cost	Simple	Update
Get repair list	Simple	Read
Get single repair	Simple	Read

Table 4: Function list for use-case "Repairing"

Writing off:	**Complexity**	**Type**
Create write-off ticket	Complex	Update
Approve write-off ticket	Medium	Update
Decline write-off ticket	Medium	Update
Get spare part per category	Medium	Read
Get write-off ticket list	Simple	Read
Get single write-off ticket	Simple	Read

Table 5: Function list for use-case "Writing off"

Register technician:	Complexity	Type
Verify credentials	Medium	Compute
Create user	Medium	Update
Delete user	Simple	Update
Update user password	Simple	Update
Update user privilege	Simple	Update

Table 6: Function list for use-case "Register technician"

Describing complex functions

The system development book states in chapter 7.3 that complex functions should be described sufficiently to understand their complexity [1, p. 146]. In total we identified four complex functions that we will briefly describe.
Complex functions are functions interacting with multiple objects, making these more advanced to design while ensuring correctness and reliability. Thus it is important to describe them properly before beginning development.

- *Start repair*: This function will be responsible for creating a new repair object and associating the technician and the product to it. The product undergoing the repair will need to change state to "being repaired". It will also require some validation of the product so that only defective products can be initialised for a repair.
- *Cancel repair*: This function will need to revert all changes that were done during a repair. To do this, all spare parts added during repair need to be unreserved, the product should go back to being defective and the repair object will need to be deleted.
- *Finish repair*: A lot of things need to happen when a repair is finished. For all the spare parts that are added, we take the total cost and add to the product. Afterwards the spare parts need to be deleted, the product needs to change state to "Repaired" and the repair change state to "Finished".
- *Create write-off ticket*: When instantiating a write-off ticket, the product undergoing the write-off will have to be "Defective". For each spare part that is marked functional, we create a spare part instance in a marked state and append it to the write-off object. To the ticket we append the reason and the user who instantiated the write-off.

4.2.3 Evaluating function complexity/type

We must evaluate whether our function list is consistent with our system's requirements. The way we did this was by comparing it to the system definition and making sure that all of the responsibilities of the system is covered by our functions.

5 Requirements

In order to divide the requirements that we have gathered in the analysis, we will be using the MoSCoW model [3] to sort critical requirements from nice to have requirements. The prioritisation of requirements are based upon interviews we have done and our general understanding of what our users mostly need. In this section we will only focus on the functional requirements as the non-functional or qualities of the system is described in section 7.1.

5.1 Functional Requirements

Functional requirements describe what the system should do and are derived from the interviews with Blue City, the feedback on our prototypes and use cases, as well as the system definition. As such they describe the specific functionalities that the system must make available to the user. They are testable with unit tests unlike non-functional requirements that are tested by e.g. a usability test.

Managers can:

- register and remove technicians with name or initials
- add new products and new spare parts to the system
- confirm or reject write-off ticket for products

Technicians can:

- search for a product by product number
- find products in system by product type
- start a repair on a product
- pause, resume and finish repairs
- create a write-off ticket for a product
- add spare parts to a product during repairs
- search for spare parts by type and product compatibility
- add spare parts to an order list common to the workshop
- clear the order list

The system must:

- update product cost price after repair based on spare part costs
- store information about all products, repairs and spare parts in a database
- present to the user lists of all products, repairs and spare parts in the database
- present the manager with statistics about repairs

5.2 Prioritising requirements using MoSCoW

Items described in "**Must have**" are the requirements that must be met before this project can be considered usable: The Minimal Viable Product (MVP).
Items described in "**Should have**" are the things that the product needs before this project can be considered a desirable alternative to Blue City's current system.
Items described in "**Could have**" are the things this product needs in order for it to feel polished.
Items described in "**Won't have**" are the things this product does not need, but if there is time would put the cherry on top .

Must have

- System must recommend which product is most profitable to repair
- System must be capable of handling more than one user at a time
- User must have a way of registering a product as undergoing repair
- User must be able to pause, resume, cancel and finish a repair
- System must keep track of time spent on a repair
- User must have a way to add spare parts to a product under repair
- User must have a way to remove spare parts from a product under repair
- User must be able to write-off defective products
- System must be able to register functional spare parts within written off products
- The cost a written off product must be split between the spare parts marked functional
- Used spare parts must be able to be traced back to the product it came from
- The total cost of spare parts used during a repair must be added to the product cost
- Users must be able to find products, repairs and spare parts
- System must provide statistics about repairs

Should have

- Allow searching for products and spare-parts in the system
- Provide an option to sort items in the system
- Give fundamental information about the product and its issues for each defective product
- Allow managers to register and remove technicians
- Provide managers with an overview of all current users
- Allow managers to reject and confirm write offs
- Search for a product by product number
- Search for spare parts by type and product compatibility
- Add spare parts to an order list common to the workshop
- Clear the order list
- Technicians can find products in the system by product type

Could have

- Help managers see visually which technicians are the most productive.
- Help managers visualise different sales statistics such as best selling products.
- Give detailed information on the supply-chain & life cycle of spare-parts.
- Allow a recount of items in the shop using the system.
- Separate instance of the system for each blue city store
- Create perfect/semantic security in the system
- Encrypt sensitive data in the system

Won't have

- Use block chain to secure the supplychain / lifecycle of different products and spare-parts
- Optimise the user interface for mobile usage
- Allow a complete overview and comparison of different Blue City store locations performance
- Be able to scan product information into the system using a physical scanner

6 User Interface Design

Designing user interfaces requires an understanding of the problem. After examining the result from the interview in 2.1, we now have an understanding of our users' problem. This chapter will cover the theory that went into the design as well as concrete examples of how it was used.
The aim of Design and Evaluation of User Interfaces (DEB) course is to develop a high-quality interactive user interface that complements the target system and users. With focus on evaluating the user interface by primarily using prototypes and sketches of the user interface. In the following section we will discuss the process of designing and evaluating the user interface for the target system.
The conceptual design goal is to understand the application and make it more understandable to all people who have no experience with it. We approach designing prototype solutions and testing the prototypes with our users to gain more knowledge and make the final product better suited for our users'.

6.1 User interface design guidelines

When designing user interfaces or designing anything in general, its always a good idea to follow principles and guidelines that have been tested and proven to work. Nielsen and Molics design principles on designing user-interfaces are well known in UI-design and are applicable when making design decisions [4]. Nielsen and Molic have outlined 10 golden rules to follow, these include:

- *Visibility of system status*
- *Match between system and the real world*
- *User control and freedom*
- *Consistency and standards*
- *Error prevention*
- *Recognition rather than recall*
- *Flexibility and efficiency of use*
- *Aesthetic and minimalist design*
- *Help users recognize, diagnose and recover from errors*
- *Help and documentation*

Throughout the chapter, we will refer to these design principles when reasoning about the design.

6.1.1 Memory

When talking about memory in the context of design, we generally talk about the working memory and memory load. Working memory for our design is about how much strain we put on our users to remember aspects of the system. To reduce the amount of working memory required of our users we use technique's such as hiding details and chunking information together. Nobody likes to see unnecessary details, so information should be kept at a minimal. Detailed information can easily be hidden behind drop boxes or separate detail pages.
As an example we design the product page list to only contain head level information, then users can choose to click on a drop-down arrow to show more product detail.
When we decide how many item menus goes into each menu, we again think about memory. George Miller (1956) [5, p. 304] describes that the short-term memory has a limit of 7+-2 when chunking information. This also applies when picking the number of elements in drop-menus and navigation bars as there should be no more than this limit.

6.1.2 Attention

Designing for attention is about highlighting and focus of certain elements. Different measures such as shape, color and brightness affect our users attention. One way we have worked with attention is through layering of different design components. The most desirable features or elements is placed on the top, while more detailed information or features is located in the bottom of the page. Therefore it is very important to think of what the elements located on the top most layer will present of information and also how this information is presented to grab the most attention. Having a layered approach improves the scan-ability of information. As an example, the approach can be seen on figure 6.1. The top level shows the percentages of different categories of products. This was requested specifically by the technician as it was one good thing about their old system. We designed it to have very vibrant colors and big boxes to highlight and grab attention. The next layer was designed with the same font size as the layer above but with less colors to make it less attractive but still important non the less. The third layer that includes detail on products was designed specifically not to be highlighted by having a small font size and a less vibrant grey color.

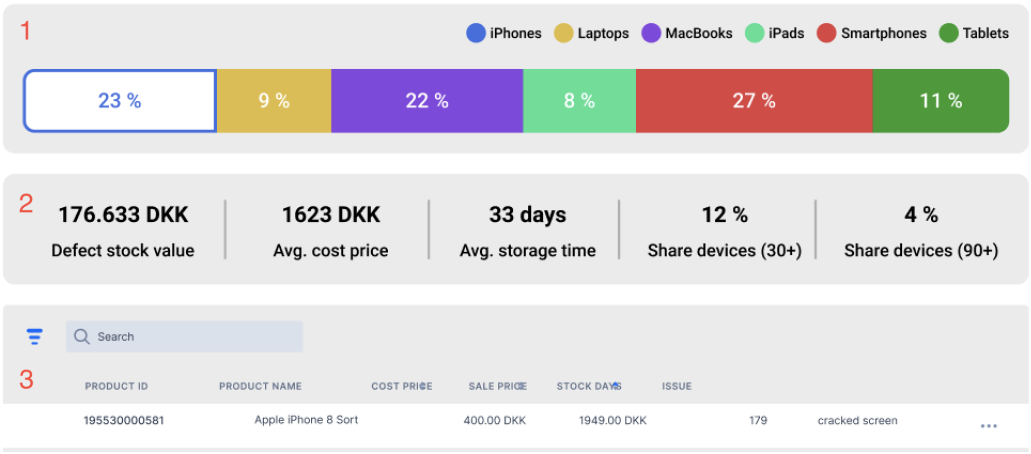


Figure 6.1: Mockup of overview page

6.1.3 The Gestalt laws of perception

A law with several principles that can be useful is the Gestalt law of perception that refers to people's perception when seeing user interfaces. In other words, this law deals with how people is mentally affected when visiting a website or program. Using these principles in our project will make it easier to create a pleasing user interface. The Gestalt law has three principles: proximity, similarity, and continuity that will affect the design process of the prototypes.

Using proximity to organise buttons

The first principle is about **proximity**, which deals with positioning of objects and how it effects users. As such, objects that are positioned closely together are perceived to be related. The complication for positioning is to consider the spacing of each element. Elements that are conceptually related can be grouped by positioning them at the same height and same spacing around them. As an example we designed the repair page with action elements such as 'pause' and 'resume'. These are conceptually related and therefore we can either choose to place their respective buttons next to each other with different colours or place them in the same height but not next to each other with the same colours. Likewise, related action elements like 'approve' and 'disapprove

for write-off' should be placed next to each other with different colours. An illustration of how we worked with proximity can be found at Section 6.4.

Using similarity to organise elements

The second principle is about **similarity**, where features such as color, orientation or size are essential techniques to give a standard design for the interface [6]. In the case of organising different categories of products, we chose to have a single color scheme that would be consistent throughout the design. This way, users will quickly be able to relate a single category name with a color.

6.1.4 Human Error

Human error is a common occurrence that can happen when using web applications. When designing for human errors, we need to think about the errors future users can make, and what they can do to recover from these errors. The important part of designing our solution is to have as small an amount of error occurrences as possible, which should be taken into account, so users do not find it difficult to use the web application and they should be able to recover from their actions. For instance, we need to deal with the situation in which a user starts a repair by accident. To recover from this, an undo mechanism needs to be in place so that the repair can be cancelled. By the nature of the repair workshop, errors are very frequent among technicians and managers as well. Therefore, we need to be able to account for these human errors where possible.
Human error also ties very well with the design principle of showing system status. System status should be read very clearly by the user to be able to help recognise and recover from the error. As an example where we thought in system status is when the user creates a write-off. The user gets notified of his action and can feel certain that the action he expected matches that of the system status.

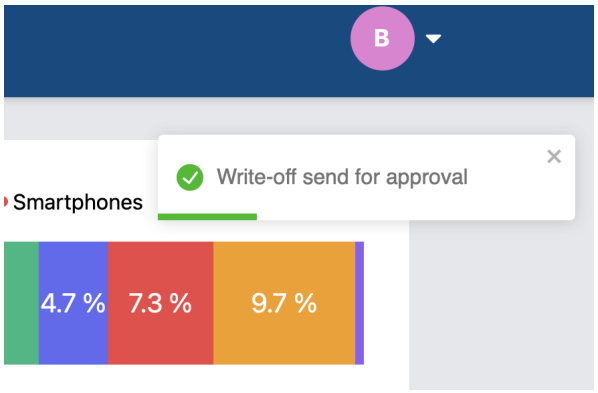


Figure 6.2: System status after successful creation of write-off ticket

6.2 Usability

Usability is vital for any system that deal with interaction of users. Usability is defined as: "*The levels to which a design can be used by a certain number of users to attain the set objectives with efficiency, effectiveness and high level satisfaction in a particular context*" [7].
What characterises a system with a high degree of usability can differ from both the user and system. There are different approaches to achieving a system with high degree of usability. The DEB book usability chapter specifies four possible approaches to achieving high usability [5, p. 108-111], shown as follows:

- Gould and Lewis design principles
- Value system design
- Dan Norman design characterisations
- Vermeulen design techniques

These four approaches each have their pros and cons. We have chosen to asses our systems degree of usability through the "Gould and Lewis design" approach as it seemed the approach that focused the most on user involvement and iterative development. The "Gould and Lewis design" approach focused on the following principles to achieve a high degree of usability:

1. Early focus on users and tasks
2. Empirical measurement
3. Iterative design

We found that usability was an essential part of our system and should be highly prioritised as our system success is highly based upon having a high degree of usability for our users. Therefore, we followed the "Gould and Lewis design" first design principle by emphasising surveying and interviewing with Blue City, frequently discussing our ideas and progress with Blue City.

This also fit with the second design principle in which we presented sketches and prototypes on parts of the system to gauge their reactions. We then use the feedback to improve the usability of our system. This process was iterative as we followed a cycle of design, test and measure, and redesign. This allowed us to achieve a higher degree of usability.

6.3 Acceptability

Acceptability is about fitting technologies and services into peoples lives. Acceptability can only be understood in terms of use, unlike usability. The acceptability of a technology can be measured by two perspectives: The ease of use and effectiveness.
Acceptability of a technology is a broad concept and can differ wildly depending on changeable features. The key features of acceptability are as follows:

- Political
- Convenience
- Cultural and social habits
- Usefulness
- Economic

As seen, it can be hard to measure acceptability as some features are hard to define. Something like if a technology is "politically acceptable" or the "usefulness" of a technology is hard to measure. However, it is vital to keep these acceptability features in mind as a successful system should be easy to use and effective.

In our target system, the most important features are the "Usefulness" and the "Convenience" of our system. We choose not think of the Economic feature as the system does not have any budget or need to be sold to Blue City.

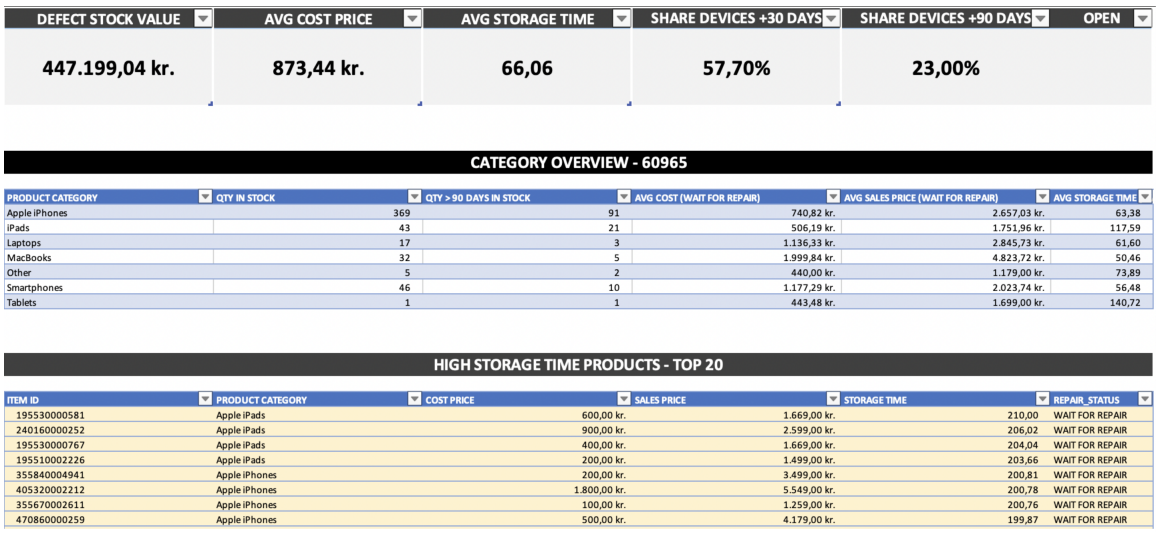
"Usefulness" is crucial as the target system is competing with Blue Cities own system. It is not enough that our system is usable as it is not the only choice for Blue City and if the target system fails to be less useful then Blue Cities current system then our system will not be used. "Convenience" is essential as our system is meant to provide a service and therefore must be flexible and simple to use. Our system does not create anything new but instead supports the already established Blue City system. If our system does not prove to be convenient then there will be no reason to use it.

6.4 Illustrating the design of the application

In this section, we focus on the development of the design. Each subsequent chapter will go through the sketches followed by a more refined wireframe of each interface. Every interface is based on system functionality explored in the application domain analysis.

Overview page

The overview page serves as the general way for our users to find products that need repair, while also showing workshop statistics. The top portion of the overview is reserved for statistics while the bottom half is for searching and prioritisation of the products. The idea is that, when the user clicks on a product category the whole overview page changes to fit that category. The products in the product section can be sorted through multiple parameters at once. This was done, so that the users of the system could chose how they want to prioritise and not us forcing them to pick any specific product. The overview page was inspired by their current system overview to include relevant metrics for our own overview. Figure 6.3 shows the overview of their current system.



DEFECT STOCK VALUE	AVG COST PRICE	AVG STORAGE TIME	SHARE DEVICES +30 DAYS	SHARE DEVICES +90 DAYS	OPEN
447.199,04 kr.	873,44 kr.	66,06	57,70%	23,00%	

CATEGORY OVERVIEW - 60965

PRODUCT CATEGORY	QTY IN STOCK	QTY > 90 DAYS IN STOCK	AVG COST (WAIT FOR REPAIR)	AVG SALES PRICE (WAIT FOR REPAIR)	AVG STORAGE TIME
Apple iPhones	369	91	740,82 kr.	2.657,03 kr.	63,38
iPads	43	21	506,19 kr.	1.751,96 kr.	117,59
Laptops	17	3	1.136,33 kr.	2.845,73 kr.	61,60
MacBooks	32	5	1.999,84 kr.	4.823,72 kr.	50,46
Other	5	2	440,00 kr.	1.179,00 kr.	73,89
Smartphones	46	10	1.177,29 kr.	2.023,74 kr.	56,48
Tablets	1	1	443,48 kr.	1.699,00 kr.	140,72

HIGH STORAGE TIME PRODUCTS - TOP 20

ITEM ID	PRODUCT CATEGORY	COST PRICE	SALES PRICE	STORAGE TIME	REPAIR_STATUS
195530000581	Apple iPads	600,00 kr.	1.669,00 kr.	210,00	WAIT FOR REPAIR
240160000252	Apple iPads	900,00 kr.	2.599,00 kr.	206,02	WAIT FOR REPAIR
195530000767	Apple iPads	400,00 kr.	1.669,00 kr.	204,04	WAIT FOR REPAIR
195510002226	Apple iPads	200,00 kr.	1.499,00 kr.	203,66	WAIT FOR REPAIR
355840004941	Apple iPhones	200,00 kr.	3.499,00 kr.	200,81	WAIT FOR REPAIR
405320002212	Apple iPhones	1.800,00 kr.	5.549,00 kr.	200,78	WAIT FOR REPAIR
355670002611	Apple iPhones	100,00 kr.	1.259,00 kr.	200,76	WAIT FOR REPAIR
470860000259	Apple iPhones	500,00 kr.	4.179,00 kr.	199,87	WAIT FOR REPAIR

Figure 6.3: Overview page of their current system

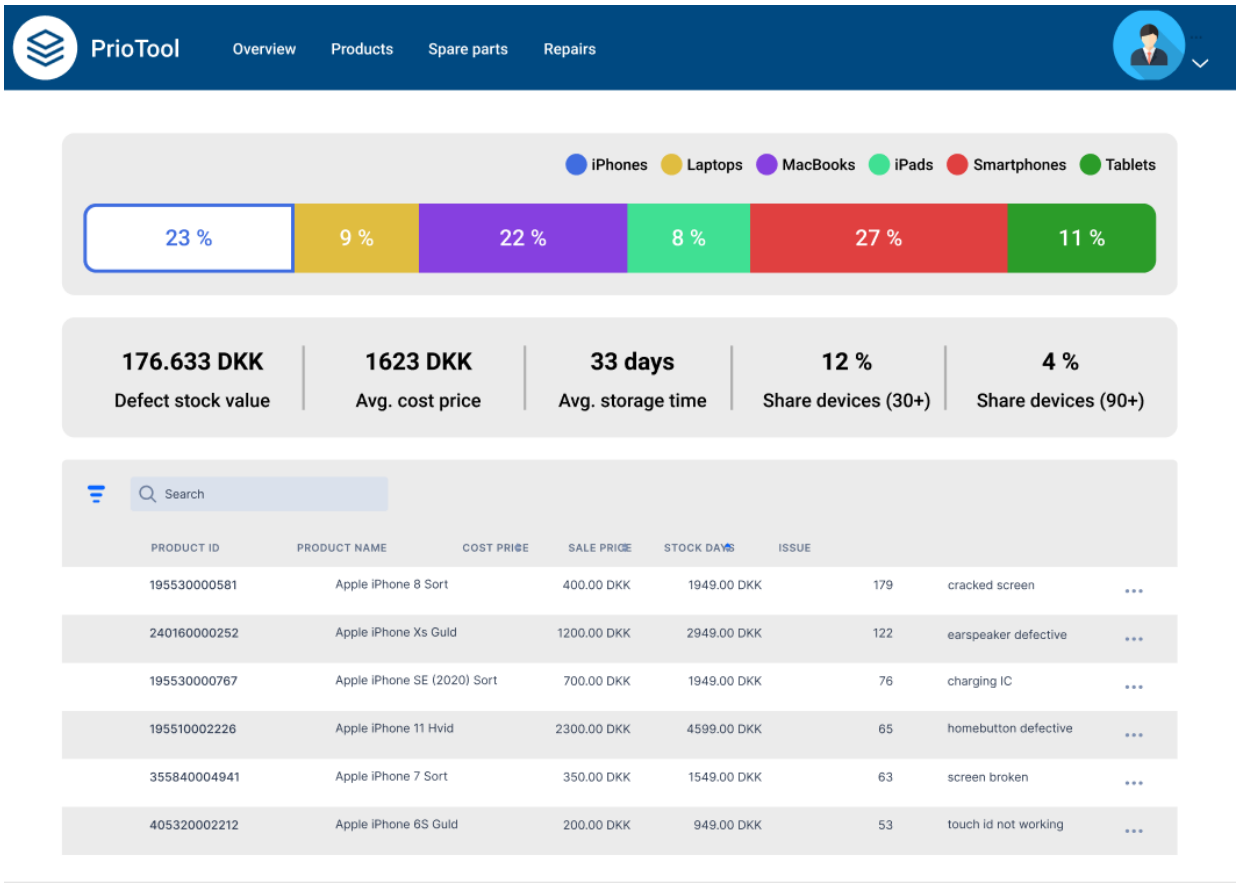


Figure 6.4: Wireframe of the overview page

Repair overview page

The repair page is responsible for everything related to repairs. From here users should be able to see a list of all the repairs in a quick overview. The user can delve deeper into a single repair by going into the respective repair detail page. Every repair on the list have some quick details to identify a single repair.

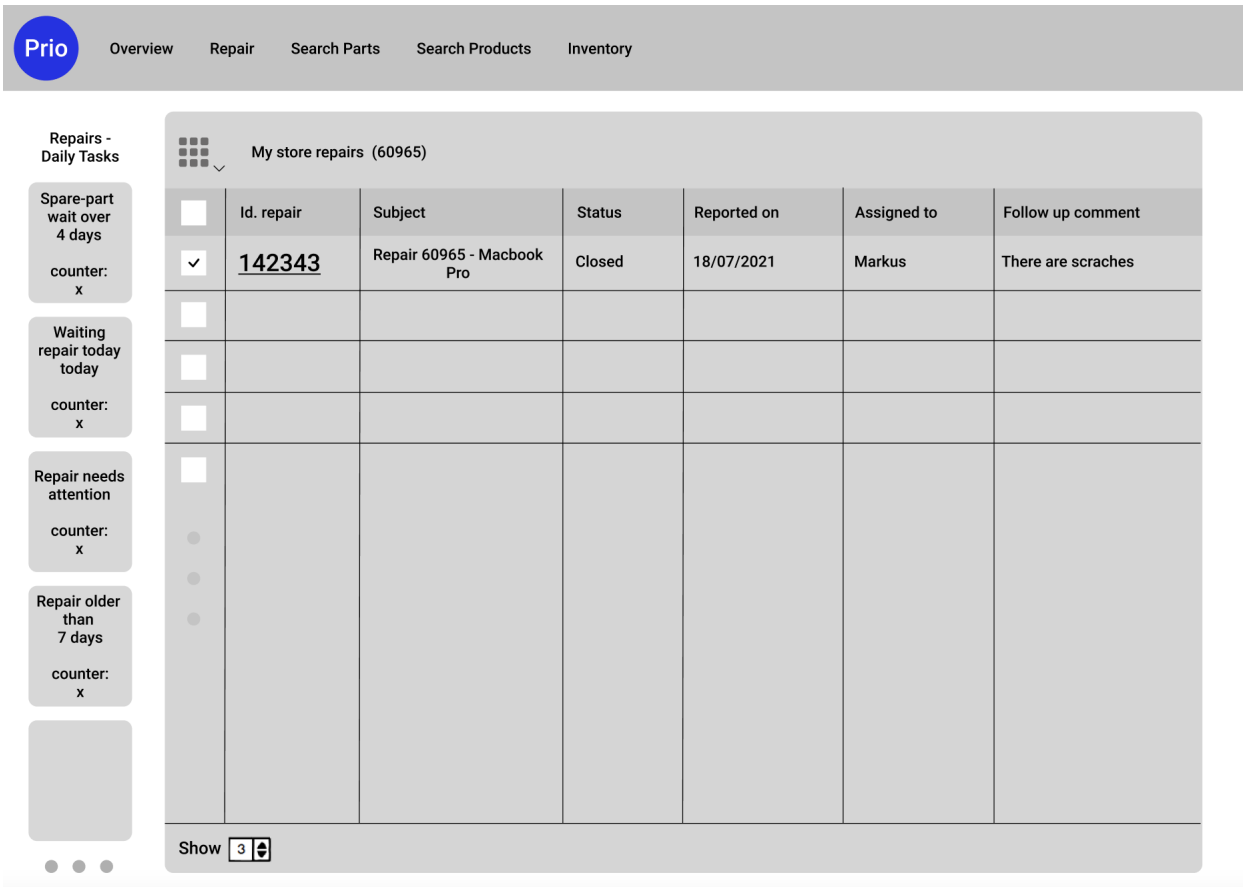


Figure 6.5: Wireframe of repair overview page

Repair detail page

When sketching the repair detail page, we needed to design the part that support the different actions that can be performed on a single repair. We chose to split up the detail page in three different sections. A detail section on the specific repair, including the repair method, the status of the repair and the id of the repair. A defective comment section holds information about the issue of the product undergoing the repair. At last a section will have detail on the product that is being repaired. On the bottom of the page the user can either save or cancel the repair.

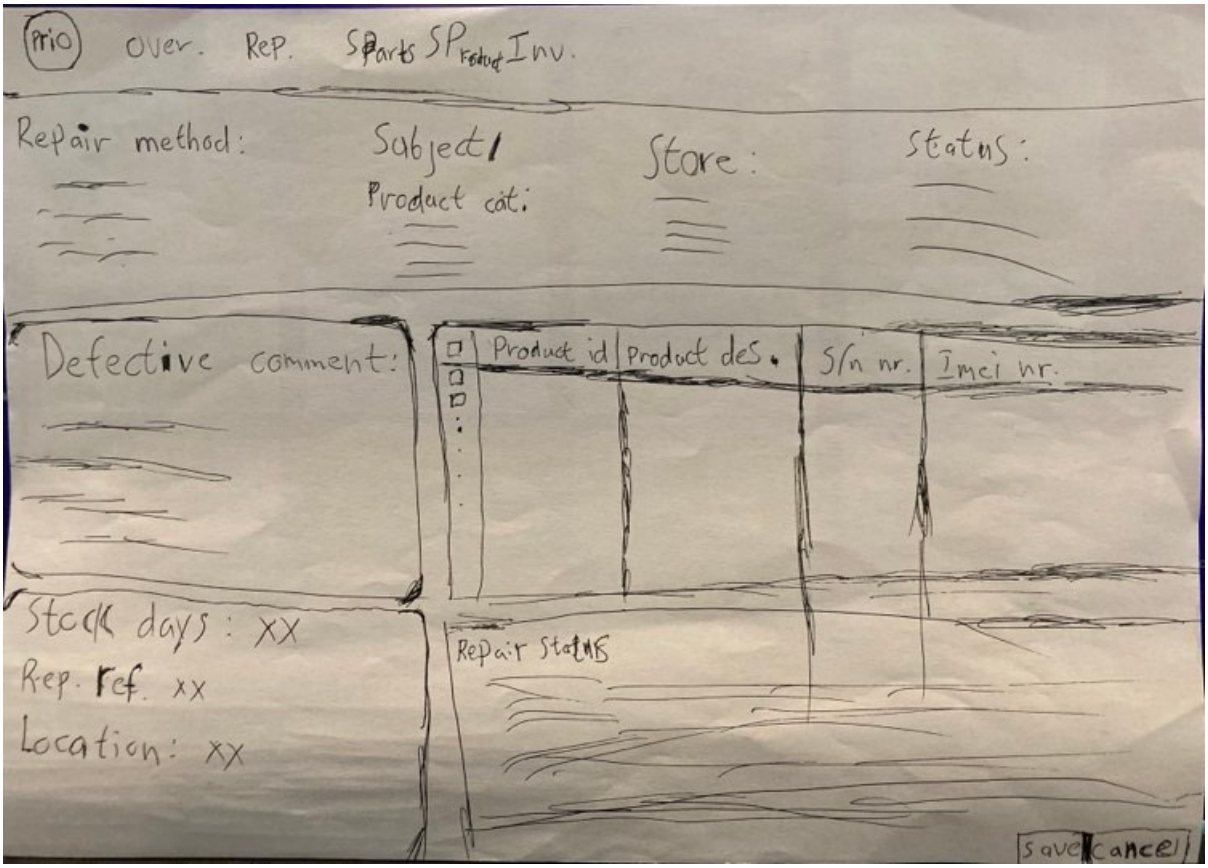


Figure 6.6: Sketching repair detail page

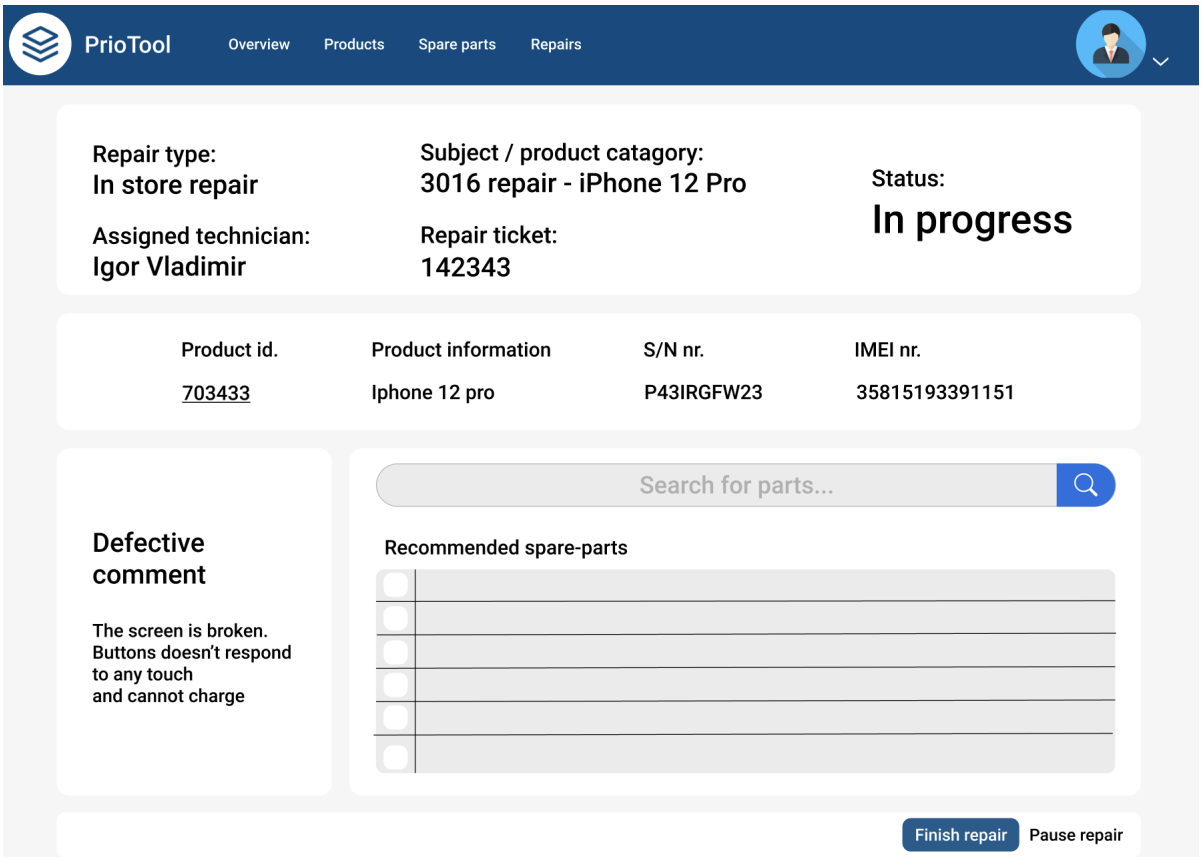


Figure 6.7: Wireframe of repair detail page

Figure 6.7 shows the further developed wireframe of the before sketch. The overall design of the page got a bit of an overhaul. We realised that we also had to support the functionality of adding a spare part to the repair. This is done by having a spare part section in where the user can search for a spare part and get spare parts that are recommended during the repair.

Product page

When designing the product page we considered how we could keep the information at a minimal. The page should include a list of all the products they have in the workshop and some methods to filter through these. We decided to have a single search bar for doing a general search. A drop-down menu of the different categories would filter the products by category and a filter section would hold the different models belonging to the chosen category.

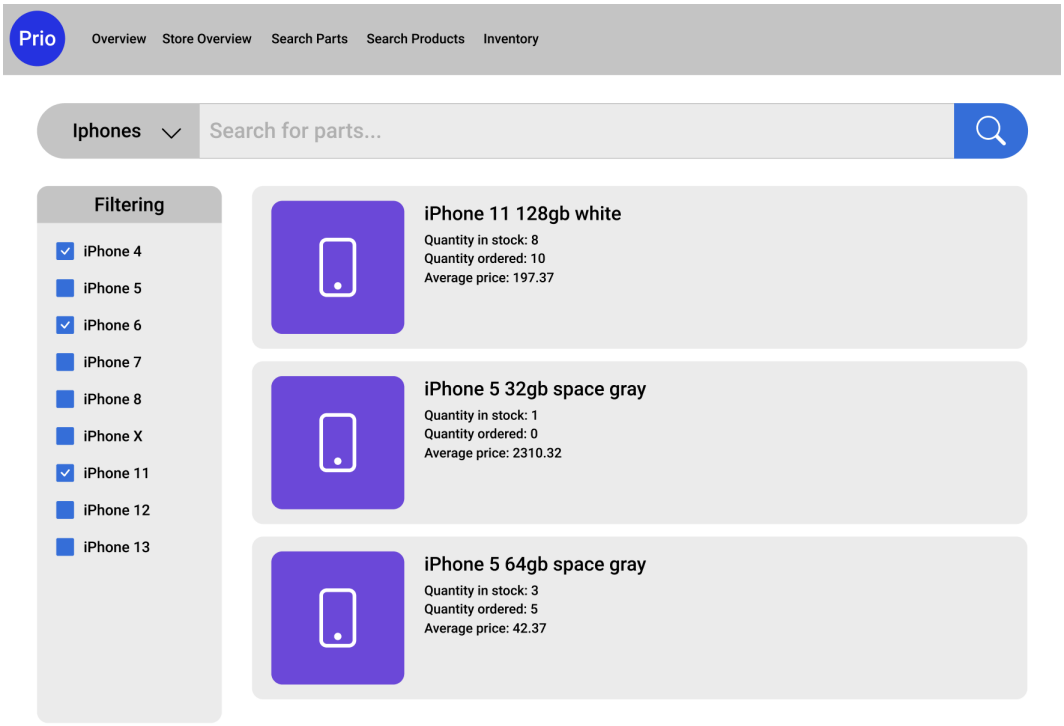


Figure 6.8: Sketch of product page

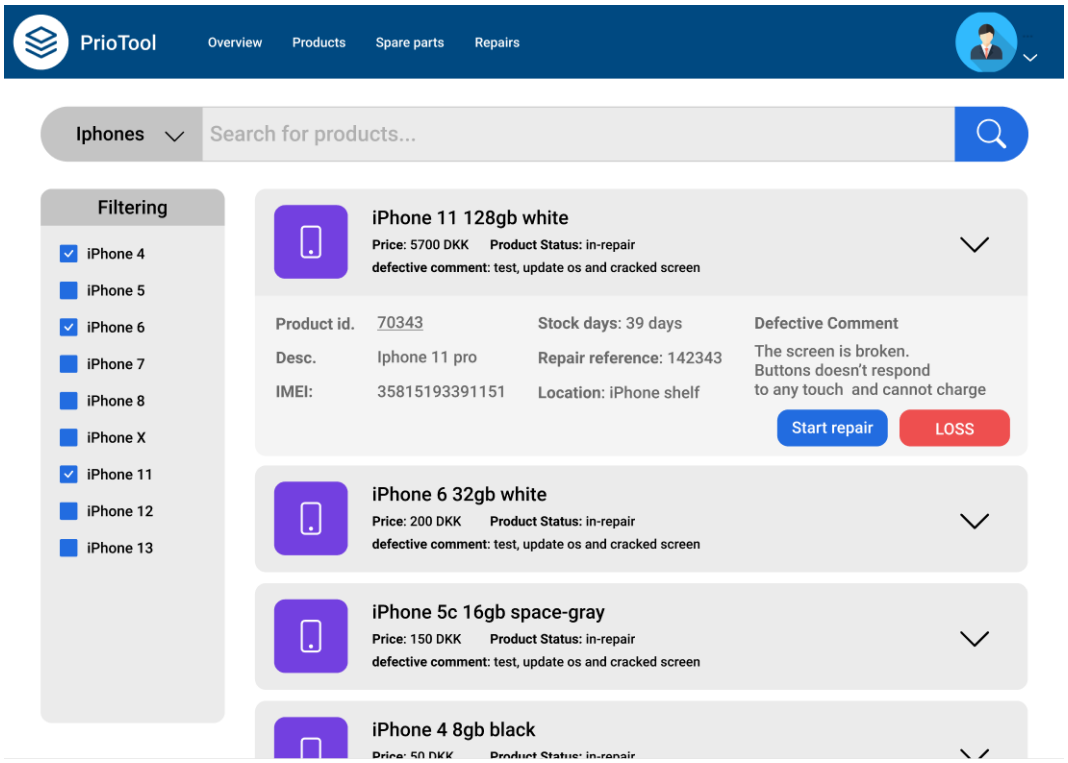


Figure 6.9: Wireframe of product page

With a bit more development on the product page sketch, we came up with the following wireframe. The individual products on the product list have gotten more detail and now includes a way to start a repair and loss the product which was a missing feature. We chose to hide the details on the product by having a collapsed detail view when clicking a little arrow icon. In the detailed collapsed view, the user sees how many days it have been in stock, where the product is located, the full defective comment and either a product id or IMEI number to identify the specific product.

7 Architectural Design

This section will cover the definition and prioritisation of a criteria collection along with how our individual components are interconnected and the process of laying the system's structure.

7.1 Criteria

The purpose of the criteria activity in Object-Oriented Analysis & Design is to set the design priorities, but it is not necessarily the way of achieving quality assurance in our application. Considering what must be prioritised for a good design can have the effect of not having major weaknesses. Below is the criteria table where we have argued and summarised from a scale: very important to easily fulfilled.

Each criterion for our system, are described in accordance to *Classical criteria for software quality* as seen in Figure 9.1 [1, p. 180].
Usable: Since the software is going to be designed around the current physical workflows, which the technicians employ at the workshop, usability is a very important criterion for the architecture. It is important the software solution does not radically change the technicians' organisational systems, and therefore should be quite adaptable to the organisational context. Since the software is to be a replacement for Blue City's spreadsheet-based solution, our solution will not be constrained by any existing technical contexts.

Secure: The software solution we are going to provide is not going to be fully integrated with the company's internal systems, but more of a standalone solution specific to the company's repair workshops. Therefore, since our solution does not interface directly with company sensitive information, security is not as vital for the system as other criteria.

Efficient: In terms of efficiency, the solution that we will provide to Blue City far exceeds the efficiency of the currently implemented system. Since the current system does not allow multiple users to perform actions in the system at the same time, this restriction for the users decreases their task execution speed. Thus, ensuring that multi-user operability is available in the new system will make it far more efficient than the current solution.

Correct: In order to complete their work efficiently, the technicians at Blue City have come to rely on some functionalities of their current solution. These functionalities become requirements for the new system and must be fulfilled for the system to become beneficial for the technicians.

Reliable: As the primary goal of the software solution is to help the technicians keep track of defective products and spare parts, the time it takes to repair a product, and correctly add spare part prices to the price of the products, it is paramount that the system precisely and reliably executes the above-mentioned, so the system does not desynchronise with the physical domain.

Maintainable: In regards to the limitations of this project, primarily the time constraint, maintainability of the system will not be the most relevant criteria for this project.

Testable: Considering test-driven development being a major part of object-oriented programming and doing so ensures that the system performs as intended, for this reason we will prioritise that the system is tested. This will promote the system's reliability, and ensure that our system effectively helps the technicians with their work.

Flexible: Since the psychical methods and the work in the workshop are not subjective to rapid fluctuations, it's not crucial for the system to be able to adapt at the spur of the moment for the users of the system to benefit from systems' functionality. With that stated, the system will be developed using object-oriented principles, which inherently provides some flexibility for later modification of the system.

Comprehensible: Considering that much of the functionality of the new system derives from requirements gathered both from surveying the technicians, the intended users of the system, and the functionality they requested from the current system, it should be easily comprehensible for the users. Along with the fact that all of the intended users have a strong technical background, comprehensibility will be deemed less imperative for the system.

Reusable: Since this project is under the constraint of being implemented in Java, most of the system parts cannot be directly reused by Blue City, since their systems are based on C# and .Net architecture. Hence thinking of reusability is more or less irrelevant for this project.

Portable: Regarding portability, a requirement for the new system is to support multi-user operability, but since this criterion is easily fulfilled by choosing an architectural pattern that facilitates such behavior, we deem this criterion less significant to problem and application domains.

Interoperable: Since the system we are developing for Blue City is meant as a stand-alone solution only to be used by the workshops, the ability to access information from their company-wide system directly is not necessary for our solution to function as intended.

Considering this list of criteria and their significance to the system, we have to prioritise which criteria should have the greatest impact on our choice of system architecture. Based on our considerations of each criterion, we have chosen to prioritise as such:

Criterion	Very important	Important	Less Important	Irrelevant	Easily fulfilled
Usable	X				
Secure			X		
Efficient					X
Correct		X			
Reliable	X				
Maintainable				X	
Testable		X			
Flexible			X		
Comprehensible			X		
Reusable				X	
Portable					X
Interoperable			X		
Additional criteria		X			

7.1.1 Architectural Pattern

Considering the prioritisation of the criterion, the modularity of the system's components, and the "must-have" requirement, that the system allows for multiple users interfacing with the system concurrently, we decided to design the system around the client-server architecture pattern.

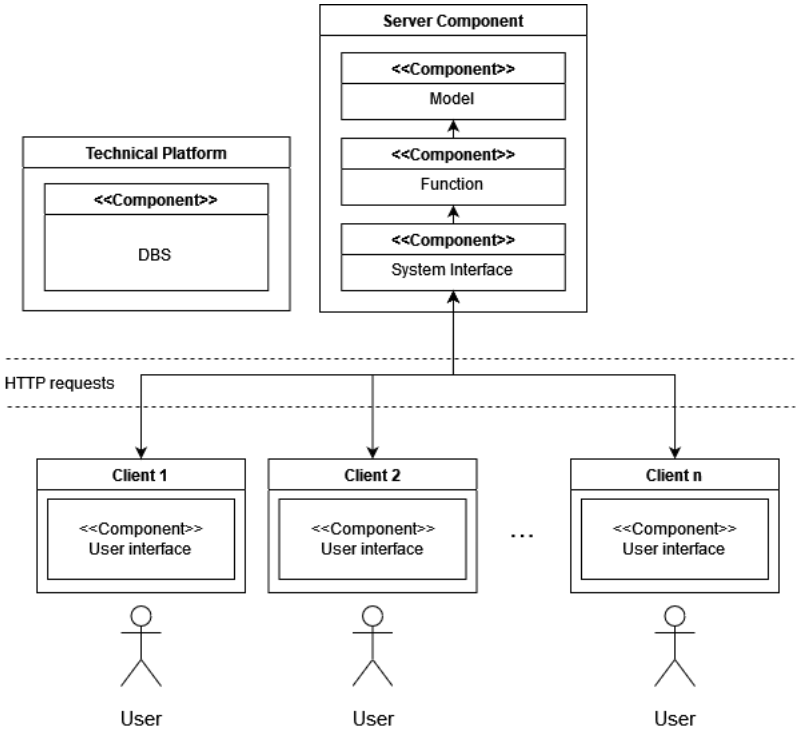


Figure 7.1: System Architecture Diagram

As seen in the diagram our System Architecture contains three main components:

- Server Component: an Ubuntu server with our java Spring system, which contains different logical components: Model, Function and System Interface components.
- Client Component: contains our client-application, which solely acts as GUI for the users to interface with our Spring Application through a REST application programming interface
- Technical Platform: contains our Database system, MongoDB Atlas, which is a service provided by MongoDB for hosting NoSQL databases in the cloud.

7.1.2 Client-Server Distribution

A Client-Server Architecture can have different forms of distribution for a system. Basically, these different forms of distribution are produced when we use the client-server pattern as a basis, and then distribute the model (M), function (F) and user-interface (U) components differently in each distribution.

In general, there are these five different forms of distributions:

Client	**Server**	**Architecture**
U	U + F + M	Distributed presentation
U	F + M	Local presentation
U + F	F + M	Distributed functionality
U + F	M	Centralised data
U + F + M	M	Distributed data

Table 7: Client-server distribution in chapter 9 in the System Development book[1, p. 202]

The client-server distribution that resembles our systems distribution is the *local presentation*.

8 Implementation

This section will cover how we tackled implementing our design on a technical platform. The section will contain the considerations we had during implementation and some of the issues we have encountered in the design. The section is split in two main sections, "Back-end" and "Front-end", going into detail on each of the program parts contained in these.

8.1 Back-end system

To define our back-end system, we take a look at the architectural diagram from section 7. In this, the system component is what defines the back-end system. The back-end will be the server application for our many client computers, responsible for communication between client and server. We chose not to over complicate the back-end system by choosing to work only with a central server wherein the back-end would be hosted. This allows us to deploy the back-end separately from the user-interface. Implementation of the object-oriented design regarding the back-end will be done in the object-oriented programming language Java. This of course followed nicely with the semester course "Object-Oriented Programming" in which we learn about java.

8.1.1 Spring Boot framework

To help us get started with writing a web-application, we chose to work with the popular java framework Spring boot [8]. Spring boot helps in many ways with the communication between different components. As an example, Spring boot offers a package to write REST APIs that will be used when making the system interface component, so that we do not need to worry too much about handling HTTP requests and other networking related activities.

Choosing to work with a framework, comes with some trade-offs. On the one-hand it limits the way in which we write code as we have to comply with how the spring framework developers have intended it to be used. On the other hand, it quickly gives us a configurable application to get started with.

The Spring boot framework comes with a lot of predefined annotations that are injected into our java source code, to tell the compiler what a particular part is, in relation to the rest of the program. As an example, when writing the REST API, we use the @RestController annotation on top of the controller class to indicate a spring REST controller. Similarly the annotation @Autowired, tells the compiler to instantiate the object that is annotated and wire it to an underlying repository in the database.

```
1 @RestController
2 public class SparePartController {
3     @Autowired
4     SparePartService sparePartService;
5
6     //Intentionally left out ...
7 }
```

Listing 1: Example use of Spring boots annotations found in "SparePartController" class

8.1.2 Product

The *Product class*' responsibility is to represent the actual products found in the problem domain analysis 3.1 and all related information regarding these products, including the product state, spare parts, etc. Throughout

the system objects created from this Product class is used as a central element in other classes such as the Repair class, Write-off class, and are even used in the generation of spare part objects from these products. As per our system requirements, the objects from this class are also used to recommend which product is the most ideal for the workshop to repair. This section will cover the implementation of the Product class, how we have chosen to model the class, and the methods/functions needed in the system, to support the above-mentioned classes and their functionality.

8.1.3 Product Class

If we look at the fields for the product class, we can see the different values that can be contained within a single object from this class. In order to the create a Product object, the system will need to pass several values matching an actual product to the constructor directly or use the product parser, which will be discussed later in this section.

```
1  public class Product {
2      @Id
3      private String id;
4      private String productId;        // In store product ID
5      private String name;
6      private String model;            // Ex: Pro, E480, 8, 9, 11 Pro
7      private String brand;            // Apple, Lenovo
8      private Category category;       // Smartphone (and iPhone), Laptop, MacBook
9      private String specification;    // Ex. 128GB, white
10     private LocalDate dateAdded;     // Date added to the shop
11     private ProductState state;      // DEFECTIVE, IN_REPAIR etc.
12     private double salesPrice;       // Can be sold for this amount
13     private double costPrice;        // Was bought for this amount
14     private String serialNumber;     // 356571101513554
15     private String defectiveComment; // Will not charge
16
17     ...
18 }
```

Listing 2: Product Class Attributes

In the products classes fields, it is noticeable that some values are more generic than others:

Generic values such as: productId, model, brand, specification, salesPrice, costPrice, serialNumber and defectiveComment, are all values initially taken directly from actual products.

The more system specific values:

- **id**: *12-byte hexadecimal* string (4-byte timestamp value, 5-byte random value, 3 byte incrementing counter), used to fetch specific products from the database.
- **name**: composite *String*, consisting of brand, category, model and specification. This is the way a product is described in Blue City's own system.
- **category**: *Enum* values matching the category as labelled in the workshop (iphone, macbook, ipad, laptop, smartphone and tablet)
- **state**: *Enum* values representing the different states a product can appear in (defective, in_repair, in_writeoff, written_off and repaired)

- **dateAdded**: *LocalDate* value used to track how long a product has been present in the system.

8.1.4 Product Controller

As mentioned earlier in Section 8.1.1, the system is built upon the Spring boot framework, in order to serve information from the system and database, to any client over HTTP. In *ProductController* Class, the logic for each of the specific endpoints related to products is implemented.

The following endpoints are implemented:

- **GET** : "/products" – Requests to this route returns all products available in the system. The requester is also allowed but not required to specify conditions for the products to be returned, such as: name, model, brand, etc.
- **GET** : "/products/{id}" – Requests to this route return the values for the specific product with the path variable: id, see Section 8.1.3
- **GET** : "/products/productId/sparepart_types" - Requests to this route return all the available types of spare parts available for the requested product's category. Further explained under the *CompatibleSparePart-TypeMap* Section 8.1.6.
- **POST** : "/products" – Post requests to this route enable users to add a single product to the system. This requires the request to contain form data needed in the class constructor.
- **POST** : "/products/file" – Post requests to this route enable users to add multiple products to the system by uploading a CSV file. Further explained under the *Parsing Product Files* Section 8.1.5.

8.1.5 Parsing Product Files

Since Blue City's workshop has many different products in stock and keeps receiving more products every day, it would not be sufficient for the manager to upload each product individually. The most direct solution to this problem would be to pull the information directly from Blue City's internal system, as their current Priotool does. But due to the constraints of the project scope, and the security precautions needed in order to directly interface with Blue City's internal system, we decided that a user should be able to upload CSV files containing the data for multiple products. This subsection will cover the implementation of said functionality.

```
1     @PostMapping("/products/file")
2     public ResponseEntity<Object> uploadProducts(@RequestParam("File") MultipartFile multipart
      ) throws IOException {
3         ...
4             try(BufferedReader reader = new BufferedReader(new InputStreamReader(multipart.
      getInputStream()))) {
5                 List<Product> productList;
6                 reader.lines().map(ProductParser::parse).toList();
7
8                 for (Product product: productList) {
9                     // Save all Products
10                    repository.save(product);
11                    System.out.println(product);
12                }
13            }
14        ...
15    }
```

Listing 3: Product Class Attributes

The controller expects a request parameter named File with the type **MultipartFile**. Multipart is an interface supported by the *springframework.web.multipart* [9]. This interface supports several methods, but the method we are interested in, is *getInputStream* which returns an InputStream to read the contents from the file. This inputStream is then used as an input for a BufferedReader, which buffers the character stream for efficient reading of file contents [10]. Now that the bytes of the file have been read from the input stream, each line is then mapped with the ProductParser, which takes a single line and parses the string's value to create a new product, returns the product and then appends it to a List of products. Lastly, the list of products is iterated over and saved to the product repository.

8.1.6 CompatibleSparePartTypeMap

For the functionality implemented in the product controller, there is the route defined which enables users to get which spare part types are available to any of the different product categories mentioned earlier in Section 8.1.3.
We found it necessary to implement this functionality for the technicians to easily find spare parts in the system during a product repair. To enable this functionality for the technician, we created an endpoint, as seen in line 1, with the route:

- **GET** : "/products/productid/sparepare_types".

```
1   @GetMapping("/products/{productId}/sparepart_types")
2 public ResponseEntity<?> getCompatibleTypes(@PathVariable String productId) {
3
4     Optional<Product> product = repository.findById(productId);
5
6     ...
7
8     List<SparePartType> sparePartTypes = CompatibleSparePartTypeMap.conversionMap.get(
      productModel.getCategory());
9
10    return new ResponseEntity<>(sparePartTypes, HttpStatus.OK);
```

Listing 4: getCompatibleTypes Controller

This route takes a string as a Path Variable, which may equivalate to any given product stored in the system. This product id is then used to query the system's database to find the given product. If a product with the id is found, the category of the product is then passed as a parameter, to the **conversionMap.get** method, which returns the compatible spare parts as a List of enums with the type of **SparePartType.**

```
1 public class CompatibleSparePartTypeMap {
2
3     public static final Map<Category, List<SparePartType>> conversionMap = new Hashtable<>();
4     static {
5         conversionMap.put(Category.IPHONE, List.of(SparePartType.SCREEN,
6                                                    SparePartType.BATTERY, SparePartType.
      HOUSING,  SparePartType.CHARGING_DOCK,    SparePartType.FRONT_CAMERA,
7                                 SparePartType.BACK_CAMERA,      SparePartType.LOGICBOARD));
```

The **conversionMap** is implemented as a Map which stores the **product categories** as the keys, and a list of enum values from the **SparePartType** enum, related to that specific category, as values.

To add the specific key-value pairs to the mapped, we use the Map method put, to create the key-value pairs for each of the catogories and the respective spare part lists.

8.1.7 Spare parts

In the class-diagram in figure 3.1, we modelled spare parts using a generalisation structure, saying that a used spare part and a new spare part are both a spare part. Implementing this way of viewing the spare part can be done using simple inheritance. We define a super class "Spare part" that have common attributes for both a

used and a new spare part. These include name, brand, category, model, added date, cost price, part number and a few other attributes. Both the used and new spare part *extends* this super class and will be inheriting its attributes. The specific things that describe a used spare part and a new spare part will then be put into these two separate classes. This allows us to have any of the two spare part types be present in a repair object, while for example only having a relation from the used spare part to a write-off ticket.

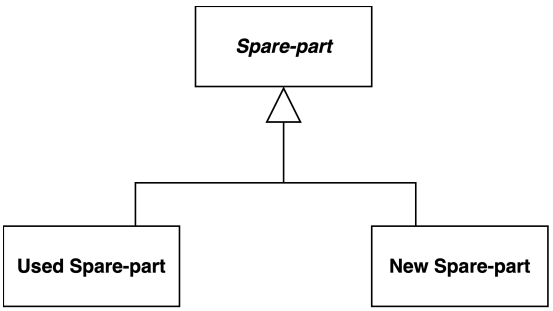


Figure 8.1: Simple spare part model

Behaviour

The behaviour of a spare part is modelled for both a used spare part and a new spare part which was done during problem domain analysis in Section 3.3. In this example we show only how the used spare part is implemented, as the new spare part has some missing implementation parts. To recap on what is going to be implemented we bring up the state chart diagram for the used spare part as can be seen below in Figure 8.2.

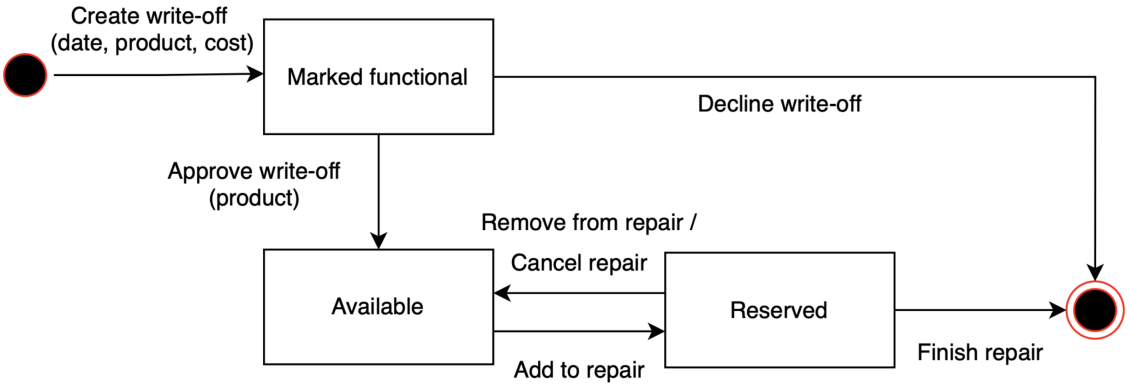


Figure 8.2: Statechart diagram for the class "Used spare part"

The state of a spare part is implemented using an internal state variable for the spare part object. We did this using an enum type. The following table shows which states are used for both the used and the new spare part. Red means it is not a state of the spare part and green means it is. Every operation that acts on the spare part and changes its state is manipulating this state variable. Checks are in-place to insure that the object can not go from example the "Marked functional" state into "Reserved" state without first going through the "Available" state. During implementation we introduced a new state "Consumed" as an intermediary step between deletion for making statistical measures.

Spare Part State	New	Used
AVAILABLE		
RESERVED		
CONSUMED		
MARKED FUNCTIONAL		
ON_ORDER		
INCOMING		

Table 8: Table showing spare part states

All of the events that act on the behaviour of a used spare part are placed mainly in the repair service and write-off service that implements the operations for creating write-offs, approving write-offs, cancellation of repairs etc. These operations have the side effects of changing the spare parts state, and so the spare part class in it self can be viewed as a static storage class.

8.1.8 Repair

Repairs are modelled in the problem-domain analysis in Section 3 and are an essential part of the system, as the system's main objective is to improve Blue City's repair process. The main responsibilities for the class Repair is to keep track of which products are being repaired, the time spent repairing the product as well as the spare parts used during the repair process. In the Functions Section 4.2 the identification of functions and their complexity is described, and in this section it is also outlined how several complex functions are required in the process of repairing a product.

Repair Class

The *Repair* class is responsible for constructing the repair objects and managing the information contained in these objects. The information the object is required to contain can be seen in the fields of the class below.

```
1  public class Product {
2      @Id
3      private String id;
4      private Product product;
5      private RepairState state;
6      private Date startDate;
7      private Date endDate;
8      private Date pausedAt;
9      private Date resumedAt;
10     private long pausedTime = 0L;
11     private long repairTime = 0L;
12     private List<SparePart> spareParts = New ArrayList<>();
13 }
```

Listing 5: Repair Class fields

As can be seen in the fields listing above, we have the fields **startDate, endDate, pausedAt, resumedAt, pausedTime** and **repairTime** these fields contain information used to keep track of time spent on a repair, which is a requirement from Blue City's management. **RepairState** is an enum class that has constants used to describe whether a repair is on-going, paused or finished. The **Product** field contains the *Product* object which was used to initialize the repair, and the **List<SparePart>** is initialized as an empty Array, which can contains the *SparePart* objects added during the repair.

The constructor for the *Repair* class takes a *Product* object as input and sets the state, starting date and product for the newly created object, as can be seen in the listing below.

```
1     public Repair(Product product) {
2         this.state = RepairState.ON_GOING;
3         this.startDate = new Date();
4         this.product = product;
5     }
```

Listing 6: Repair constructor

The rest of the fields are populated with values later using the corresponding set methods of the *Repair* class. These methods are used in collaboration with the *RepairService* interface in the *RepairController*. The following subsection will go into detail with the correlation between the *RepairService*, the *RepairServiceImpl* and the *RepairController*.

Repair Service Implementation

The *RepairServiceImpl* class is responsible for implementing the logic to the methods provided by the *RepairService* interface. These methods are in turn used by the *RepairController* which relays the information between the server and the client through HTTP requests.

In the **RepairController** the following endpoints are implemented:

- **GET** : "/repairs" - Requests to this route calls the **getRepairList** method from the **RepairService** interface which returns all the repairs within the repository.
- **GET** : "/repairs/{id} - Requests to this route calls the **getRepairByID** method from the **RepairService** interface and returns the repair with the **id** provided as the path variable.
- **POST** : "/repairs" - Requests to this route calls the **createRepair** method from the **RepairService** interface. An **@id** for the product is required as a request parameter, and the requests updates the RepairState of the product and saves both the repair and the product in their respective repositories.
- **POST** : "/repairs/{id}/pause" - Requests to this route calls the **pauseRepair** method from the RepairService interface. The method updates the RepairState to "PAUSED" and the pausedAt time, for the repair with the **id** provided as the path varibable. An exception is thrown in case no repair is found with the given id or in case the provided repair is not in the "ON_GOING" state.
- **POST** : "/repairs/{id}/resume" - Requests to this route call the **resumeRepair** method from the **RepairService** interface. The method searches the repository for the repair with the path variable **id** and updates its RepairState and its pausedTime.
- **POST** : "/repairs/{id}/cancel" - Requests to this route calls the **cancelRepair** method from the RepairService interface. It cancels the Repair with the **id** provided as the path variable . As a Repair can have SpareParts added to it with the addSparePart method, these have their SparePartState updated to "AVAILABLE". In order to achieve this, the method getAddedSpareParts collects a list of the SpareParts added to the repair, and sets the state to "AVAILABLE" on each of them. The SpareParts are then disassociated with the Repair.
- **POST** : "/repairs/{id}/finish" - Requests to this route calls the **finishRepair** method from the RepairService interface. This complex function is explained in further detail in section 8.1.8 below.

- **POST** : "/repairs/{repairId}/add/{sparepartId}" - Requests to this route calls the **addSparePart** method from the RepairService interface. This method allows a SparePart to be added to a Repair, both **ids** provided with the request as path variables. To keep track of which spare parts are available to use and which ones are not, the state of the SparePart is set to "RESERVED", when it is added to a repair.
- **POST** : "/repairs/{repairId}/remove/{sparepartId}" - Requests to this route calls the **removeSparePart** method from the RepairService interface. The method functions contrarily to **addSparePart** and has a purpose in two situations. The first is when a repair is cancelled, in this scenario the technician forfeits the repair and removes the spare part that was applied to the product. The second situation is when a spare part appears to be faulty and has to be removed, in order for the technician to apply a functional one.

Finishing a repair

The most complex method pertaining repairs is the **finishRepair** method called through the "/repairs/id/finish" endpoint. The method relies on many of the "get" and "set" methods provided by the **Repair** class and is responsible for finishing the repair, which includes updating cost price of the product being repaired, setting the state of the spare parts from "used" to "consumed" and setting the repair time for the specific repair.

```
1   public void finishRepair(String id) {
2
3     [...]
4           if (repairModel.getState().equals(RepairState.ON_GOING)) {
5
6               repairModel.setState(RepairState.FINISHED);
7               repairModel.setEndDate(new Date());
8               repairModel.setRepairTime((
9                       (repairModel.getEndDate().getTime() - repairModel.getPausedTime())
10                              - repairModel.getStartDate().getTime()));
11
12              productModel.setCostPrice(productModel.getCostPrice()
13              + repairModel.getRepairCost());
14              productModel.setState(ProductState.REPAIRED);
15
16              repairModel.getAddedSpareParts().forEach((sparePart -> {
17                  sparePart.setState(SparePartState.CONSUMED);
18                  sparePartRepository.save(sparePart);
19              }));
20
21              productRepository.save(productModel);
22              repairRepository.save(repairModel);
23          }
24          throw new IllegalRepairOperationException("Repair must be on-going before it can
    be finished");
25      }
26      [...]
27  }
```

Listing 7: finishRepair

The listing above shows a code snippet from the **finishRepair** implementation after a **Repair** is found with the path variable id and set to the local variable repairModel, the product of this repair is set to productModel. The method asserts that the repairState of the Repair is "ON_GOING" and throws an IllegalRepairOperationException, if it is not. We then proceed to update the RepairState with **setState**, the endDate with **setEndDate** and

the repairTime is calculated using startDate, endDate and pausedTime. This allows us to satisfy the requirement of keeping track of the total time spent repairing a product. In order to satisfy the requirement of adding the cost of spare parts used during the repair to a product, the method calls the repair method **getRepairCost**, which sums up the cost of each SparePart in the list and adds this to product. The state of the Product is set to "REPAIRED" and each SparePart has its state set to "CONSUMED", such that they can no longer be used. Lastly the updates made to the Repair, Product and SpareParts are saved in their respective repositories.

8.1.9 Write-off

To model the problem domain process of writing off a product we have implemented the class *WriteOffTicket*. This class is responsible for remembering events relating to the product being written off, the spare parts that can be salvaged from the product and which technician is suggesting the write-off. The class *WriteOffTicket* depends on several methods contained in the class *WriteOffServiceImpl*, which in turn rely on the class *WriteOffTicketController*, which is responsible for handling post and get requests from the client users.
The creation of a write off ticket involves multiple classes and is the most complicated of the processes regarding write-offs, outlined here in four steps:

1. Collect the data posted by the user in a controller method using the class *WriteOffTicketForm*.
2. Pass that data to a service method called **createWriteOffTicket**, which parses the data from String objects to the enum SparePartType using the method **parseTypes** in the class *WriteOffFormParser*.
3. Create new SparePart objects from the data and a new WriteOffTicket object to connect product, spare parts and technician.
4. Save modifications to involved objects and return a response via the controller to the user.

WriteOffTicket Class

This class represents a write-off process concerning a product and any spare parts that may be salvaged. Therefore it contains fields to hold information about these objects and events since they logically belong together in one place. Since the class contains SparePart objects and a Product object, all relevant information about a write-off and the aggregated objects can be accessed through a single object. This is convenient in that, if we require information about a write-off product's category, we can access it through the WriteOffTicket object without having to query the database for the product.
The **productId**, **reason**, and **technicianName** are included because they are important for a manager when deciding whether to approve or decline a write-off.
Finally, to allow for a write-off to be initiated by a technician, but only approved or declined by a manager, the class has the field **state**. This allows us to differentiate between write-offs in different states, and control which operations are allowed, given its state. For instance, as we show in section 8.1.9, we do not allow a write-off to be allowed or declined if it is not in the state: AWAITING, and the program can throw a relevant exception in such an abnormal situation.

```
1  public class WriteOffTicket {
2
3      @Id
4      private String id;
5      private List<SparePart> sparePartList = new ArrayList<>(); // Contains the functional
6                                                                 // spare parts to be made
7                                                                 // available on approval
8      private Product product;            // The product for write-off
9      private String productId;           // Id of product for write-off
10     private String reason;              // Reason for making the write-off
11     private String technicianName;      // Name of technician initiating the write-off
12     private Date creationDate;
13     private Date approvalDate;
14     private WriteOffTicketState state;  // AWAITING, APPROVED, DECLINED
15
16     // constructor
17     public WriteOffTicket(Product product, String technicianName){
18         this.product = product;
19         this.productId = product.getProductId();
20         this.technicianName = technicianName;
21         this.creationDate = new Date();
22         this.state = WriteOffTicketState.AWAITING;
23     }
24     // getters and setters...
25 }
```

WriteOffTicketForm Class

This class is used to hold the data from the POST request body that, in JSON format, contains the reason for initiating the write-off and the types of parts marked functional in the product being written off.

```
public class WriteOffTicketForm {

    private String reason;                                  // the reason for write-off
                                                            // supplied in post request
    private List<String> markedParts = new ArrayList<>();   // the type of spare part marked
                                                            // functional in post request
    // getters and setters...

}
```

WriteOffFormParser Class

This class contains a single method that is used to return a list of SparePartType enums from the string values in a WriteOffTicketForm object. This conversion is necessary because we use enums to define the different types of spare parts in our program and to instantiate SparePart objects. The method is public static so that it can be called in *WriteOffTicketServiceImpl* without instantiating a WriteOffFormParser object.

```
public class WriteOffFormParser {
    public static List<SparePartType> parseTypes(WriteOffTicketForm woForm)
    {
        return woForm.getMarkedParts().stream().map(SparePartType::valueOf).toList();
    }
}
```

WriteOffTicketController Class

This class contains methods responsible for handling GET and POST requests. Their job is to pass the received information to the correct methods in *WriteOffTicketServiceImpl*, and handle exceptions thrown by those methods. Below the specific PostMappinng and method involved in creating a write-off is shown. The *consumes* "*application/json*" is required for the mapping to receive a json object, which it converts to a WriteOffTicketForm object called woForm to pass to the service method *createWriteOffTicket(...)*.

```
    @PostMapping(value = "/writeoffs/create", consumes = "application/json")
    public ResponseEntity<?> createWriteOffTicket(@RequestBody WriteOffTicketForm woForm,
                                                  @RequestParam(value = "prod_id") String prod_id,
                                                  @RequestParam(value = "tech_id") String tech_id)
    {
        try {
            writeOffTicketService.createWriteOffTicket(woForm, prod_id, tech_id);
            return new ResponseEntity<>("Write-off created", HttpStatus.CREATED);
        } catch (NoSuchElementException e) {
            return new ResponseEntity<>(e.getMessage(), HttpStatus.NOT_FOUND);
        } catch (RuntimeException e) {
            return new ResponseEntity<>(e.getMessage(), HttpStatus.INTERNAL_SERVER_ERROR);
        }
    }
```

WriteOffTicketServiceImpl Class

The last class involved in a write-off is the *WriteOffTicketServiceImpl* class. This is where most of the methods and their logic controlling the write-off process is located. In this section, we discuss two of them.
The method *createWriteOffTicket* contains logic to ensure that the product being written off exists in the database and has the right state: DEFECTIVE. Exceptions are thrown if these conditions are not met, otherwise all of the relevant objects such as product, spare parts, and write-off ticket are manipulated, created and saved in the database before the created write-off ticket is returned to the controller method.

```
1   public WriteOffTicket createWriteOffTicket(@RequestBody WriteOffTicketForm woForm, String
    prod_id, String tech_id) {
2
3       Optional<Product> DBproduct = productRepository.findById(prod_id);
4
5       if (DBproduct.isEmpty())
6           throw new NoSuchElementException("Item not found in database");
7
8       Product product = DBproduct.get();
9
10      if (product.getState() != ProductState.DEFECTIVE)
11          throw new RuntimeException("Product state not applicable for write-off");
12
13      List<SparePartType> sparePartTypes = WriteOffFormParser.parseTypes(woForm);
14
15      product.setState(ProductState.IN_WRITEOFF);
16      productRepository.save(product);
17
18      WriteOffTicket ticket = new WriteOffTicket(product, tech_id);
19      ticket.setReason(woForm.getReason());
20      ticket.setTechnicianName(tech_id);
21
22      /* Generate used spare-parts from the product under write-off and add to the ticket */
23      for (SparePartType type : sparePartTypes) {
24          UsedSparePart part = new UsedSparePart(product.getProductId(), product.getBrand(),
25          product.getCategory(), product.getModel(), type,product.getCostPrice() / (double)
26          sparePartTypes.size());
27          part.setState(SparePartState.MARKED_FUNCTIONAL);
28          sparePartRepository.save(part);
29          ticket.addSparePart(part);
30      }
31
32      return writeOffTicketRepository.save(ticket);
33  }
```

Below is a short excerpt of the method **approveWriteOffTicket** showing, as mentioned in section 8.1.9, a check at line 10 where we confirm whether a write-off is AWAITING before allowing its approval, thus preventing allowing tickets that are in an incompatible state.

```
1     public void approveWriteOffTicket(String id) {
2         Optional<WriteOffTicket> DBticket = writeOffTicketRepository.findById(id);
3
4         if (DBticket.isEmpty())
5             throw new NoSuchElementException("Write-off ticket not found in database");
6
7         WriteOffTicket ticket = DBticket.get();
8
9         if (ticket.getState() != WriteOffTicketState.AWAITING)
10            throw new RuntimeException("Write-off ticket is not awaiting");
11
12    // intentionally left out...
13
14    }
```

8.1.10 Users

In the problem domain analysis in section 3.1, we deemed there only needed to be a technician class and not a manager class, since we were not going to log anything to do with the manager. However as the implementation started, it quickly became apparent, that we needed both classes, since both needed a way to login into the system with varying levels of access. After further discussion, we realised that the two classes were identical in all ways but one: What degree of authority they have in the system. So instead the two classes were merged into one class: The *User* class. In order to allow for the roles of manager and technician in this class, there needs to be a distinction in the system between these two roles. This distinction comes from the field "privileges" in the *User* class.

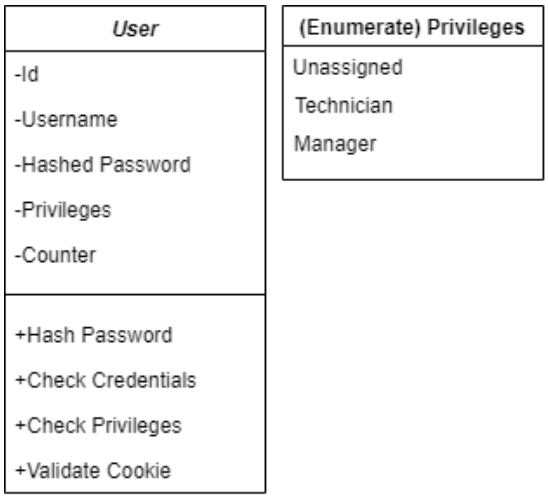


Figure 8.3: User Class - Manager is called "$FULL\_ACCESS$" and technician "SEMI_ACCESS" in the code

This way different users can be assigned different levels of privilege. All users start as **Unassigned** without any access privilege at all. Only a person with the highest privilege, **Manager**, can change other users' privilege. In this way, all new users need to be authorised by an existing manager. This is done to make sure no user, has access to the entire system by default. As another preventive measure, which was of high priority to Blue City,

the only user who can create other users, is the manager. This prevents anyone else from creating duplicate accounts or allowing individuals currently not employed by Blue City to access the system.

Authentication of users

When communicating with the front end, every time there is a restricted page, the function **checkPrivilege** on the front end is called, and if the privilege is too low, access is denied. This is done with a conditional statement in react, where it will only render the page if the user who is logged in has the right privilege. Unassigned users have access to nothing, and technicians have access to all but a few administrative pages.

When a user wishes to login to the site, they will type their username and their password into a login screen (see Appendix C.8). This information will be sent to the back end where the method **checkCredentials** is called.

```
1 public boolean checkCredentials(String inputName, String inputPassword){
2     if(this.username.equals(inputName) && this.password
3     .equals(SHA3.hashPassword(inputPassword)))
4     {
5         return true;
6     }else return false;
7 }
```

This method hashes the password that they provided, and checks if it is equal to the hashed password stored in the database. If it is, a token is sent back to be stored in the browsers LocalStorage (see appendix C.9). Every time the user sends a request to the system, the method **authenticateSessionCookie** from the user controller is called. If the cookie is validated, then the requested information is returned if they also have the necessary privileges.

Encryption

The reason for hashing, is to make sure sensitive information won't be exposed in the case of a data breach, which would be a risk to employees if the data were to be leaked. When a user account is created, their password is hashed using the method **hashPassword** before it is stored in the database. The hashing algorithm we use, is the SHA3 256 byte hash. The function is assembled using multiple functions from the standard java library "Java.security".

```
1  public class SHA3 {
2
3      private static final Charset UTF_8 = StandardCharsets.UTF_8;
4      private static final String OUTPUT_FORMAT = "%-20s:%s";
5
6      public static byte[] digest(byte[] input, String algorithm) {
7          MessageDigest md;
8          try {
9              md = MessageDigest.getInstance(algorithm);
10         } catch (NoSuchAlgorithmException e) {
11             throw new IllegalArgumentException(e);
12         }
13         return md.digest(input);
14     }
15
16     public static String bytesToHex(byte[] bytes) {
17         StringBuilder sb = new StringBuilder();
18         for (byte b : bytes) {
19             sb.append(String.format("%02x", b));
20         }
21         return sb.toString();
22     }
23
24     public static String hashPassword(String password){
25         String algorithm = "SHA3-256";
26         byte[] passwordInBytes = SHA3.digest(password.getBytes(UTF_8),algorithm);
27         return bytesToHex(passwordInBytes);
28     }
29 }
```

What is happening in the code here is:

1. The message is first turned into a byte stream encoded as UTF-8
2. Then it message digests the byte stream. A message digest is a one way hash function, meaning it turns the variable length byte stream into a fixed length hash
3. The byte stream is then reformatted into hexadecimal, which is then turned into a readable string.

By the nature of hashes, a secure hash is a collision resistant hash: Simply a hash where the likelihood of two inputs creating the same hash as output being computationally improbable. This makes it secure, since you cannot request a lot of different hashes and try to find a pattern by colliding the hashes. A property of this is, that it is extremely easy to turn something from plaintext (the raw message) to ciphertext (the hashed message), but close to impossible to turn ciphertext into plaintext.
This property solely ensures the confidentiality of the plaintext (password), not integrity nor availability. For

ensuring integrity we could use a MAC (Message Authentication Code), but as it stands, it is not a priority. This makes this system extremely vulnerable to a man-in-the-middle attack.

Cookie

The controller for how the cookie is sent, can be seen here:

```
1   @PostMapping("/auth")
2     public ResponseEntity<?> authenticateUser(
3             @RequestParam(value="username") String username,
4             @RequestParam(value="password") String password) {
5
6         Query userQuery = new Query();
7
8         if (username != null) { userQuery
9         .addCriteria(Criteria
10        .where("username")
11        .is(username)); }
12
13
14            // Find users with matching username
15            User requestUser = operations.findOne(userQuery, User.class);
16
17            if (requestUser != null) {
18                // Check user password against input string
19                if (requestUser.checkPassword(password)) {
20                    requestUser.compoundCounter();
21                    String cookie = SHA3.hashPassword(requestUser.getCounter()
22                    +requestUser.getId());
23                    repository.save(requestUser);
24                    return new ResponseEntity<>(cookie, HttpStatus.ACCEPTED);
25                } else {
26                    return new ResponseEntity<>(
27                    "Password didn't match ", HttpStatus.FORBIDDEN);
28                }
29            } else {
30            return new ResponseEntity<>("No such user", HttpStatus.NOT_FOUND);
31        }
32    }
```

How it works is after a user's credentials are authenticated, a special login counter is incremented and saved to the user profile in the database. This incremented counter is then hashed together with the user Id as such:

- Cookie = Hash(Counter+Id)

and sent back as a cookie.

Every time a person logs in, this counter goes up. Which ensures that only 1 login session at a time is possible, and that simply copying another person's cookie does not allow for permanent access. Every time the system validates the cookie, a very similar function is called (without incrementing the counter) and compared to the cookie stored locally to check if they are equal.

Doing it this way doesn't make our encryption a form of perfect security, nor does it make it a form of semantic security, but more so just security by obfuscation, as is most of this login system.

One could improve the security in many different ways, such as encrypting all data stored in the database, but that is not crucial in the case of our system. It is not meant to be perfect, as it is of a low priority to create a very secure login system, as discussed in Section 7.1. There is also little incentive in actually hacking the system, as the amount of users in the database is minimal, and the monetary opportunity of breaking into the system is little to none. This makes the general risk of an attack, larger then what a script kiddie could attempt, small to none. Therefore a simple hindrance to hack the system and the sensitive data being hashed is adequate.

8.2 Database

The database chosen for this project is a non relational database: MongoDB. The reason for choosing this type of database, is that it is extremely easy to work with, and that the group had previous experience in working with document type databases. As an example, this is how the user class is stored within MongoDB:


```
_id: ObjectId("61b9a2d47c310f2d07bfc4b3")
username: "Bertan Smokes"
password: "c0067d4af4e87f00dbac63b6156828237059172d1bbeac67427345d6a9fda484"
initials: "BS"
dateRegistered: 2021-12-14T23:00:00.000+00:00
employmentTime: 0
userPrivilege: "FULL_ACCESS"
counter: 8
_class: "com.skarp.prio.user.User"
```


Figure 8.4: User stored in the Database

8.3 User-interface

In this section we describe how our user-interfaces have been implemented in the final design and some of the technologies used.

React

For our front end we choose React.js, which is a lightweight front end JavaScript library. It's dynamic, modularized and extremely powerful. This comes in handy when loading and updating a lot of different products fast and efficiently.

Interfacing

Our system interface is made up of multiple sub pages that can be accessed from the following homepage:

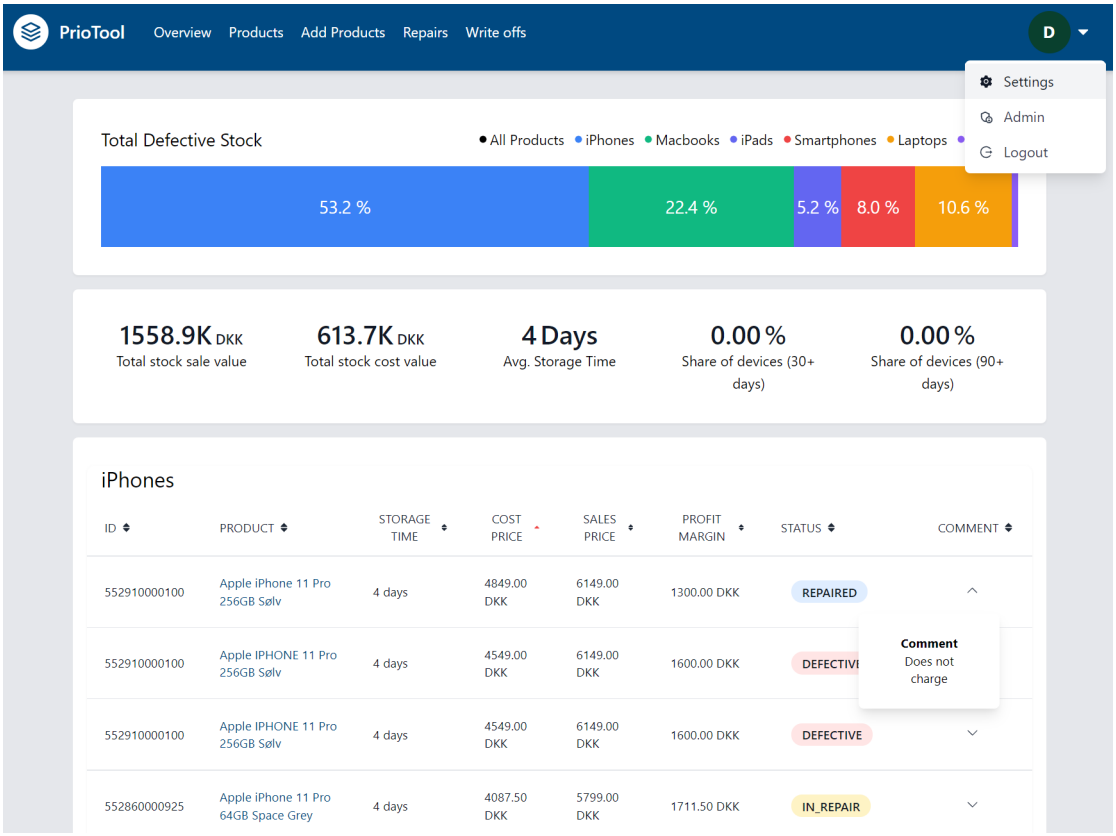


Figure 8.5: The homepage of our system

On the front page (Overview), there is a toolbar with links to:

- **The products page**: to list all products, and start repairs or write-offs on the different products. Clicking on the product name on a product on the front page takes you there.
- **The add products page**: that allows the user to add new products to the database
- **The repair page**: listing the current ongoing repairs
- **The write-off page**: listing all products scheduled to be written off by the technician. Only the manager can access this page

In the rightmost corner there is a personalised icon with the user's initials. Here you can access settings as well as the admin menu, which is only accessible by the manager.

The homepage includes an overview of the current defective stock, sorted by different categories, and other additional relevant information about the state of the workshops inventory. These categories were established in interviews with Blue City. Clicking on one of them filters all the products at the bottom of the page such that only that category is shown. For each category, all the products can then be sorted by the different labels as well.

We have carefully considered never to have more than 7 elements in each module at a time anywhere on the website in order to enhance usability. Furthermore, everything is at max 3 clicks away, to make sure everything is easily accessible for the end user, as shown in this site-map:

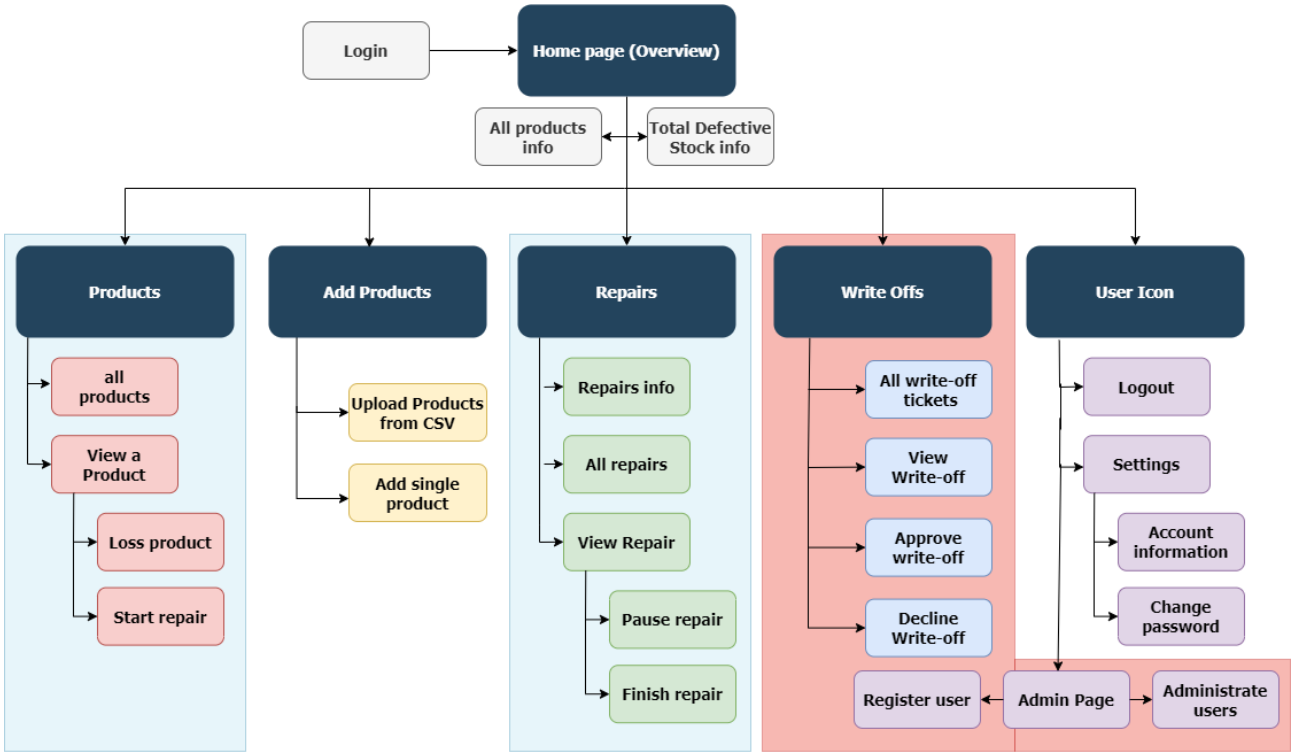


Figure 8.6: Sitemap

The colored background signifies:

- **Blue**: Primarily/exclusively the technician will be on these pages
- **Red**: Only the manager can access these pages, and it is these pages where they will spent most of their time
- **Not colored**: Both the manager and technician will visit and use these pages regularly.

Though it is not shown, there are usually multiple ways to get to each page, crossing over from different sub pages, in order to enhance user experience.

The user interface will be described as a function of the everyday workflow of the two actors: manager and technicians.

Manager workflow

The manager will have 4 different fundamental active tasks to do:

- Registering new technicians
- Overseeing technicians
- Approving write-offs
- Adding product(s)

Registering and overseeing technicians is done from the admin (manager) page, as shown in the sitemap:

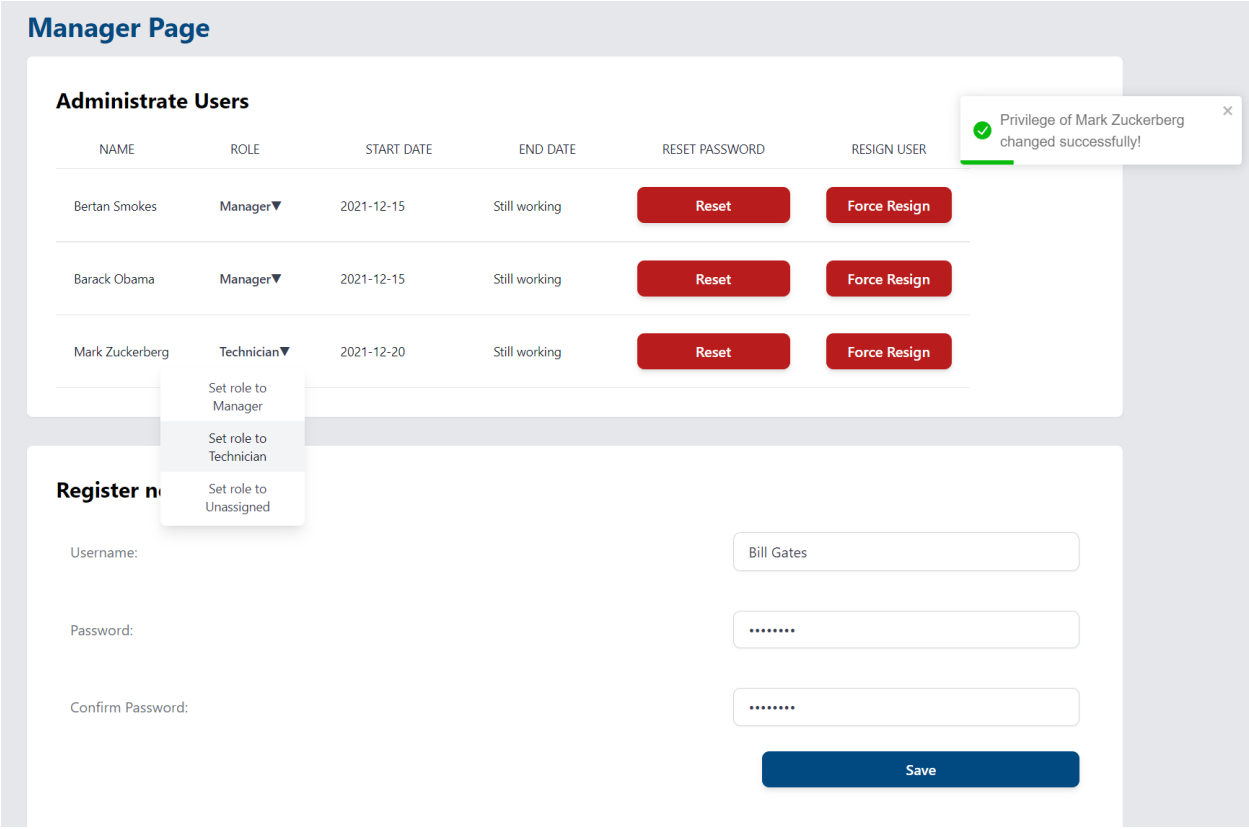


Figure 8.7: Admin Page

Here the manager can register new technicians as they are hired, resign them from the system, reset their password and change their role, and thus their privilege to access different areas of the system.

Approving write-offs from the menu item "Write offs" lets the manager approve or decline different write-offs, and only they can do it, in order to ensure a level of separation of duty:

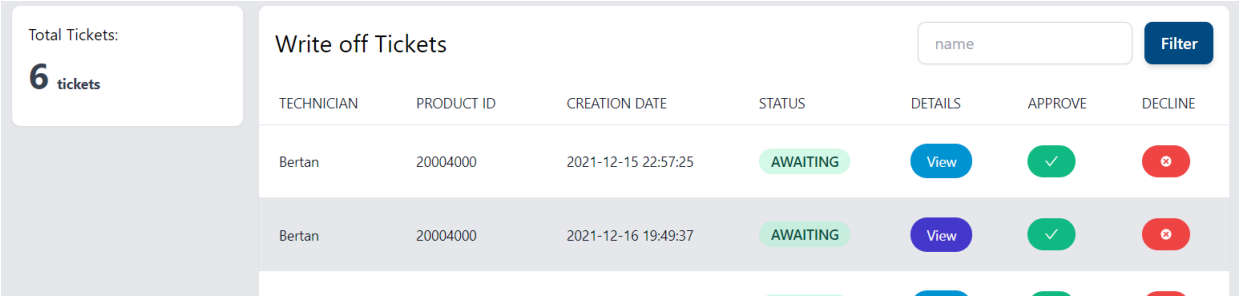


Figure 8.8: Write offs page

Before approving the write off, the manager might want to see the details of the write off, such as the spare parts marked as functional:

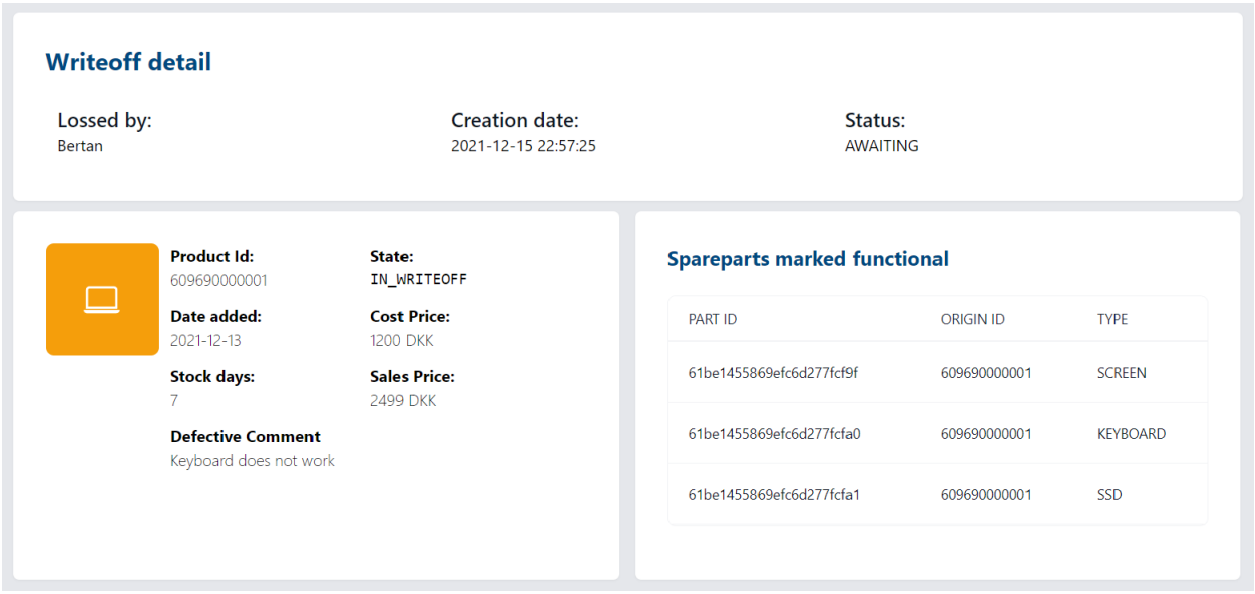


Figure 8.9: Write off detail

Lastly, the manager, as well as the technician, may want to **add a new product** to the system, either for migrating purposes, or if they have bought a single broken item. An illustration of the implementation can be seen in Appendix C.1.

Technician workflow

The technicians will have 3 different fundamental active tasks to do:

- Loss Product
- Start Repair
- Finish Repair

From either the home page or the products page, the technician can select a product they wish to either start a repair or loss process on.

If the technician has run diagnostics and deemed the product isn't worth repairing, they can loss the product, selecting the spare parts that can be recovered, and send it to the manager for write-off approval:

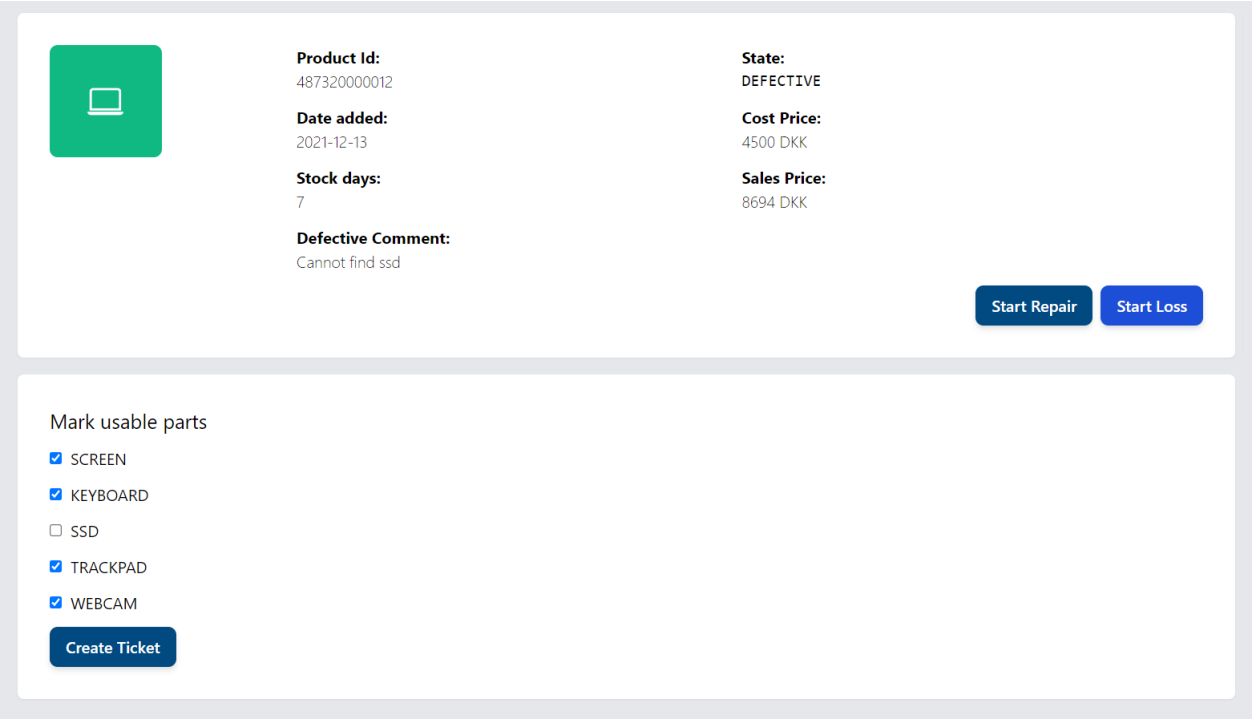


Figure 8.10: Loss product

If the technician instead decides to repair the product, they can click "start repair", which will bring them to the "repairs page":

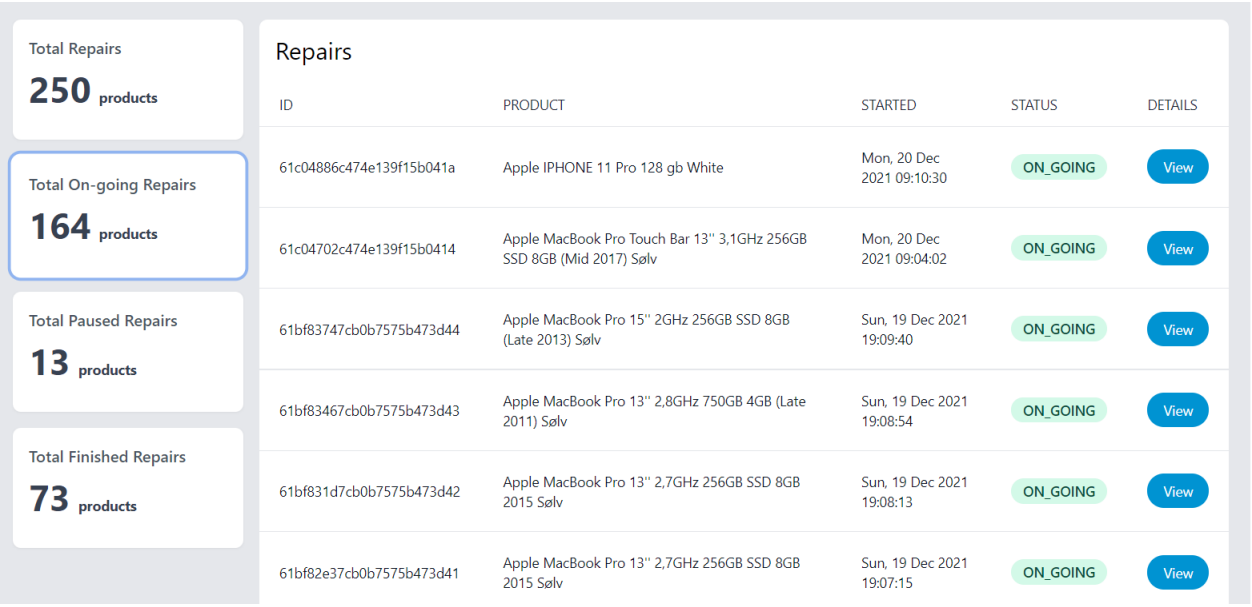


Figure 8.11: Repairs page

They can then click on view for a product they wish to repair (the one just added being the first one on the list), to see the repair information. Here the technician can add the required spare parts to the product. If they are missing a part, they can decide to pause the repair, while they wait for the spare part to be ordered. Once they are finished with the repair, they can click on finish repair. This will add the repair to the system as "finished", such that they can later on check what products have been repaired.

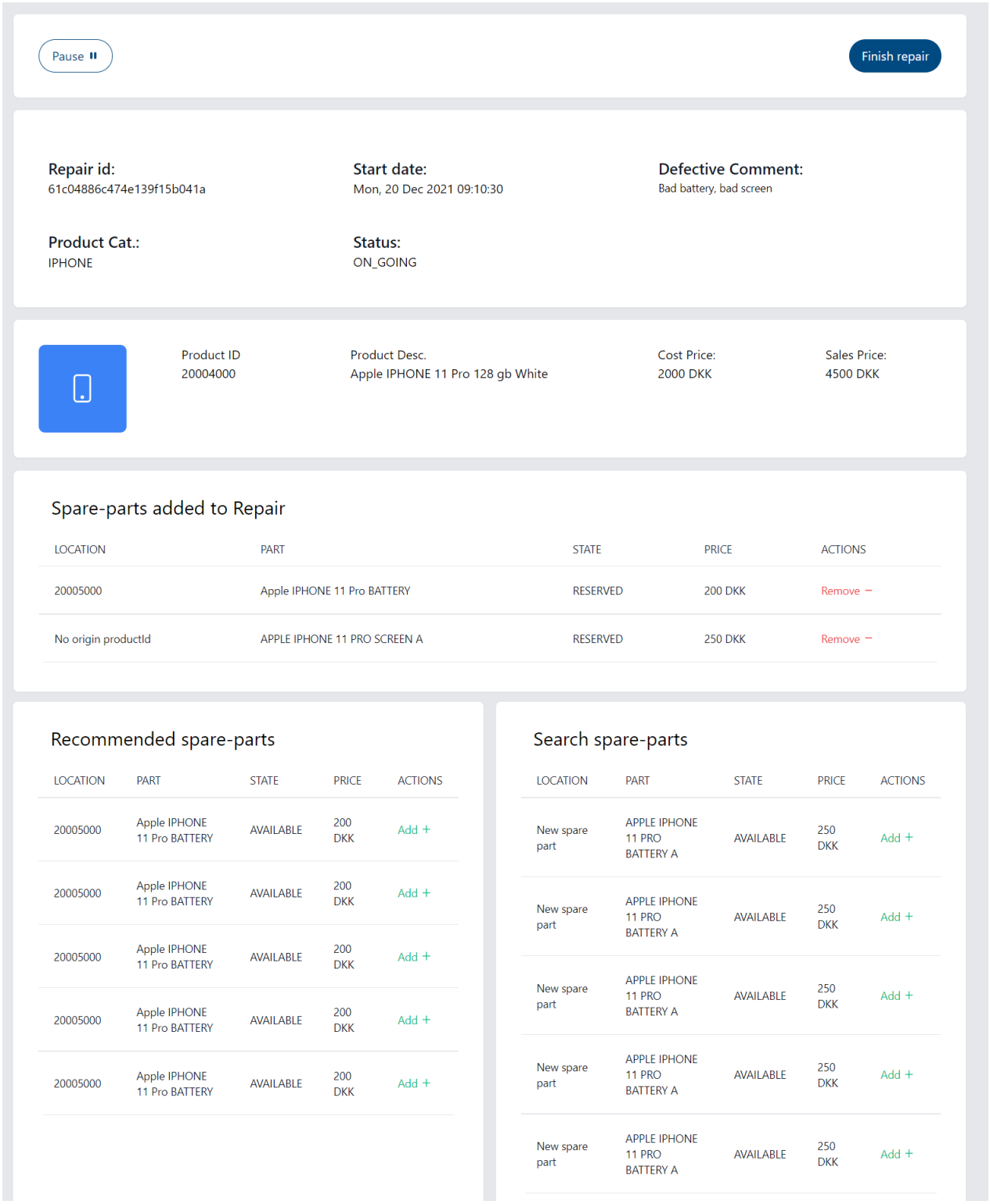


Figure 8.12: Repair detail

On the last page, the settings page, they can see their account information, and create a new password if they so desire. See illustration in appendix C.7.

9 Testing

Test Driven Development(TDD) is a method where unit tests are written before writing any production code. We tried it in the early stages of the implementation phase, designing the first classes and their methods. It was a great method to implement to follow the principle as an experiment, but it became more complex the moment we started implementing controller- and service methods. The complexity of these methods was so high it confused us a lot when trying to design tests before writing the code, possibly because we were learning how to write them while writing them. Instead we adopted the old fashioned approach of writing tests after writing the code.

Testing is used to identify errors in the code and ensure that all the functionalities fulfil the users' requirements, which are outlined in Section 5. Assuming our tests are considered a success, there can still be issues in the results where they do not meet the users' requirements. WHY?
In this project we will test our solution in 2 ways, each with its own purpose to fulfil. Usability testing to ensure good user experience and unit testing to verify that the code works as expected.
We are using unit testing to test the service layer, so we can identify if the system is able to create repairs, write-offs, etc. These tests are very important to hold the foundation of the system. The usability testing will cover all the functionalities which is to start a repair, write off, order spare parts, add product, login and register technician. In terms of test coverage, 100% code coverage is a goal to strive for, but due to the time constraints and the amount people in group, we do not find it realistic to reach 100%. We will test the most critical areas of the system, to ensure reliable functionality. Of course, we cannot ensure reliable functionality if we don't test 100%. And even then we cannot ensure reliability in case we make a logic error in the code or a bad test.

9.1 Unit testing

Unit testing is a method to create individual tests, only related to a single unit or method. These tests will validate the implementation to see if it performs as expected by returning whether the test failed or passed under the given circumstance, and unit tests will pinpoint any flaws in the individual execution. Furthermore, unit tests can allow us to test if we can read or write to our MongoDB database.

The purpose of unit testing is to provide the developers insight and knowledge about how a method/unit should behave and which results it should provide in a controlled environment. If the developer knows what to expect if the method/unit were to execute properly, the developer is able to write tests to see how certain changes to code affect the outcome of these tests. This method can be a form of white box testing, which is used to test the internal structure of the code [11].

Writing unit tests also includes writing standalone code tasks where each test should only have one clear objective. At the end of each unit test, we are going to use various assert methods if the expected result is equal to the actual result.

We chose to write unit tests because it is a reliable way to ensure that the methods are returning the correct results. Each unit test is implemented for only one method, because if we write multiple methods in the same test, it can change the focus of the test and can possibly skew the test result. As an example, if we test methods from both repair and write-off at the same time it is no longer a standalone test or a unit test.

The benefit of unit testing is catching errors that would not necessarily be detected during program execution. A lot of small errors might not result in the system crashing, but rather give unexpected results and result in inconsistencies. For instance, a repair might be created without getting a technician assigned, which would result in a discrepancy between problem domain objects and the system model.

If methods being tested are later changed in some way, it will make the test fail, and the test could therefore be unusable, which requires a revision of the test. Rewriting unusable test code would eventually make all of the tests operational, leading to re-testing the methods again, and all test cases will pass.

9.1.1 JUnit

Junit is a Java unit testing framework which is used to write and run tests that are repeatable. JUnit has a fourth and fifth version, which are almost the same, but have rarely differences with annotations when using one over the other [12]. JUnit has these features that are described below:

- It finds bugs early in the code, which makes our code more reliable.
- Unit testing forces a developer to read code more than writing.
- JUnit is useful for developers who work in a test-driven environment.
- Developers develop more readable, reliable, and bug-free code which builds confidence during development [12].

In the project's development, we chose to work with Junit5 since JUnit 4 and 5 would not make a difference in terms of features. In Junit, we will write a springBootTest which loads the complete Spring application context. In a spring boot test, we have to include the repositories for our products, spare parts, write off and so on to help make the tests, which is why we use the annotation AutoWired to inject the repositories that we will be working within the test implementations. Before testing we have a beforeEach to indicate it should be executed before all other tests.

9.1.2 Repair test

For our repair test, we have tested all the methods in the repair service. Repair service as described in the 8 provides our repair controller with methods in an interface where the methods are going to be implemented in repair service implementation. It is responsible for implementing the logic to the methods.
In a general perspective, we need the repair, product, and spare parts repositories to get and set the stage of each product, repair, and the spare part where database handling is essential for the tests. The repair test has a default product used when conducting the tests. When testing our system, products, spare parts, and repairs, we mainly switch states to see if the repositories work as intended with the MongoDB database.

Create repair test

Creating a repair has two needed instances, which are a product and the technician. The condition for repairing the product is that it needs to be **defective** and that any employee from Blue City is required to have at least a technician privilege to qualify. Making the user and how to give the technicians privileges is described below.

```
1     // Making a user and set its privilege
2     User user = new User("Ming", "password");
3     user.setUserPrivilege(UserPrivilege.SEMI_ACCESS);
```

After a user has been created, a default product is saved in the product repository. Creating a repair is coming from the repair service and requires to get the specific product id and the username from the technician. At last, to identify if the product is in repair state, using assertEquals method allows us to find out the product's state by writing the expected and the actual result. The result of the test can be found below.

```
1     @Test
2     public void whenRepairIsCreated_ProductIsUnderRepair() {
3     //Get back the saved product to retrieve ID
4         Product savedProduct = productRepository.save(iphone);
5
6         // Service under test for creating repair
7         Repair updateRepair = repairService.createRepair(savedProduct.getId(), user.
      getUsername());
8
9         assertEquals(ProductState.IN_REPAIR, updateRepair.getProduct().getState());
10    }
```

Listing 8: Repair test

Pause and resume repair test

By making a repair from the repair class with a selected product and saving it in the repair repository, we can now switch states for the repair. In this test, the purpose is to pause the repair, so the technician can notify the manager it is not finished and will not be in progress at the moment. Using the pause repair method will set the state of the repair to paused, so nothing cannot be changed in the repair until the technician resumes the repair. The method also set a date for when it was paused and will save the repair state.

The result of the resume test is to take the specific product entity and set its state from pause to resume. This lets the technician continue his work where he left off. The pause method and the resume method, which both are methods in the repairSerivce interface, will execute accordingly to fulfil the expected result. The test structure is shown in the listing below.

```
Repair repair = new Repair(iphone);
    @Test
    public void whenResumeRepair_ExpectRepairToBeResumed() {
        //Get back the saved repair to retreive ID
        Repair savedRepair = repairRepository.save(repair);

        // Repair is now PAUSED
        repairService.pauseRepair(savedRepair.getId());

        //Service under test for repair
        Repair updatedRepairToResume = repairService.resumeRepair(savedRepair.getId());

        //The repair is now RESUMED
        assertEquals(RepairState.ON_GOING, updatedRepairToResume.getState());
    }
```

Listing 9: Pause and resume repair test

Finish and cancel repair

A technician can either finish or cancel an ongoing repair. The two tests will finish and cancelling a repair is the last step that is written in the Section 8.1.8. The code steps are the same as resume and pause a repair. The only two difference are to test a different method from the repair implementation and the expected result will be also be different.
The test result for a finished repair will set a repair's state to finished, and the product will be set to repair after being set from in repair. The test of cancelling a repair will only change the product's state back to defective and the repair will simply be deleted after using the cancel repair method. The test code is written below to simplify how cancel and finish repair is structured and differs from each other.

```

        //Repair should be finished
        assertEquals(RepairState.FINISHED, updatedRepair.getState());

        //Product should be repaired
        assertEquals(ProductState.REPAIRED, updatedRepair.getProduct().getState());
```

Listing 10: Finish repair test

As for the cancel repair test, it should simply have the product change its state to **defective** with a "assertEquals" as well. The added spare part from the repair would change its state from **RESERVED** to **AVAILABLE**.

Add and remove a spare part to a repair

The add and remove a spare part test involves a repair while it is in progress, where technicians can add or remove spare parts to it. This test aims to change a spare part's state and find out if a spare part is added or removed from the repair.
Testing requires a new spare part to work with a specific product and repair. The spare part test does include the previous element of creating a **new repair** and saving both repair and spare part in their repositories. Testing the add spare part method should change the part's state and allow us to check with assertEquals if the result is correct. The test structure is shown below:

```
1     @Test
2     public void whenSparePartIsAddedToRepair_SparePartIsReserved() {
3         SparePart battery = new NewSparePart("Apple",Category.IPHONE,"11 Pro", Grade.A,
      SparePartType.BATTERY, 250, "23124124");
4         Repair repair = new Repair(iphone);
5
6         //Get back the saved repair to retrieve ID
7         Repair savedRepair = repairRepository.save(repair);
8         SparePart savedSP = sparePartRepository.save(battery);
9
10        //Service under test for adding spare part
11        SparePart updatedPart = repairService.addSparePart(savedRepair.getId(), savedSP.
      getPart_id());
12
13        assertEquals(SparePartState.RESERVED, updatedPart.getState());
14    }
```

Listing 11: Add spare part test

The test for removing a spare part is similar to how adding a spare part is tested, except for the method in line 11 and the expected value in line 13 of listing 11.

Illegal Repair Operation Exception test

Suppose a technician tries to finish a repair when it is paused. Since this is not allowed in our system, the test should clarify if an exception, called "IllegalRepairOperationException", will be thrown to counter this action, thereby preventing the program from executing the illegal command.
The way to test the exception is to create a repair, finish it and asserting a throw operation to call back the pause repair method, where an exception is thrown to prevent unexpected behaviour from the users. The code itself is the same as the previous ones, but the assert methods are different, as seen below.

```
1  //The repair cannot be paused when it is finished
2         assertThrows(IllegalRepairOperationException.class,
3                 () -> repairService.pauseRepair(savedRepair.getId()));
```

Listing 12: Repair exception test

9.1.3 Write off test

The purpose of Write off tests is to make a write-off ticket with some of the spare parts inside the product as we expect them to be still functional. The technicians perform the write-off ticket process.
After making a ticket, it is supposed to be the manager's turn to approve or disapprove the write-off, depending on whether he thinks it is reasonable to scrap the product. Therefore two more tests are the approve and disapprove write-off ticket test.

Create write-off test

Creating a write-off ticket is done by using the create write-off ticket method, which requires a product and a person who initiate the write-off. Three objects are being affected by the method, by which each of them is going to change state: the product, write-off ticket, and the spare part included in the product. There is a initial product for each time we are going to test write-off.

The test results should only asserts the expected results for each of these objects, which should be in their new states, which is **IN WRITE-OFF** for the product, **AWAITING** for write off ticket, and **MARKED FUNCTIONAL** for spare parts.

```
1  @BeforeEach
2      void setup() {
3          testProduct = new Product("20004000", "Apple", Category.IPHONE, "11 Pro", "128 gb
       White", 4500, 2000);
4          testProduct = pRepository.save(testProduct);
5
6          fakeForm = new WriteOffTicketForm();
7          fakeForm.setReason("Product under test");
8          fakeForm.addMarkedParts(Arrays.asList("SCREEN", "BATTERY"));
9
10     }
11
12     @Test
13     void testStatesWhenCreatingWriteoff() {
14         testTicket = woService.createWriteOffTicket(fakeForm, testProduct.getId(), "Bertan");
15         assertEquals(ProductState.IN_WRITEOFF, testTicket.getProduct().getState());
16         assertEquals(WriteOffTicketState.AWAITING, testTicket.getState());
17         assertEquals(SparePartState.MARKED_FUNCTIONAL, testTicket.getSpareParts().get(0).
       getState());
18     }
```

Listing 13: Create write off test

Approve and disapprove write-off test

Approve and disapprove write off In this test will first require the same beforEeach inputs from listing 13.
The test should follow the step with; creating a new write-off ticket The test should mark usable spare parts that can be used for other future products.
Approving a write-off will get the ticket that has been created and retrieve spare parts' and a product's id from the database described in 8.1.2, and the spare parts' state will be changed to **AVAILABLE** and the product will change its state to **WRITTEN-OFF**.

Disapprove write-off test will only have to set the written-off product back to defective. The test will follow up by finding the product's new state, and we expect the test is showing that the product is now **DEFECTIVE** again.

```
1     //Retrieve part and product after write off approval from db to test states
2     SparePart part = spRepository.findById(testTicket.getSpareParts().get(0).getPart_id()).get
      ();
3     Product product = pRepository.findById(testTicket.getProduct().getId()).get();
```

Listing 14: Create write off

9.1.4 Spare part test

The spare part test's purpose is to find compatible spare parts from a repair, get the list of spare parts in the database, and upload a spare part. The test will follow the previous tests' principle of creating a write-off ticket, but assertEquals has changed from the previous tests.

Compatible spare part test

The test should find compatible spare parts for a specific repair. The first step for the test is to create a product and then get a specific used spare part and save that in the spare part repository. The repair will now be created for the product and get recommended spare parts, and we can now test if a single spare part is compatible with the product. In this test, we would like to have an iPhone 11 pro battery. The test asserts that we getting the brand **Apple**, the category **iPhone** and the model **11 Pro**. The method of asserting the result can be seen below.

```
1 /* It is sufficient to test only that a single spare part is compatible with the product since
       they share the same query */
2          assertEquals("APPLE " + Category.IPHONE + " 11 PRO", spareParts.get(0).getBrand() + "
     " + spareParts.get(0).getCategory() + " "  + spareParts.get(0).getModel());
```

Listing 15: Finding compatible spare part for repair

9.2 Result of unit tests

Along the way, there were some logical errors and internal errors while writing the tests. Several errors happen when getting the specific data which were either to repair, write off and fetch the data from the database was not possible, which caused the test to return an internal error. Parameters from the create repair method required an object that was not needed where it has been discovered by continually testing it.
The hardest part of the test was still the logical errors that caused it to choose the wrong entity element in the database since cost prices for every product did not update when testing them. We managed to fix them with Junit's help to indicate which and where the errors were and handle them.

After running the tests on the different parts with unit testing and being thoroughly reviewed, we concluded that all tests above come out as a success where no error is detected throughout. The below screenshots can be seen in 9.1, 9.2, 9.3 and this is taken after we have run the tests.

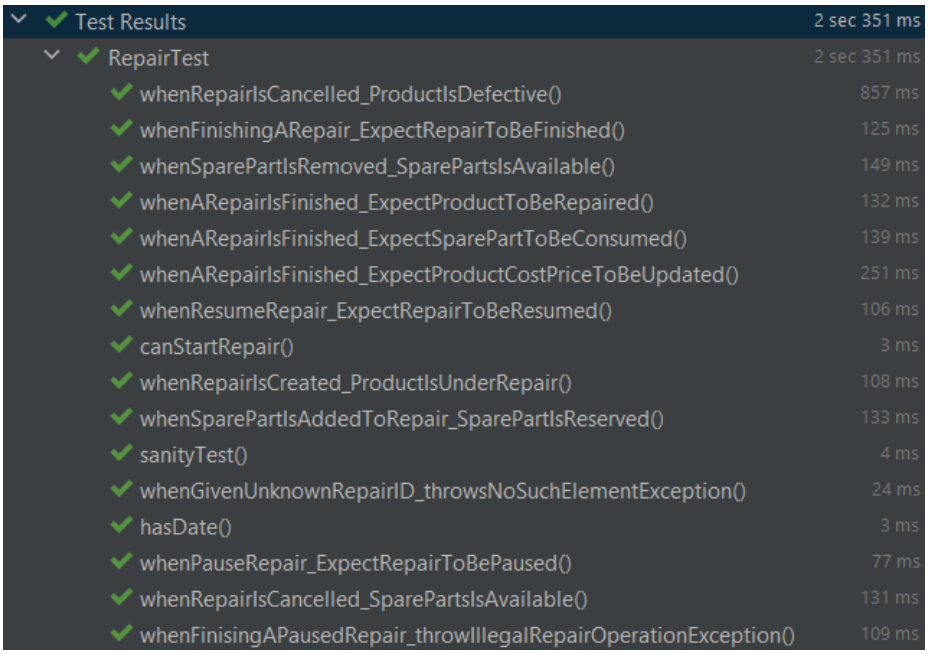


Figure 9.1: End results of Repair test from Intellij with Junit

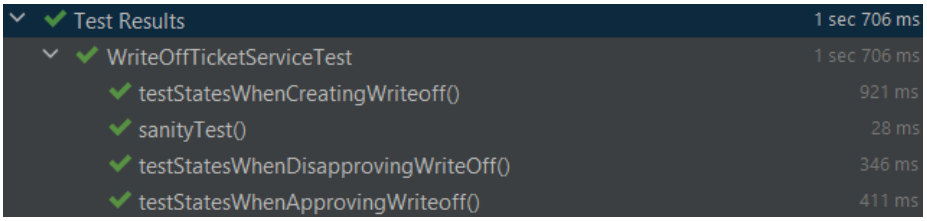


Figure 9.2: End results of Write-off test from Intellij with Junit

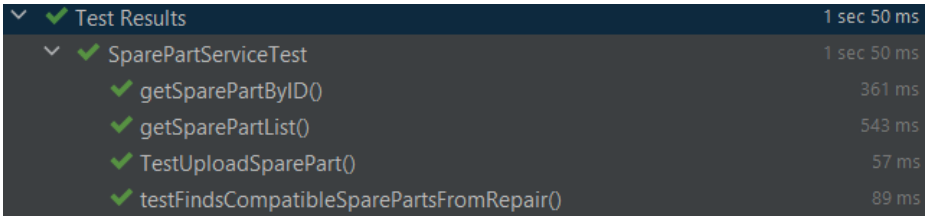


Figure 9.3: End results of Spare part test from Intellij with Junit

9.3 Usability test

The purpose of our usability test is mainly to be aligned with what our users is expecting of the system after a lot of work have gone into implementation. The overall goal is to:

- Assess the effectiveness of the system for both the technicians and the managers performing basic tasks
- Identify usability problems to gather valuable feedback for further development

When planning the usability test, there is two main questions that we have to ask our selves: **Who** will be our test persons and **what** part of the system is being tested. The first two chapters will be answering these questions. The last chapter will be dedicated to any problems identified during the tests and the feedback we got.

Picking participants

Having found that we have two actors in the system it is pretty clear that we need two different kinds of participants to successfully test the system. A participant that takes the role of a manager and another one as a technician. Due to the small size of the workshop team, we knew that we were very limited by the participants who were able to participate in the test. As a result, we were only able to reach out to two participants, the manager and one of the technicians.
The manager participant can be characterised by having a broad understanding of the systems context, while missing the technical depth that a technician might have. Therefore, we do not expect him not to fully appreciate the parts of the system that are meant for the technician. He is, despite his manager role, involved in what goes on when a technician makes repairs and therefore we also had him participate as the role of a technician.
The technician who agreed to participate in the test can be characterised as highly skilled and highly involved in the repair process. This test-person had no knowledge about the system prior to the test, and had not been involved in the development process.

Test methodology

The usability study will have the following outline.

- **Introduction (2 min):** Here we introduce the purpose, objective, test managers role, observers role, recording of the session and think aloud.
- **Task session (15 min):** This is the session where the participant will be performing specific tasks. Depending on the participants role, the time of the session will vary.
- **Feedback (10 min):** Here we ask questions to follow up on the test and discuss any of the problems they encountered.

Task list

Our task list is split up into two test modules. Module A for technician and module B for the manager. Each task is designed after one use-case and is made to only give information on a high level. The individual steps leading to completion of a task, were left out and needed to be filled out by the users intuition and understanding of the system. For each task we provide the scenario in where this task appears in the broader system context.

Module	Test	Scenario	Task
A	Login as technician	Given credentials, you want to login to the system	Login to the system
	Make repair	You are working as an iPhone technician and want to make repairs in this category	Start a repair on the product with the highest profit margin. Add the necessary spare-parts to complete the repair
	Make write-off	A laptop can not be repaired and needs to be lossed. From the inspection test you know the keyboard and screen is still functional	Write-off the laptop and mark the functional parts
B	Login as manager	You want to login to the system. Your username is 'Bertan' with password 'password'	Login to the system with your given credentials
	Register a new technician	You have hired David as a new technician in the workshop and want to register him into the system	Register David as a new user with technician privileges
	Approve write-off	A technician is attempting to make a write-off on a product where the battery is still functional	Find the write-off from the technician and approve it
	Register a new product	A new defective product have entered the workshop and you want to register it	Find a product in the workshop and register it into the system

Table 9: Table of the tasks that were performed

Data collection

For data collection, we will measure the time it takes to complete each task. Here we were aware of the fact that measuring time when also wanting the participant to think aloud would affect the accuracy of the results. The table right below shows the results of the test.

Module	A			B			
Test	Login as technician	Make repair	Make write-off	Login as manager	Register a new technician	Approve write-off	Register a new product
Performance (sec)							
P1	10	240	60	10	130	20	120
P2	<20	<20	<20				

Table 10: Performance summary

Participant P1 is the manager playing both the role of the technician and the manager. Participant P2 is the technician.
All the tasks performed by P2 were done in under 1 minute each, with no note worthy frustration, indicating a very pleasant and easy workflow.

P1 spend notably longer time on the tasks in module A when compared to P2. The reason might be that P1 is the manager in the workshop and does not have the same domain knowledge when performing technician tasks.

Identify problems

During the task "*Register a new technician*" P1 had a lot of trouble finding the manager page where the user registration is located. After finding it under the profile menu, he admitted that the location chosen was the most intuitive location and should not be changed.

Another problem occurred in the task "*Register a new product*" performed by P1. He would go and search for this feature under the "Products" tab and missed the tab named "Add products". The task came to a full stop when he filled out the product form and pressed "submit" as the system rejected due to an error when selecting a product category.

The last major problem was encountered in the task "*Make repair*" where the system would not allow P1 to view a specific repair. This resulted in the test manager having to step in to fix a minor URL problem to make it work.

Follow up questions

In the last part of the test we asked about what they like and what they would like see changed in the future. These were the answers P1 and P2 gave after the task session was over.
What was the best thing about the system?

Only manager can approve write off's
Extremely intuitive site
Ease of use and easy to navigate. A much better version than our current solution

What changes do you desire?

Missing repair history, in order to see if the same issue persists within the same product, such that a customer doesn't come back again and again asking the same thing to be fixed
Need to be able to filter the repairs and products by their current state, such that it only shows products with the selected state
Needs a search bar to be able to search after certain Id numbers and other details
Makes little sense to have an initial state of "unassigned" before becoming a technician
Be able to see cost of all spare parts
A notification to the manager upon log in to show write offs awaiting approval
Show the grade on each spare part on the repairing page
When a repair is finished, there should be an option to "add finished comment". This would allow the technician to specify what have been replaced repaired. Moreover the technician could note if something was not fixable or not covered by warranty.
When finishing a repair, there should be an email sent directly to the customer to inform them of the repair status, and when they can pick it up (relates to street repairs and reclamations)

Conclusion

To conclude, in general they were very happy with the system that we have developed, and were excited for the potential possibilities of adopting a new and improved system. Though, there were a lot of features that Blue City felt, could improve the system and thus their efficiency if the system was to be adopted. There is a solid foundation, with a lot of room for further development as indicated by the list of desired changes. Concluding anything on the measure of performance is hard, as we would need a lot more participants to get an accurate picture on the effectiveness of the system.

10 Discussion

In this section we evaluate on the overall success of our system by evaluating requirements. Furthermore, we will look at some changes and discuss further development that could be made to improve the system.

10.1 Requirements Fulfilment

When evaluating the overall success of the system we revisit the requirements from Section 5. These requirements stem from our problem domain analysis and are categorised according to the MosCoW model Section 3 as well as our interviews with Blue City. There were no initial requirements set by Blue City, as their current system was deprecated and any new system would be an improvement, however as the project came to fruition, they had several ideas, wishes, and wants for the system. The table below shows our "must have" requirements and the fulfilment degree for each of them.

Requirement	**Priority**	**Fulfilled**
Recommend which product is most profitable to repair	Must have	Partial
Handle more than one user at a	Must have	Yes
Register a product as undergoing repair	Must have	Yes
Allow pausing, resuming, cancelling and finishing a repair	Must have	Yes
Keep track of time spent on a repair	Must have	Yes
Must have a way to add spare parts to a product under repair	Must have	Yes
Must have a way to remove spare parts from a product under repair	Must have	Yes
Allow products to be written off	Must have	Yes
Must be able to register functional spare parts within written off products	Must have	Yes
The cost of a written off product must be split between the spare parts functional	Must have	Yes
Used spare parts must be able to be traced back to the product it came from	Must have	Yes
The cost of spare parts used during a repair must be added to the product cost	Must have	Yes
Users must be able to find products, repairs, and spare parts	Must have	Yes
Provide statistics about repairs	Must have	Partial

Table 11: Must have requirements

Recommend which product is most profitable to repair

The reason for the partial fulfilment of this requirement is based on the approach technicians take to select which product to start repairing. During an interview, the technician explained how Blue City's upper management would provide specific workshop metrics which needed to be improved within a given time frame. These metrics could, for example, be "average storage time must be lowered" or "average cost price for defective products is too high," the technicians would select products to repair which could achieve these goals. The solution implemented in the system partially fulfils the requirement. Instead, it fully satisfies the requirement "provide an option to sort items," as the technicians can sort the products on the overview page by cost price, sales price, storage time, and profit margin. This solution was proven to be satisfactory to Blue City as well, as the technician effortlessly sorted the products and selected the correct one during the "start a repair" task in our usability test 9.3.

Provide statistics about repairs for the manager

The system currently provides information about how many repairs the workshop has in total, ongoing, paused, and finished. Ideally, the system should also convey the average repair time for products per category, the number of repairs finished today, the number of repairs finished this week, the number of repairs finished per technician, and the average repair cost.

Fulfilment conclusion

As table 11 illustrates, the system fully satisfies 12 out of 14 of our "must have" requirements and partially satisfies the last two. In addition to the must have requirements, we also fulfil or partially fulfil 9 out of 11 of our should have requirements. The unfulfilled "should have" requirements pertain to the searching and order list requirements for the most part. The order list was originally included as a requirement based on an interview with Blue City where they expressed interest in the functionality, however, due to time constraints, we were forced to re-scope the system and focus our efforts on the repair process. The requirements regarding search functionality are largely satisfied by how information is presented to the users. Instead of searching for spare parts during a repair, the technicians are automatically presented with a list of parts compatible with the product being repaired. Instead of searching for repairs, the technicians can choose to view repairs with a particular state. Overall the system satisfies a sufficient amount of our must have and should have requirements to be considered a success.

10.2 Further development

Our application is complete to the extent of our outlined limitations and requirements, but we could add many new functions to improve the application and provide a better user experience. Create repair method lets a technician start a repair with a specific product, but the he/she cannot edit the repair description before and after starting a repair which also follows in the detail page entirely. As for the search engine, it can only input a string which is the name of the product and could therefore be better to search more than a string. The groups and the user's further development are discussed more in-depth below.

10.2.1 Groups points for further development

Conforming to REST API guidelines

One of the things, that could be improved upon in this projects implementation is to conform more closely to some of the REST API best practices, i.e those we used, defined by Microsoft [13]. As our back end is developed using the Spring boot framework with the REST package as a dependency, we have sought to adopt as much of the best practices regarding REST, into the project as possible. If we were to develop the system further, we could improve the system by adopting or improving the implementation of following areas:

- Defining API operations to accommodate HTTP methods
- HATEOAS for related resource navigation
- Resource Pagination

Most of the our systems endpoints are defined according to the strategy presented in the REST architecture article by Microsoft:

Resource	POST	GET	PUT	DELETE
/customers	Create a new customer	Retrieve all customers	Bulk update of customers	Remove all customers
/customers/1	Error	Retrieve the details for customer 1	Update the details of customer 1 if it exists	Remove customer 1
/customers /1/orders	Create a new order for customer 1	Retrieve all orders for customer 1	Bulk update of orders for customer 1	Remove all orders for customer 1

Figure 10.1: Example of endpoint design by Microsoft

But some of the endpoints like this post route: */writeoffs/create*, could be changed to */writeoffs*, since it is implied that when you send a POST request to this endpoint, it is to create a new write-off ticket, thus rendering the /create obsolete. Having these conventions in mind, would help the developers, ensure uniformity across the system.

HATEOAS: Adopting a *Hypermedia as the Engine of Application State* (HATEOAS) [14] strategy could be become useful for navigating between related resources, as a lot of the different classes have relations to each other to some extent. Currently, the systems uses the ReponseEntity interface, to send information about other resources back to the front-end, to automatically navigate to the page that requires this information. This strategy is not essential for the system, but could make the front end's interaction with the back end less prone to errors.

Pagination: The practice that would have the most significant impact on the performance of the system, would be to paginate the API resources. In the current implementation, the database stores approximately 500 products, sending a GET request to the products endpoint returns all 500 products in a single JSON response. Querying an endpoint for a resource of that size impacts the request-response time. A quick network test with the Chrome's network inspection tab, shows that the request time could take as long as 2.8 seconds. As the user is not looking at 500 products at once anyway, it would provide a better user experience to limit the amount of products for each request. By implementing pagination, the system would limit the number of products requested to a specific size limit, and thus decrease the time it takes for the API to handle a request.

Microsoft's REST guideline [13] gives an example of how to implement this strategy, with the use of two Request Parameters: *Limit* and *Offset*, ex:

/orders?limit=25&offset=50

- **limit**: the specified size, of how many items the response returns
- **offset**: used to specify which batch of items, should be returns. offset set to 0, would return items[0...25], and offset set to 50, would return items[50...75]

10.2.2 Blue City's notes for further development

Our usability test provided us a lot of feedback which will be discussed in the following subsections. This part should give insight into how we could further develop the solution based on the feedback we received.

Order list implementation

Another point from the usability test is not having a dedicated order list. Without an order page for the manager, he has no way to track how many spare parts are ordered. We had originally planned to implement an order list, which would include a list of spare part types as descriptors and would create spare part objects with a state suggesting they had been ordered or was en-route. This functionality is something Blue City would be highly interested in, should we develop the system further.

Repair history

Another point of improvement for the system we got from the employees at the workshop is to implement a repair history view to see if the same issue persists within the same product, such that a customer does not come back again asking for the same thing to be fixed.

Since we already store all information related to a repair in the database, even after a repair has been finished, it would be possible for the technician to search for a repair with the same product in the application and read the repair comments and remember them. This would work, but this solution requires more from the user, and it would not be problematic to implement functionality that would make this task significantly easier for the technician.

One solution could be to allow the technicians to add IMEI/Serials number when adding new products and then, before creating a new product on the back-end application, check if a product with an identical IMEI/Serial number exists, and if it does, redirect the users to its product detail page in the front-end application. From there, with some minor changes to how the system checks if a product is eligible for repair, the technician would be able to start a repair on said product.

Now that the technician has started a repair, only two things are missing to display the products repair history. Firstly, the front-end would require an endpoint to query the system for all repairs related to the product. Secondly, we would have to build a React component to present the results to technicians, perhaps sorting based on the end date to get a chronological overview. With that said, it is our belief that this makes for an excellent feature if the system were to be developed further.

Location management

One problem the blue city has right now is that technicians have trouble physically locating the products recommended by their current system to be repaired. This is because they have to look through hundreds of products in a very big pile and check each product number if it matches the recommended product. To solve this issue, a product location could be used to keep track of the position when a new product enters the workshop. This could be facilitated physically in the workshop by labelling boxes with locations where the products would be put. Then, technicians who are about to start a repair would be able to quickly go to the product's location and start working on it. The idea of location on products had always been in the back of our heads during development. Still, to work in practice, it would require the blue city workshop to reorganise a good portion of their products, and we did not want to intervene too much on that part. As so, we put it up for a future development feature.

Description of repairs

The last piece of feedback from the usability test regarding repairs was a repair description. During the usability test, the test subject expressed that a repair description is needed by the technician, in order to document the repair process. Documenting the repair process could include what parts were replaced or repaired or a note if the issue was not fixable.

Email notification upon repair completion

One point of feedback from a technician was for an email to be automatically sent to the customer when a repair is finished on their product. This pertains to two types of repairs, street repairs and reclamations, which our application does not handle. For security reasons, our system currently has no direct interface with Blue City's main system and, therefore, no way to receive information about customers. We could implement a way for the technician to enter customer information, but the lack of this proposed emailing functionality is not actually an issue that exists in their Priotool, which our system aims to be an improvement to, and therefore it lies outside of our project's immediate scope to implement such a feature.

Overview of available spare parts with quantity.

During our usability testing, the manager and the technicians provided us with useful feedback on how they thought the system could be optimised to fit their needs. One of the things that the manager sought was the ability to have an overview of the spare part inventory, and more specifically, see the different spare parts grouped with a given quantity as opposed to the more singular object view. This would improve the manager's ability to gain a quick overview of the spare part inventory and be especially helpful when the manager must decide on which parts they need to order for the workshop.

Ability to search for product id numbers

Currently, the product search functionality implementation only supports searching for string matching the product class variable name. The manager and technician pointed out, during the usability testing, that it would be of great use to them if it was also possible to search by product id. This is a valid point since the technicians often use bar-code scanners for inputting product data into various applications. To accommodate this request, we could provide this functionality by either creating a separate search bar that takes product ids as inputs, or the more elegant way, using regex to check if the search value provided matches with a product id, which is always a 12-digit numerical value. Because the internal Blue City product ids have this specific format, the product id would not collide with the alpha-numerical value stored in the product name class variable.

Initial State of unassigned

From our usability testing, the Blue City Manager also noted that it made little to no sense of having the initial state of a new user set to unassigned. Instead, the manager wanted to set the new users state to a technician. This was valuable feedback and pointed out a flaw in the way we were thinking about the users. Naturally, the new technicians should not enter the system as if they have resigned from the position even before starting. A user is associated with a repair, such that we can keep track of which technician is working on which repair. This same user should be responsible for finishing the repair and his credentials would follow the repair history. Therefore, we can not simply delete the users from the system, we introduced the "unassigned" state for the user. It would be relatively easy to change this oversight by changing the initial state of new users in the back-

end user controller. Having thought of this, it also makes little sense to display unassigned users in the admin section of the front-end application, this could easily be solved by only rendering the user information based on the condition that a user does not have the state of unassigned.

Notifications: Manager gets notified of Write-off tickets after login.

Another requested feature we extracted during the usability test is that the Workshop Manager would like to be notified if they had unseen write-off tickets after they logged in. This would fall in the "could have"-category compared to the feature requests. This statement could be disputed because they specifically said that this would be an improvement to the application, but amongst the other improvements, this is less significant regarding the overall usability of the system. With that said, the task of implementing this feature is not the biggest challenge. This notification could be easily implemented by creating an endpoint in the back-end that only returns unhandled write-off tickets. Querying that in a small React component, which is rendered in the front-end Menu bar since the Database system (MongoDB) supports querying the length of the specified collection.

11 Conclusion

This project aimed to develop a system for Blue City that would help the employees run their repair workshop more efficiently. This section will describe how and to what degree we managed to do that.

First, we had to understand their problem. We did this by conducting semi-structured interviews with their manager and a technician. We recorded the first interview and later transcribed the information. This gave us an accurate description of their problem and resulting requirements in their own words. Hereafter, we used a rich picture to visualise the problem field to all members of our project group, ensuring a common understanding. We were now able to formulate the following problem statement encapsulating the problem:

> *How do we make a system which increases the productivity of Blue City's technicians and improves the repair flow of their workshop, lowering repair times and potentially increasing profits.*

To begin solving this problem, we applied methods presented in the Object Oriented Analysis & Design course [1]. By making class diagrams, event tables, and behavioural diagrams, we were able to attain a detailed understanding of how all of the different objects in the problem domain behave individually and as parts of a whole. After this, performing the application domain analysis of actors and use cases allowed us to determine how users interact with the system. From this, we were able to define the functions that would be needed in order to fulfil the requirements of the system.

To make these functions available to the users, we then designed a user interface, applying concepts from the Design and Evaluation of User Interfaces course [5]. We used these to make a UI that would meet the non-functional, mainly usability, requirements of the system established in our architectural design analysis. Through the use and showcasing of prototypes, we were able to keep the UI design in line with the users' expectations.

By having regular meetings with Blue City throughout the system development, we managed to continually verify that our implementation aligned with our clients' requirements. Regarding meeting the requirements in our MoSCoW table, we have implemented all of our must-haves to an extent, and all of our should-haves except for two due to time constraints. No could-haves or would-haves were implemented.

We employed unit testing throughout the implementation process to ensure the correctness of our system, thus fulfilling another non-functional requirement of correctness. Additionally, a usability test with Blue City showed a high degree of satisfaction with the UI and functionality of the application, with great interest in potentially adopting the system in the future. However, before they were willing to adopt the our system, several features needed to be implemented. The test revealed a few requirements we had yet to implement, mainly because we had not become aware of them through the interviews and showcase of prototypes. That we performed during the design and implementation phases, thereby eliciting several tacit requirements.

This puts our application in a functional, close to complete, state that leaves a few features yet to be implemented before reaching optimal functionality required for Blue City to adopt it.

To conclude, we have developed a system that manages products, spare parts, repairs, write-offs, and technicians, and accurately updates factors like storage time and cost price. The system is an improvement to Blue City's current system in that it supports multiple concurrent users, enables quicker acquisition of spare parts,

potentially resulting in lower repair times, and provides a more accurate stock value of written off products. Therefore we have managed to devise a system that is capable of potentially improving the workflow in the Blue City repair shop, with potential interest from Blue City of adopting our system.

12 Development Model

In this project we have followed a Software Development Model that can best be described as a Waterfall model with iterative elements to it. We have worked from a very stable foundation of requirements, but have iterated over our design and implementation repeatedly. This was necessary when talks with our clients required us to make changes, but also in great part due to the fact that we were learning the OOA&D method while we were using them for our project. This meant that every time we had misunderstood some part of the theory and thus implemented something incorrectly, we would have to redo the process to remedy our mistakes. This proved to be quite time consuming, but it is possible that we put too much effort into performing the activities.

It was difficult to know how much effort we should put into an activity before it could be considered sufficiently completed and we should move on to focus on the next activity. For example, in the functions activity it was difficult to describe the complexity of functions accurately before actually implementing them. At other times, activities from OOA&D were difficult to perform because we had to design something without knowing exactly the tools, frameworks, and modules we could use in the implementation. This was especially the case during the model component and function component activities. We started performing them but did not feel we made any progress towards better understanding what our implementation should look like. Therefore we resolved to begin implementation without completing these activities. Our implementation was thus based on both the outcome of the earlier activities and a more iterative development style in which we continually modified our code to fit the needs of our system. Specifically, we moved a lot of functionality from our classes into a function component when we decided this was a better system structure.

In conclusion, what we have learned from working with the OOA&D method can be summarised in the following to paragraphs:

Eliciting a comprehensive list of requirements early on in a software development process provides a great starting point, but the chance that new requirements arise or that changes to existing ones are made during the process is often guaranteed. Therefore applying the OOA&D method to a non-iterative method like Waterfall, requires that some degree of flexibility is taken into consideration when planning the process.

On the positive side the OOA&D method provides a solid foundation in terms of establishing necessary classes and functions before starting implementation. The trick is not to spend too much time on the activities, but we believe this is also dependant on developer experience. Were we to use OOA&D again in a future project we would undoubtedly complete the activities with greater efficiency.

References

1. P. A. N. J. Lars Mathiassen Andread Munk-Madsen, *OBJECT ORIENTED ANALYSIS & DESIGN* (Metodica ApS, Lyngbakken 20 - 9560 Hadsund - Denmark, 2000).
2. *Figma: the collaborative interface design tool.* en-US, (2021; https://www.figma.com/).
3. *What is MoSCoW Prioritization? Overview of the MoSCoW Method*, en-US, (2021; https://www.productplan.com/glossary/moscow-prioritization/).
4. E. Wong, *User Interface Design Guidelines: 10 Rules of Thumb*, The Interaction Design Foundation, (2021; https://www.interaction-design.org/literature/article/user-interface-design-guidelines-10-rules-of-thumb).
5. D. BENYON, *DESIGNING USER EXPERIENCE: A guide to HCI, UX and interaction design* (Pearson, 2019).
6. *Gestalt Laws: Laws of Proximity and Similarity*, (2021; https://isle.hanover.edu/Ch05Object/Ch05ProxSim_evt.html).
7. P. Martinez, *What is Software usability and how to do it succesfully*, en, (2021; https://mockitt.wondershare.com/ui-ux-design/software-usability.html).
8. *Spring boot*, (2021; https://spring.io/projects/spring-boot).
9. *MultipartFile (Spring Framework 5.3.14 API)*, (2021; https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/multipart/MultipartFile.html).
10. *InputStreamReader (Java Platform SE 7)*, (2021; https://docs.oracle.com/javase/7/docs/api/java/io/InputStreamReader.html).
11. D. Chacon, *JUnit Tutorial With Examples: Setting Up, Writing, and Running Java Unit Tests*, Aug. 2018, (2021; https://www.parasoft.com/blog/junit-tutorial-setting-up-writing-and-running-java-unit-tests/).
12. T. Hamilton, *JUnit Tutorial for Beginners: Learn in 3 Days*, en-US, Section: JUnit, (2021; https://www.guru99.com/junit-tutorial.html).
13. EdPrice-MSFT, *Web API design best practices - Azure Architecture Center*, en-us, (2021; https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design).
14. *HATEOAS Driven REST APIs*, en-US, June 2018, (2021; https://restfulapi.net/hateoas/).

13 Appendices

A Semi-structured interview with Blue City

1. Which challenges regarding repairs of Blue City's own products are you experiencing with your current system?
   (a) Can you give a concrete example of a problem that occurs using your current system?
   (b) Are you aware of any challenges your colleagues are having with your current system?
2. What is your current system capable of?
   (a) What must an optimal system be able to do?
3. What is the process for a product from the time it is bought or declared defective to when it is repaired?
4. Why do you chose one product ahead of another when starting a repair? How do you chose which product to start repairing?
   (a) Which factors are part of this decision process?
5. Where in the process do you think is the greatest potential for improvement?
6. How often would you like for us to meet with you?

Transcript of the interview

Sugam being the interviewee, and Marcus, Hans, Frederik being the interviewers

Markus: Okay
Markus: first off we would like to get a general idea of what kind of system you have right now, when it comes to your repairs of the products that blue city already owns. So anything you have to stock that is defective, that needs to be repaired.
Markus: Uh,
Markus: uh, what kind of challenges do you have with the current system.
Sugam: The current system, like we have a lot of challenges. So let me start with the system, basically from what we have, then we can go forward with the challenges. Yep. Perfect. So right now we have one Prio-tool that we only technician use at the moment to update our, all this repair data. And then we have this blue system that is access to all the store to.
Sugam: Update information about all the product from buying to selling. But these two doesn't have any link with our repair data. So whatever has been updated on Prio-tool no one can see on the blue system. So that means that whatever we do, whatever we update, all this information is being lost in somewhere middle.
Sugam: And no one can check that. It will be really good to integrate all this stuff into the blue system. That will help a lot of problems with the operation. And, uh,
Frederik: so you say you could be really helpful if something could help migrate the data from Prio-tool
Sugam: Yeah. Or maybe if we can build similar system inside the blue system instead of having separate prio-tool, because this is something that is built with Xcel that is very old.
Hans: . What about the blue system?
Sugam: Thisis built with the esp.net? I guess so

Markus: Yeah, C# and JavaScriptwhen
Markus: it comes to the defective products that you have, uh, you need to fix them. And do you use the prio tool to tell you which product to fix next?
Sugam: Um, in most cases, yes, at least for the laptop, because, uh, it's easy to find the laptop as compared with the phone because. Let's say in the Prio tool, we have a system to prioritize the product, which needs to be repair first.
Sugam: But if I select phone, find me the top five form to fix the, then it will find random phone. And we don't have that system inside the store to find where that product is. So it's really hard to use for the phone, but at least for the laptop we have basic shelf structure are based in our system. So for the laptop we are using to prioritize.
Sugam: which one to fix, but for phone, so far, we don't have any good system. So we never used that.
Markus: Okay. And do you know what, eh, this system takes into account when you, when it picks a product for you to fix, like, uh, how does the system decide which product to show you to fix?
Sugam: I think it's based on, I think it was based on different factor.
Sugam: One of them might be the storage time, which has been here for a long time. And, uh, one factor might be, I'm not sure, a hundred percent on this one, but I think one of them is like the current trend product, which is on high sale right now, which laptop is mostly sold so I know about this two factor, but there might be more, but I'm not sure.
Markus: Okay. And,
Markus: um, so you mentioned that, uh, for the phones, it would just show you five random phones and. Would it be right to say that a problem occurs when the system shows you five phones and you don't have them categorized in a certain way, physically in the workshop. So there's a link missing between how you store the products and how it's displayed.
Sugam: Yeah.
Markus: Okay.
Markus: So, are there any, do you have any like concrete examples, like any specific examples from using prio tools where the system you have now where you think this is really stupid, or this could be done in a better.
Sugam: You mean the alternative solution for prio tool or what can be done better inside priotool
Markus: yeah. What can be, what can be done different than how it is done right now?
Sugam: The tool itself is not that bad. It's good. But the problem is the problem with the prio tool is, uh, in the system itself. On which it is based on excel because right now we have like three technician working and the file we have the system prio tool to we have, can we access with only one technician. So three people cannot access one file at one time.
Sugam: Okay. So that's the main problem.
Sugam: And this problem leads to measure all the problem. Like if we have one computer, which has only one prio tool to running on. We have to be on queue to update our data always.
Markus: Uh, yeah. So one of your colleagues mentioned that there was an issue regarding the usage of spare parts, like the, uh, like, uh, it was difficult or impossible to attach a spare part like you're taking a spare part and use it for a product, then the spare part would be in storage or would it be in the system and does it, do they like combine?
Sugam: See it's a new spare one that we buy separately or the uses that we dissembled from defect product
Markus: um, that's up to you. I have. Would you prefer that those both spareparts were in the same system?
Sugam: Uh, no. No, it would be better to have like used spare parts and new spare parts in a separate system,
Sugam: at least for now we have a feature inside prio tool to register all the spare parts that we buy. And when it will be used that new spare part, we update in our prio tools so that they can track that, okay, this part has

been used at least for the new spare part. We have that system, but it's very, very implemented in a basic level.
Sugam: I just figure out that all this data never analyzed. It's just there to so people that it is there. Okay. But it's better to have that problem. So.
Markus: Yeah. Then, um, if you could please take us, uh, through the journey of a product from when it's, uh, bought in until it's been fixed.
Sugam: Okay. I'm mostly working with the laptop so I have the journey for the laptop, but it's somehow similar with other devices as well.
Sugam: So here that you drew up in could represent when we buy the product. Like not only by the product when we have product get in, then. We have, uh, four categories on this layer where we buy the product. And when we received the return product, like this return can be like as from reclamation or
Frederik: yeah, yeah.
Frederik: Other sources than buying.
Sugam: And, uh, then sometime customer just come with old laptop and just to give it to us for free that we don't buy it. And they just want to give it as a spare part so this is free products and this is lost product. The product that cannot be fixed, like if it's a reclamation that cannot be fixed and if it is lost from our system, then it is also considered as one of the input.
Sugam: But this is not that important for this system because these two must be related with the spare parts system.
Hans: It, can I say a picture of that or do you mind?
Sugam: I can send you an email or you can take a picture.
Hans: You can send the email.
Sugam: Otherwise if you want, you can take a picture Hans: Lets do both yeah. Just so I can follow up when you are saying
Sugam: okay. So let's just start with the "buy" product when we buy a product, it needs to be first go with the testing and that we says ?offended?, least. Uh, I think it's more clear on this picture
Sugam: so all the product category, which is ABC, like "perfekt" stand and "good" stand and "rimelig stand" configure as ABC, that all goes on ?offended? test
Sugam: And then here. Uh, at least for the laptop is what we do is like first it goes to the data deletion and then is real state so that we see all that, all the data on the laptop has been deleted and we can show customer and then it go for ready for sale which will be it will be updated at the store side.
Frederik: Okay. From when you do the data deletion and reset, then you will enter the information about the product into the blue system. And then we pull the data ,
Sugam: so this is something that we are missing on the blue system, whatever we do here, it's only updated on prio tool have nothing on the system. So once we are done with data deletion and report we update on prio tool okay, this laptop has been fixed with this, this, this issue. Okay, but this data is never updated on blue system.
Sugam: So this is something which is on ?F and T? Test. Then we have the product, which is grade a with the D stand, which has some kind of defect. So this one will run through one full extend test first we'll figure out, okay, this is defect, is it worth repairing or not, if this product is worth repairing, or this product is worth dissemble and you as parts, we'll go with extra case for testing
Sugam: and then if it's worth selling then again, it goes for data deletion. And
Sugam: so on this case, it is more important because on ABC standard. We just test, clean and, uh, make a data deletion. And so in this case, we don't really fix the defective parts. So on this case, we don't actually repair the defective parts. So it's not much important to update any information about this process on blue system, but mostly on.
Sugam: All the D-stand product, whenever update, for example, if i change battery, if we change the battery or

the screen on laptop or our phone, or if we change any speaker, whatever we change here, we can't update that information on blue system. So people always
Sugam: miss the information, that what has been fixed. So the store itself, they can never see what technician has done. It would be really good to be in the system to collect data from all the D-Stand repair and integrate that to the blue system. Okay.
Frederik: Uh, one important question can, can you tell us how you insert, uh, these items into the blue system or you don't actually yourself put items into the blue system.
Frederik: Or how has the process of entering data into the blue system? Because, uh, yeah. Do you understand what I mean?
Sugam: You mean, when you, when you say the entering the data, you mean repair data or information about the..
Frederik: So from, from when you buy the product, it goes through these steps. And then you need to enter the details into the blue system, or you cant enter the details into the blue system.
Frederik: If I'm correct, because I don't think we can make something that makes it easier to enter into the blue system. If we don't know how you enter data into the blue system,
Markus: as I understood it, the problem is that there's, the data is lacking in the blue system. There's not enough information in the blue system right now.
Markus: What's that, is that correct? Yeah. Uh, so it was missing like, uh, something like this product had its screen replaced at this time, by this technician, something like that. That's never entered into the blue system as far as I understood, right?
Sugam: True.
Frederik: I'm just thinking about what Peter Axel said, because. Uh, we can't in the scope of the project of our assignment, we can't build on top of another system, but we can build our own system from the bottom. And, uh, maybe we can make some connection layer between the two, but we can't build on top of the blue system.
Sugam: Uh, for that point.
Sugam: I can give you all a little bit solution as well. So to solve a similar problem. We already started a small project that never got into it.
Sugam: So what we have done is like we have built a similar feature related to the prio tool inside our blue system that will mostly collect what has been fixed. Like what has been repaired and who repaired it and how much time we spend and what kind of part has been changed with the part ID and how much, what is the cost price for the part?
Sugam: So the scope of this project was to, to add the price of the part with the cost of the product itself, because right now our another major problem with product is that if we buy a laptop for 500. On defect one. And we are sending that one for 3000. And if the part costs 2,800 for changing, and if we change the part and if we fix it, but if you look on the system, then we will still see that cost price of product is 500 and we are selling 4,500.
Sugam: So people will think that, okay, this product has a lot of margin, but in actual. We're spending like 2,500 for the parts itself. So we are not making so much money on that. So this information is also something we are missing. So when you scope a project to collect all of these data that will be more worth to use the price for the part as well.
Sugam: Hmm. Yeah. Is our system more confusing and fucked up or is it a little bit?
Frederik: No, no, it's, it's, it's clear. Um, but it's the. It's a journey from the product to the prio tool to the blue system that I understand what's happening in here and in the prior tool. But, um, as I understand the problem is that a lot of data is not able to be entered into the blue system, but to understand how the data is getting lost.
Frederik: I need some kind of picture of how the data is coming from you and, uh, into the blue system, because

we can't really do any fixes on the blue system,
Markus: the data from him doesn't go to the blue system.
Frederik: Yeah. But, but is it because it is blue system is lacking input fields and such. Okay.
Markus: Yeah. So as I understand it's like uh there's prio tool and the blue system, there needs to be like something that is both.
Frederik: Yeah. But if we have data and what's stopping us from getting caught in the same mode, because we can make something, uh, we can make it a better version of the prio tool uh, more sleek, updated, uh, with some extra features. But if we still can't enter the data into the blue system Then the data is still getting lost because the interaction layer between our program and prio tools
Frederik: so the blue system is still going to be the same because we can't change the blue system. You see what I mean? Because if it's the blue system that is lacking in opportunities of entering data, then the blue system needs to be changed. To accept the excavator.
Sugam: Oh, what kind of tool you guys using?
Frederik: Uh, Java, Java swing.
Frederik: We, uh, yeah, so we talked about doing a Java backend and a web front-end because, uh, as I, me and Markus worked here we know that the blue system already uses a web interface. So that would maybe be familiar and easy to use also because it's easier to make a web interface, then it is to do a system GUI.
Frederik: But there was a lot of good .
Sugam: Yeah. So far for that solution, I don't know,
with the Java and what kind of database. Markus: Don't worry about the system, uh, because it's a very small project in it's very limited, what we're able to do. So all we have to get from you is just, uh, what you have of current issues and, uh, how you, if you have any ideas of how it could be improved, the stuff you have right now, then, then we'll just, uh, we'll worry about building it, and what we can do.
Hans: We need a list of requirements for the product.
Markus: So in the end, we're gonna, uh, after some meetings we're gonna have like a full list of things that you want a system to be able to do that it can't do right now. And, uh, and then we'll, uh, we'll come with some prototypes and demonstrate stuff at a later time.
Frederik: Also this first interview is just to get a feeling of what your current situation is now. Yeah. Because we are going to integrate the solution into,
Sugam: and I think that will be another discussion maybe. So I think this discussion, at least we should collect all the requirement for the prio tool
Markus: yeah. So like we just make an upgraded version of what you have right now, a different one.
Sugam: So for that one have this small table, this is somehow similar to prio tool. I don't have access to prio tool right now. . So I cannot show it, but what most of the data that we need from prio tool input on this. So we can use the same field on the tables. So we have like three kind of repair . We have, uh, reclamation, self chosen and street repair, and we have the product category and we have the item ID.
Sugam: So this is the actual, the product number. So all the phone or all the laptop has their own item-ID and then we have the repair part type. This is something that our prio tool doesn't have. So it really be good if you guys can build this one extra. So what it does is like whenever use, whenever we use new parts or whenever you use old used past, or sometime we never use parts because it's just testing and resetting and sometime we send to external repair.
Sugam: So what does these four things to. With this data, we start, at least we can analyze how many used spare parts we are using and how much we're spending on new parts and how much money we are spending for external repair
Frederik: and tracking how many parts you have. In-house.

Sugam: yeah
Markus: Yeah. so every time you, every time you write a part here, like, if it's a new part, you have some new parts in stock, it should take one from your stock and apply it to this product. Yeah.
Frederik: Okay.
Sugam: But this is when you were using the parts. This is not when we are buying the parts but if we have this system, it's easier to make another system for buying process (of parts)
Sugam: um, which will make a stock for how much new parts we have. Yeah. And this one, and this another system can integrate together to make a stock management.
Sugam: And this can be the description. What kind of repair we have fixed. And, uh, this is the actual part-ID like all the parts we use, like the screen or dock-connector everything that has like similar kind of part ID, same like item ID that going to be unique for every parts and then price suggestion
Sugam: so, so right now what our prio tool have is like, whenever we use parts. It has a price suggestion, but it is, uh, not working as expected. Like we have to figure out manually all the technician that, okay, this part ID got this one and we have to find them. We have to put in manually, but this is something that can be done automatically.
Hans: What are the factors? And the price suggestion is the time spent and the price of parts.
Sugam: It's only the price of parts. So for example, if we have a system to stock all the new parts. Yeah. While registering, then we will know that, okay, we have cost like 79 kr. For this part there.
Sugam: So that price will automatically come here and suggest to us, not as a technician, we have to go and find it manually.
Sugam: So if we have that system fully functional, then we don't need this price. Because it will fill up itself, but right now it will be the second layer to build up. So at least if it come as a suggestion, then we can just manually fill easily. We don't need to go to another website to find, okay, it cost this much.
Sugam: And then we have repair time and we have a comment it's necessary. This is not mandatory. And we have the initials (technician name) like which technician has filled this one and, uh, This data and store number. This is something that systems will automatically track.
Sugam: So this is the basic requirement for the prio tool to you guys need to be.
Sugam: Yeah, at least with all this data. What we can expect.
Sugam: So at least we can know how much the spare part is being used. Like all the spare parts and which category of product is mostly repaired at different store.
Sugam: So the scope of this one (shows pages from a project) is like, with all this data, we can store different thing because right now we have a lot of data in prio tool to what no one is actually using, maybe they're using, but we don't know. So this one we can analyze, like, make a comparison between different blue city, five, six blue city like who is using a lot of spare parts, new one, or used one.
Sugam: And, uh, which product is mostly being repaired and on which product we are mostly using new parts on, on which product we are mostly using used parts. So a lot of different comparison between product.
Markus: Between stores.
Markus: Okay. So, uh, we're thinking maybe. We meet up every second week, every other week, and, uh, as the project progresses, we'll have more and more specific questions most likely. And, uh, they, they won't take up that much time by then, but, uh, we're always happy to hear your thoughts. If there's something else you just, thought of during your Workday and you think, oh, I have to tell these guys that the prio tools can't do this.
Markus: Or I really wish that I was able to do this when I have repaired a product or something like that. Then, uh, if, if you get those thoughts, just scribble a small post-it.
Markus: Yeah. And, uh, No one has anything else.
Frederik: Yeah. So if you don't have any more things, so

Sugam: that's true. I don't have anything more to add. It's like, I have a lot of things, but if I say everything together, you guys will be more confused because I don't know what is the main scope of this project and what is the limitation?
Sugam: And what is the exam. I don't want to mess up everything in one conversation.
Hans: So I think the biggest limitation we have is we cannot use any technology you're using to build on top of, that we have to build it from the ground up. So our solution has to be all encompassing.
Frederik: Uh, so another problem is that as, as a student and what we are writing, In the reports, because alongside this, a physical product, we are going to write a report, which is uploaded to our, uh, project database on our, so we as i understand, uh, the blue city won't be happy that we made some interaction with the blue system that is a company knowledge, and then it's posted.
Frederik: So I think if we can make it a seperate sort of like an improvement of the prio tools now. Yeah. That would, that would make sense because there's no interaction with company sensitive
Sugam: and I don't think it's really necessary to integrate your system. Together with blue system.
Markus: We'll just do a standalone system and then it could be, you know, translated to be a part of the blue system that you have, your it department could use our system and then somehow implemented it into the blue system. Perfect.
Frederik: Okay. Thank you so much. It sounds like a really, really exciting.

B Contextual Inquiry with Blue City

Questions for Interview 2: man 20. sep.
Goal of interview: Understand the workprocess better.
Result: See and walk with the technician through a repair
- Do you actually use PRIO tool as the way to know what products to repair?
- What is your system for spareparts?
How do you keep track of spareparts?
How do you find the spareparts needed for a repair?
What happens in the case where you have more spareparts for a product than products that need repair?
- How do you sort the repair products on the shelf? Is there a system for that too?
- How do you distinguish / prioritise customer repairs and repairs of blue city's own products. Is this information found in PRIO-tools?
- You mentioned last time that you wanted an improved dashboard, can you explain what would make a good dashboard?

C Illustrations

Adding new products

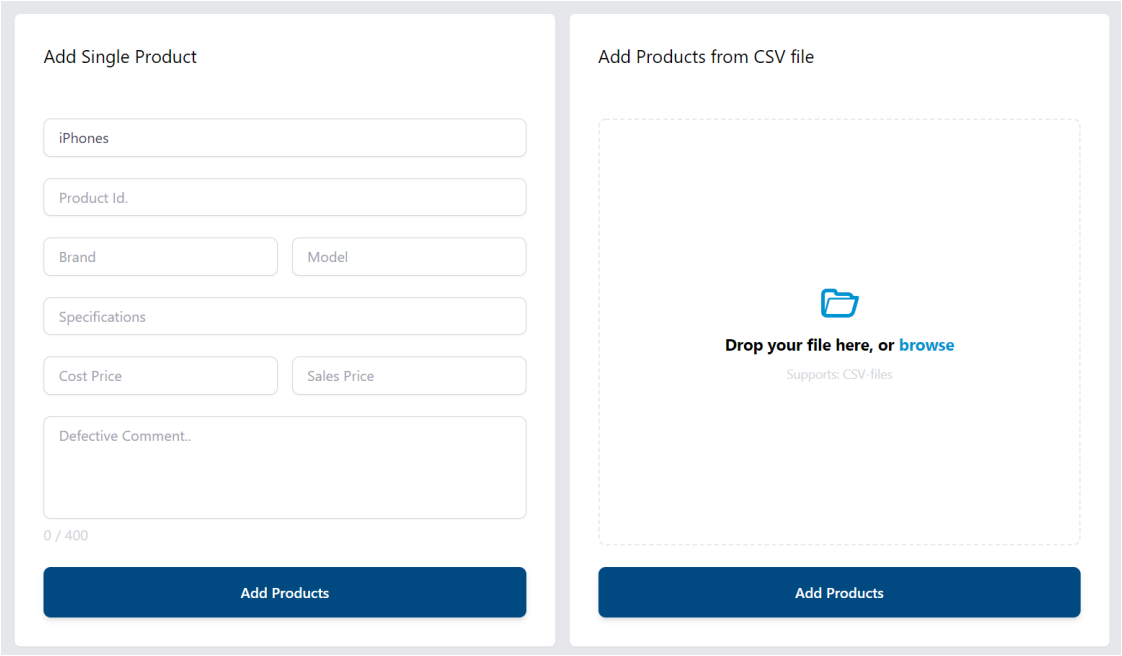


Figure C.1: Add product page

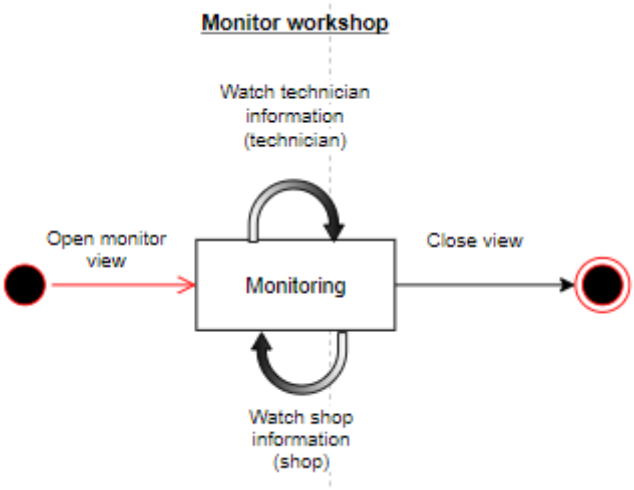


Figure C.2: Monitor Workshop

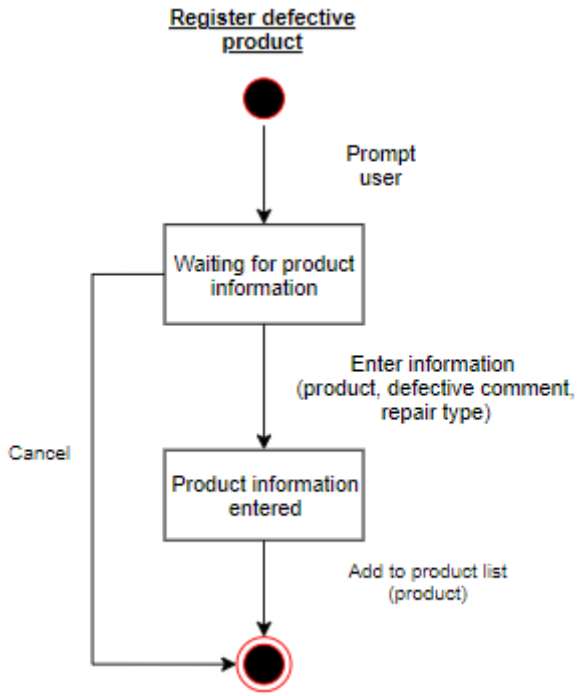


Figure C.3: Register Defective Product

Figure C.4: Use case diagram

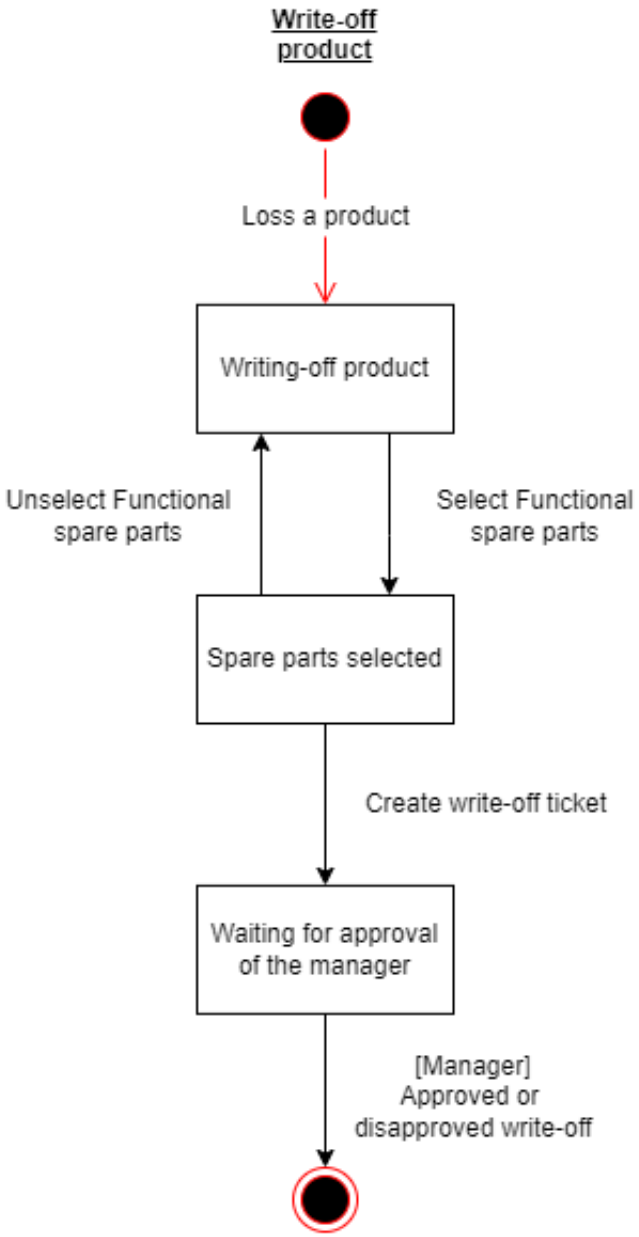


Figure C.5: Write-off Product

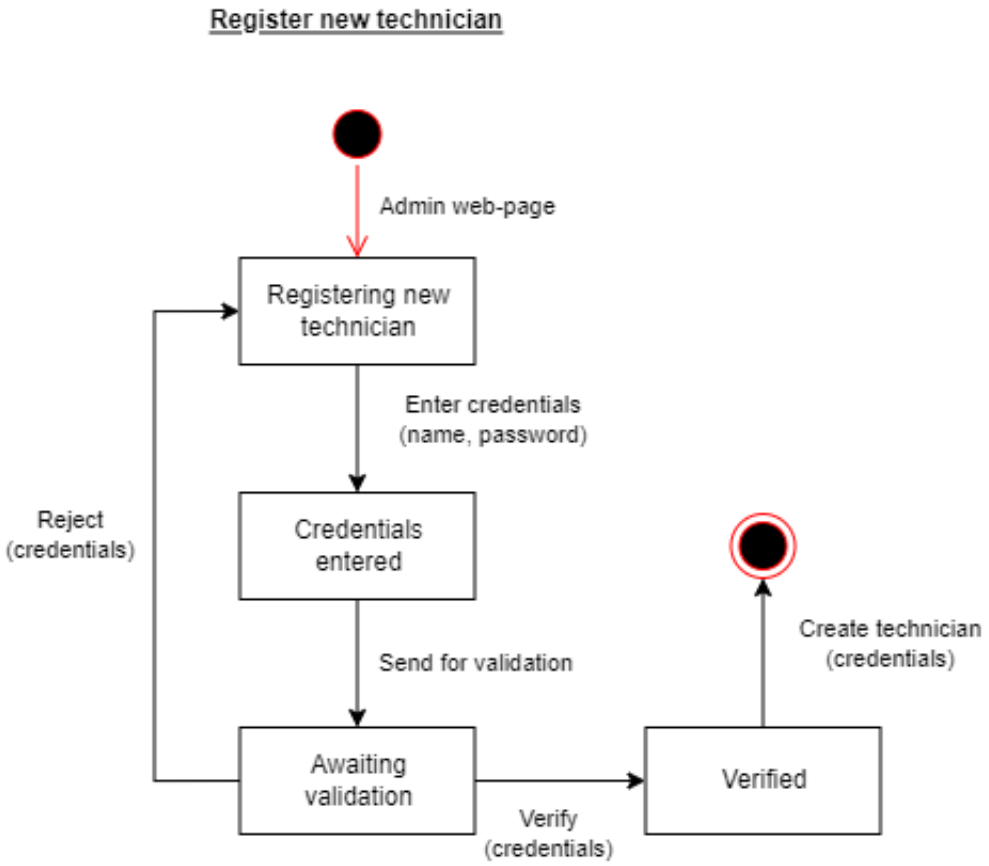


Figure C.6: Register-Technician

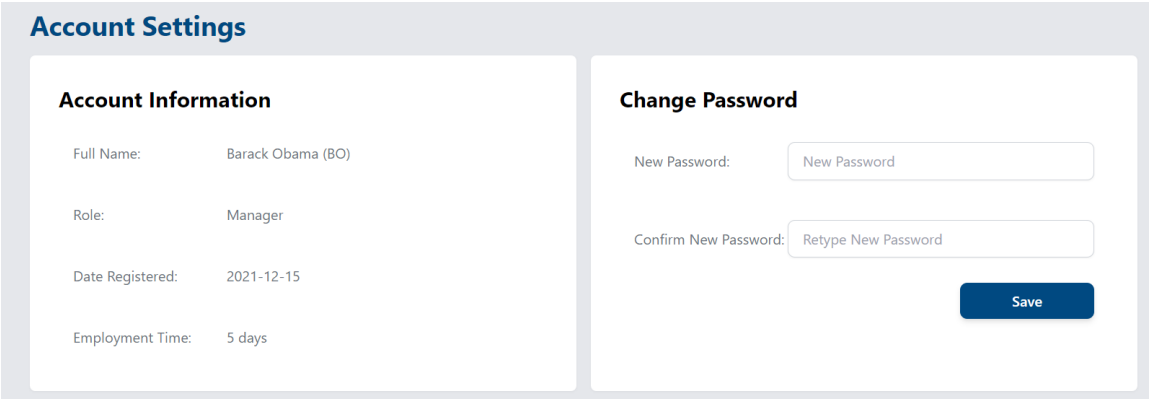


Figure C.7: Settings page

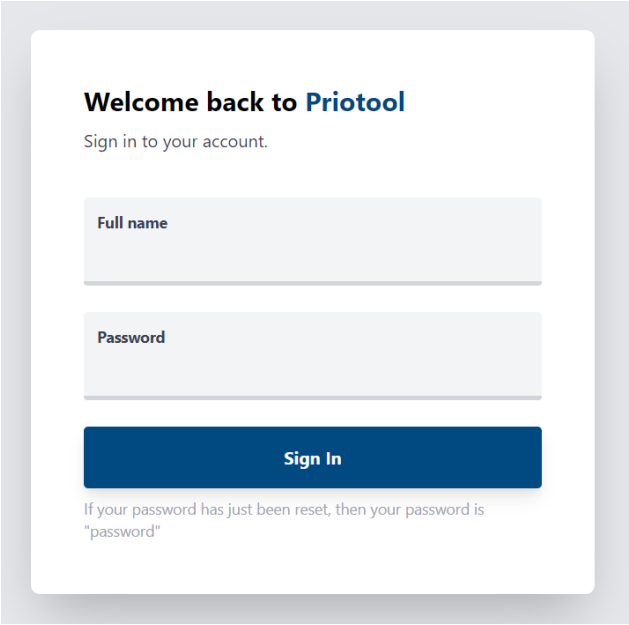


Figure C.8: Login page

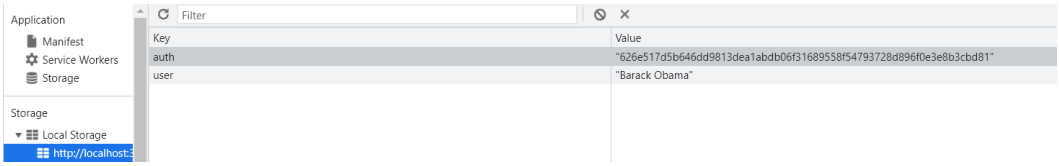


Figure C.9: Cookie stored on local storage

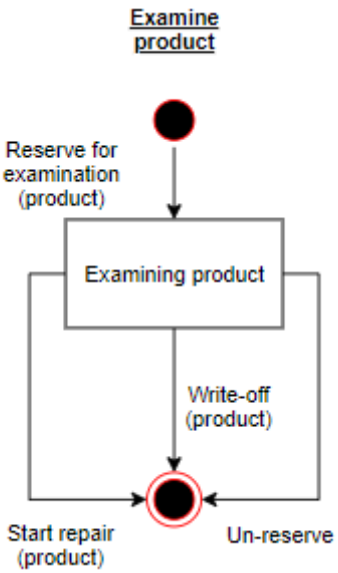


Figure C.10: Examine Product

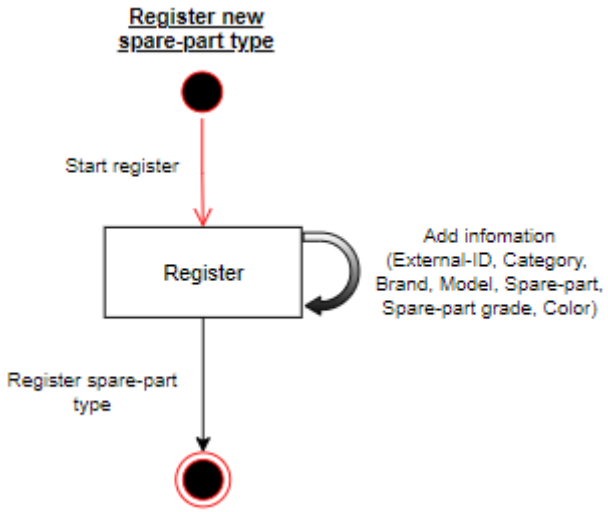


Figure C.11: Register New Spare-part Type

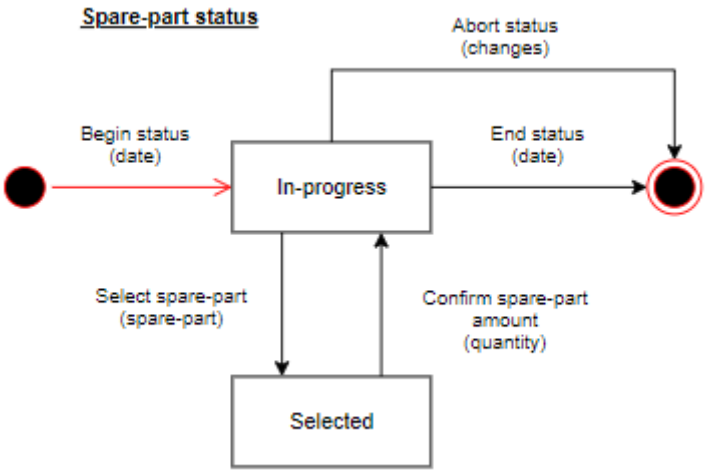


Figure C.12: Spare-part Status