Computer Engineering Project II

By Group 2

Anton Geneser Matzen
202008936@post.au.dk
Std.nr: 202008936
Au-id: 683185

Emil Hilligsøe Lauritsen
202004154@post.au.dk
Std.nr: 202004154
Au-id: 668867

Martin Michaelsen
202007433@post.au.dk
Std.nr: 202007433
Au-id: 672598

Kevin Vollesen Schønberg
202007282@post.au.dk
Std.nr: 202007282
Au-id: 674059

1. Abstract

This report describes how the Kitchen Guard system were designed, implemented, and tested. The system is developed using the agile development process Scrum. The systems requirements were engineered following the requirements engineering process described by Sommerville. Use case based requirements were created from this. The systems architecture was designed using Kruchten's 4+1 model view. From the use case based requirements test-cases were created, which tests all functionality specified by the functional requirements. Finally, an acceptance test was successfully performed to verify that the system functioned as expected. The system has some flaws in terms of the testing of the system. The environment in which the system was tested did not entirely reflect a realistic environment, which could lead the system to perform poorly in a more realistic setting. The implementation had slight defects, however a working implementation was created and the requirements were met.

2. Page Counting Method

As stated in "WEEK PLAN COMPUTER ENGINEERING PROJECT II" the report should be between 15 to 20 pages where each page is counted as having 2400 characters. When finding the page count of this document some characters can be ignored in the final count. This will be our tables, figures, code snippets and all sections before the introduction. The character count can be seen in the table below.

Section	Character count
Whole document	63711
Section before introduction	3653
Tables	4435
Code snippets	9396
Final character count	46287

This means that this report is 19.28 pages

3. Individual contributions

The contributions from the development team can be seen in the Appendix.

Sebastian Scheby Berg was a member of the group for the first couple of weeks, however he did not contribute enough to take responsibility for any part of the project development.

4. Table of Contents

1. Abstract 2
2. Page Counting Method 2
3. Individual contributions 2
4. Table of Contents 3
5. Introduction 5
6. Problem description 5
7. Methods 6
   7.1. Development process 6
   7.2. Project management 7
   7.3. Requirements engineering 7
      7.3.1. Requirement engineering processes 7
   7.4. Architectural design 9
      7.4.1. Scenarios 9
      7.4.2. Logical view 9
      7.4.3. Process view 9
      7.4.4. Development view 9
      7.4.5. Physical view 9
   7.5. Design patterns 9
   7.6. Testing process 10
      7.6.1. Development Testing 10
      7.6.2. Release Testing 11
      7.6.3. User Testing 12
8. Results/analysis 12
   8.1. Requirements Specification 12
      8.1.1. List of use cases 13
      8.1.2. UML Use Case diagram 13
      8.1.3. Functional Requirements 15
      8.1.4. Non-functional Requirements 17
   8.2. Architecture and design specification 17
      8.2.1. Architectural design method 17
      8.2.2. Physical view 17
      8.2.3. Logical view 19
      8.2.4. Process view 22

8.2.5. Development view 26
8.3. Test Specification 26
8.4. Implementation Documentation 27
8.4.1. The Kitchen Guard Controller 27
8.4.2. The Web API 30
8.4.3. WAMP 32
8.5. Acceptance Testing 33
9. Discussion 33
9.1. SCRUM 33
9.2. Requirements Specification 33
9.3. Architecture and design specification 33
9.4. Test Specification 35
9.5. Implementation 36
10. Conclusion 37
10.1. Objective 1: Requirements Specification 37
10.2. Objective 2: Architecture and design specification 37
10.3. Objective 3: Implementation 37
10.4. Objective 4: Test Specification 37
11. Sources 38
12. Appendix 39

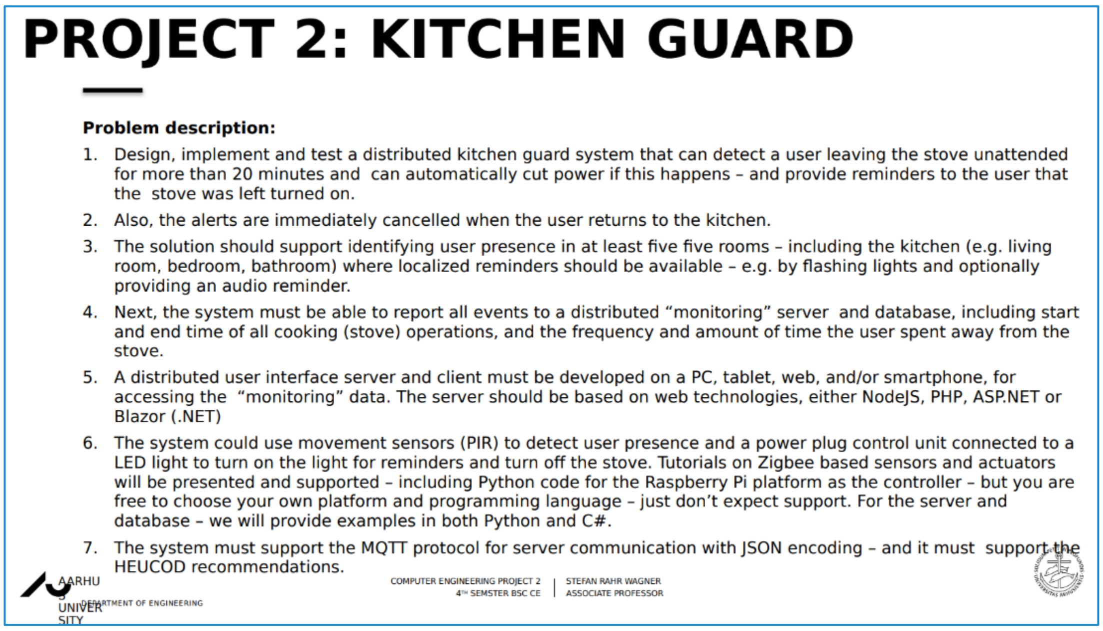
5. Introduction

Safety is very important to maintain especially when cognitively challenged people live on their own as they can easily increase the risk of accidents unknowingly. They can simply forget to lock the door or leave candles on when leaving the house or going to sleep. Situations like these are difficult to prevent without having a caretaker or family member in the house to help.

But with technology progressing, the possibility to place monitoring systems in the house can help maintain a safe home without the presence of another person. In this document we will show how we specified, designed, implemented, and tested a kitchen guard system. We will cover what methods we used and what results they lead to. We will then discuss the results and reflect on what could have been done better.

6. Problem description

For this project we were given the following case description:



PROJECT 2: KITCHEN GUARD

Problem description:

1. Design, implement and test a distributed kitchen guard system that can detect a user leaving the stove unattended for more than 20 minutes and can automatically cut power if this happens – and provide reminders to the user that the stove was left turned on.
2. Also, the alerts are immediately cancelled when the user returns to the kitchen.
3. The solution should support identifying user presence in at least five five rooms – including the kitchen (e.g. living room, bedroom, bathroom) where localized reminders should be available – e.g. by flashing lights and optionally providing an audio reminder.
4. Next, the system must be able to report all events to a distributed "monitoring" server and database, including start and end time of all cooking (stove) operations, and the frequency and amount of time the user spent away from the stove.
5. A distributed user interface server and client must be developed on a PC, tablet, web, and/or smartphone, for accessing the "monitoring" data. The server should be based on web technologies, either NodeJS, PHP, ASP.NET or Blazor (.NET)
6. The system could use movement sensors (PIR) to detect user presence and a power plug control unit connected to a LED light to turn on the light for reminders and turn off the stove. Tutorials on Zigbee based sensors and actuators will be presented and supported – including Python code for the Raspberry Pi platform as the controller – but you are free to choose your own platform and programming language – just don't expect support. For the server and database – we will provide examples in both Python and C#.
7. The system must support the MQTT protocol for server communication with JSON encoding – and it must support the HEUCOD recommendations.

AARHUS UNIVERSITY DEPARTMENT OF ENGINEERING

COMPUTER ENGINEERING PROJECT 2 4TH SEMSTER BSC CE | STEFAN RAHR WAGNER ASSOCIATE PROFESSOR

Figure 1: Kitchen Guard Problem Description (Wagner, Week 1 - Introduction, 2022, p. 17)

We used this case description to develop both functional and nonfunctional requirements. That means that the problem description is the base for what we later used to design the architecture and test specification.

For this project we were given a list of milestones which needed to be performed. From this and the given problem description we derived the following objectives for our project:

1. Construct a requirement documentation with use case based requirements
2. Construct an architecture design specification
3. Implement and document the Kitchen Guard system
4. Construct a test specification and perform acceptance test of the system

7. Methods

The Kitchen guard system were developed using several software engineering methods. The different methods are descried in the following sections.

7.1. Development process

For our software development processes we used an incremental development process. This made sense since we are a small team, and we expected the shape of our system to change during the design processes. The incremental development process[1] is shown in Figure 2.

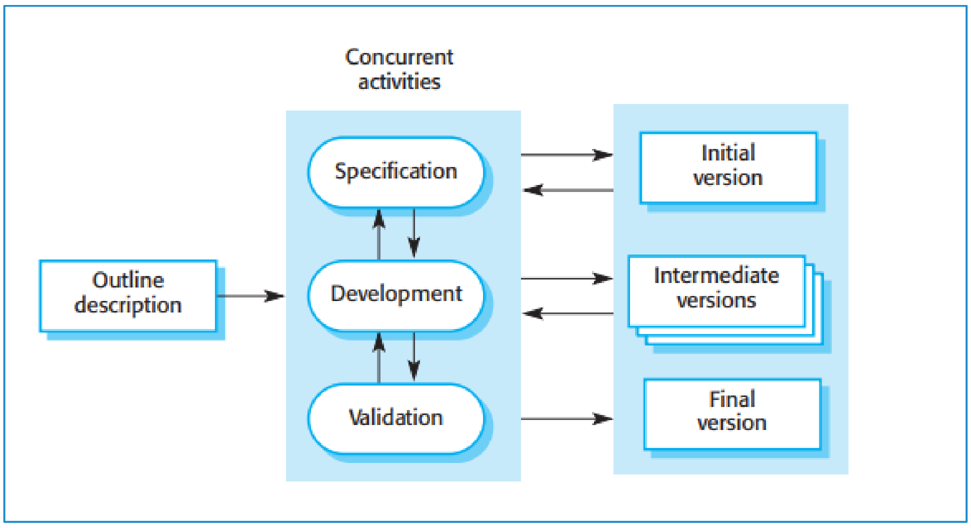
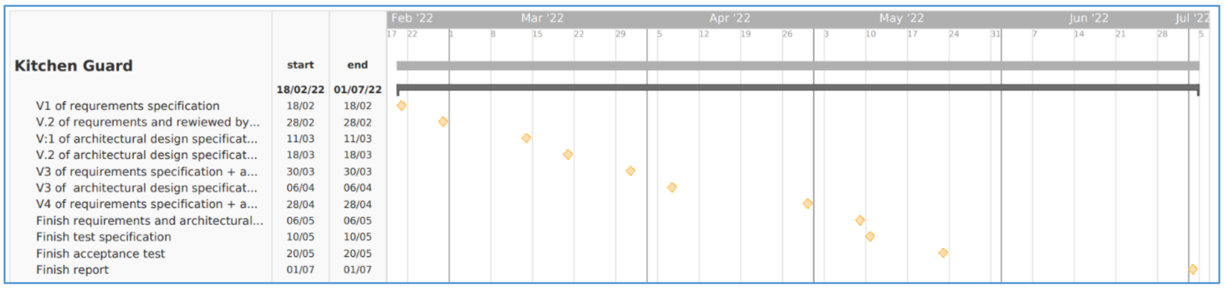


Figure 2: The incremental development process (Sommerville, 2016, p. 50)

If we take a closer look at Figure 2 we see that the specification, development, and validation process happen concurrently. This was also the case in our processes. As Figure 2 indicates we ended up with several version of requirements, design, and test-specifications. The most recent versions of these specifications can all be found in the Appendix. By looking at the Gantt chart on Figure 3, we get a more precise look at when and in which order we created the different versions.



Kitchen Guard	start	end
	18/02/22	01/07/22
V1 of requrements specification	18/02	18/02
V.2 of requrements and rewiewed by...	28/02	28/02
V:1 of architectural design specificat...	11/03	11/03
V.2 of architectural design specificat...	18/03	18/03
V3 of requirements specification + a...	30/03	30/03
V3 of architectural design specificat...	06/04	06/04
V4 of requirements specification + a...	28/04	28/04
Finish requirements and architectural...	06/05	06/05
Finish test specification	10/05	10/05
Finish acceptance test	20/05	20/05
Finish report	01/07	01/07

Figure 3: Gantt Chart

[1] (Sommerville, 2016, p. 50)

7.2. Project management

In this project the agile management method used was SCRUM[2]. It was used to break down complex tasks into bite sized pieces, which were able to be solved bit by bit over an extended period.

Every week an informal meeting would be held, and a plan made of what parts of the larger problem should be solved, then through intermittent sprints a new iteration would be created. Here an iteration is simply a more detailed and complete version of one of the objectives.

There were iterations of the requirements-, design- and test specifications, the implemented kitchen guard system as well as its documentation. Each part of development would end up having around three or four iterations. In this manner it was possible to easily validate and or verify our solutions throughout the process, continuously testing and checking each iteration.

A specific scrum master was not assigned in the group, instead each member contributed to gathering the group when meetings needed to be held. The creation of iterations was designed around the milestone deliverables assigned. In this manner it was also possible to easily keep a weekly meeting schedule. Each member of the group would volunteer to solve a piece of the problem with each iteration created. Then once solved the group would gather to validate and verify the proposed solution. An overview of the schedule can be found in Figure 3.

7.3. Requirements engineering

One of the first steps when designing a system is requirements engineering. It is here one constructs the requirements for the system which describes the services the system should provide. The requirements also describe the constraints the services can have on the systems operation. When constructing the requirements for the Kitchen Guard application we followed the process described in Sommerville[3]. From this process a requirements specification was made, which can be found in the Appendix.

7.3.1. Requirement engineering processes

As Sommerville describes the requirement engineering process is an iterative process containing the stages shown on Figure 4. These stages are requirement elicitation, requirement specification and requirement validation. These steps are often repeated throughout the process of constructing the requirements for the system, which is also the case for our system where, we performed each step multiple times.

[2] (Sommerville, 2016, p. 84)
[3] (Sommerville, 2016, p. 111)

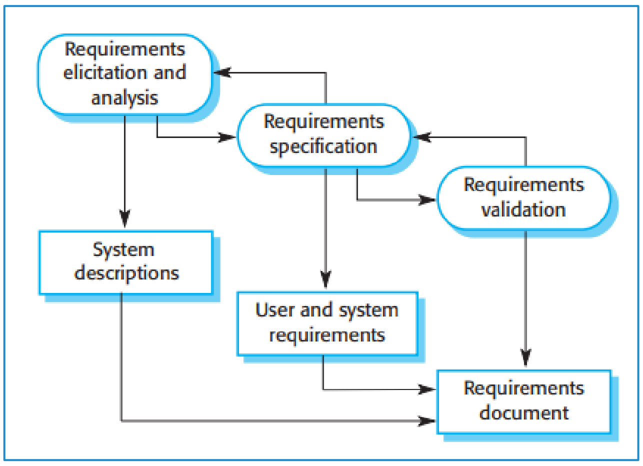

Requirements elicitation and analysis
Requirements specification
Requirements validation
System descriptions
User and system requirements
Requirements document

Figure 4: Requirements engineering process (Sommerville, 2016, p. 55)

7.3.1.1. Requirement Elicitation

The requirement elicitation process is the process of deriving the system requirements. This is often done in collaboration with the client and future users of the system. For the kitchen guard application, we derived the requirements from the given case description[4], which can be seen as our client's description of the wanted system. In this process we discussed which parts of the description should be considered functional or non-functional requirements[5]. This was then noted on a shared document. This process was performed until the whole team had reached an understanding of the scenarios described in the case description. At this stage informal use case scenarios were also made for the functional requirements.

7.3.1.2. Requirement Specification

The requirement specification process is where we take the information found in requirement elicitation and use it to construct user and system requirements for our system. For the Kitchen Guard application, we took the informal requirements found in the requirement elicitation process and rewrote them to follow the definition of user requirements[6]. We also took the informal use case scenarios and rewrote them to be our use case based system requirements. We also took the non-functional requirement and rewrote them to follow the description of a system requirement[7].

7.3.1.3. Requirement Validation

In the requirement validation process we check that the found requirements define the system as described by the client. For the Kitchen Guard application, we performed validity, consistency, completeness, realism and verifiability checks[8] of the requirements. We also performed requirement reviews, which were performed by a peer-review group. Throughout the process of designing our system this was repeated multiple times to ensure that the require-

[4] (Wagner, Week 1 - Introduction, 2022, p. 17)
[5] (Sommerville, 2016, p. 105)
[6] (Sommerville, 2016, p. 103)
[7] (Sommerville, 2016, p. 103)
[8] (Sommerville, 2016, p. 129)

ments were defining the needed system. As a last step in the requirements validation, we performed test-case generation which we did by constructing simple informal testcases for our requirements. This was to ensure that the requirements were testable.

7.4. Architectural design

To design and structure the architecture of our system we used Kruchten's 4+1 model view[9]. It provides standards for developing system architecture which we have used to make sure the design was clearly defined. In the following sections we will briefly mention the 5 views.

7.4.1. Scenarios

This view is different compared to the other 4 views because it describes how the system should work using use-cases. It is used to define how the other 4 views should be put together.

7.4.2. Logical view

This view defines the system mainly from the functional requirements. This is illustrated by UML state and class diagrams. The state diagram explains the logical flow of the system, this is the states of the system and what will trigger it to transition between states.

The class diagram is used to describe how the actual software classes are linked and what their functionality are.

7.4.3. Process view

This view gives a clear example of how the classes in the system communicate during scenarios. This is illustrated by a UML sequence diagram and will normally show communication between classes, but in our system, we have used functions within classes where classes would normally go. This is further explained in 8.2.4.

7.4.4. Development view

This view shows the layers of the system using a UML package diagram. This splits the system into presentation, business logic and data access layer. The different software modules are then placed in the layer in which they belong.

7.4.5. Physical view

The physical view, using a UML deployment diagram, shows how the systems software components are divided over the different hardware components. It also shows data and control flow between these components.

7.5. Design patterns

Design patterns help solve general patterns in computer engineering and helps maintain good practice. During our design process we looked at a few that would benefit our system.

This first design pattern we considered was the model view controller patten. This pattern is used when there are multiple ways to view and interact with data. In our case we also foresaw that the system should be scalable, which would likely lead to changes in the data.

[9] (Sommerville, 2016, p. 173)

Secondly, we looked at the broker pattern. The broker pattern deals well with a controller that needs to communicate with more than one server, actuator, or sensor. As would be the case in our system.

We ended up implementing the model view controller pattern[10]. If we look at the deployment diagram in Figure 8, we see that the sensors and actuators reflect the model. Our raspberry device reflects the controller and finally the webserver is the view element.

7.6. Testing process

To help ensure proper testing of the implemented kitchen guard system, the software testing process was used. The process can be seen in Figure 5.

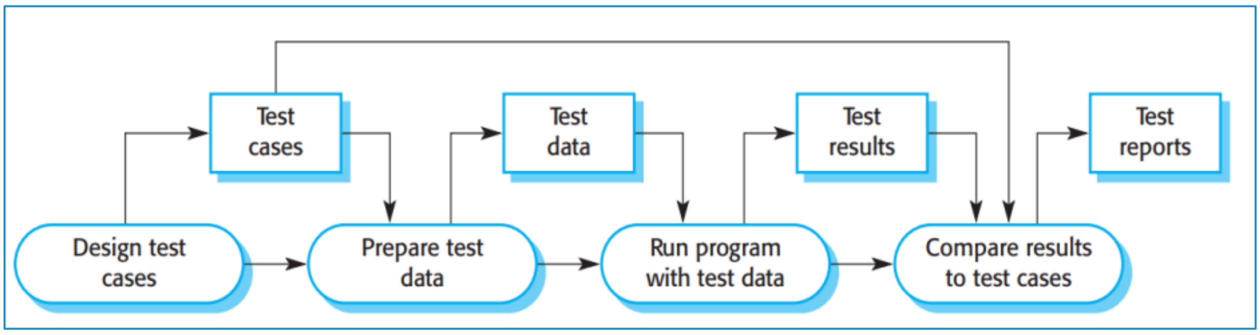


Figure 5: Software testing (Sommerville, 2016, p. 230)

The first step in the process was to design the test cases. This had started right from the beginning when creating the use cases. As the use cases would later be what was used as the test cases. The use cases were therefore made sure to be testable. Being testable means that one can insert an input and get an output to compare with an expected output, as in something that is not arbitrary and vague, but rather well defined and absolute.

The process was utilized in different ways throughout the project, one can split up its usage in three different stages. These stages are development testing, release testing and user testing. The entirety of the software testing process was not utilized for all stages of testing as a test report was only made for the acceptance test[11].

7.6.1. Development Testing

When developing the implementation of the kitchen guard system there was more focus on unit testing and making small incremental changes with multitudes of tests to validate and verify if the development was going in the right direction. We used the use cases and test cases to validate and verify the implementation.

One can split up development testing into three separate stages called Unit testing, Component testing and system testing.

Unit testing was used to verify that each unit meets its specification. It is defined by Sommerville as:

[10] (Sommerville, 2016, p. 196)
[11] (Sommerville, 2016, p. 81)

"*Unit testing*, where individual program units or object classes are tested. Unit testing should focus on testing the functionality of objects or methods."[12]

When these units, as in objects or methods, have been tested to work. One can start integrating these working units together to make what is called a component, which allows the developer to test how the units interact. This kind of testing is called Component Testing and is defined by Sommerville as:

"*Component testing*, where several individual units are integrated to create composite components. Component testing should focus on testing the component interfaces that provide access to the component functions."[13]

The final stage of development testing is called System testing. This kind of testing involves integrating some of or all the components to test the complete system. System testing is defined by Sommerville in the following words:

"*System testing*, where some or all of the components in a system are integrated and the system is tested as a whole. System testing should focus on testing component interactions."[14].

7.6.2. Release Testing

The release testing stage ensured that the use cases defined in the requirements specification were satisfied. This was done by creating a test case around each use case ensuring that the functionality of the requirements was tested meticulously. This is very useful because it ensures that all the necessary components and attributes of a good test are included, as they already can be found in the use cases.

Each test case was constructed according to the acceptance testing process[15] which can be seen in Figure 6: Acceptance testing processFigure 6.

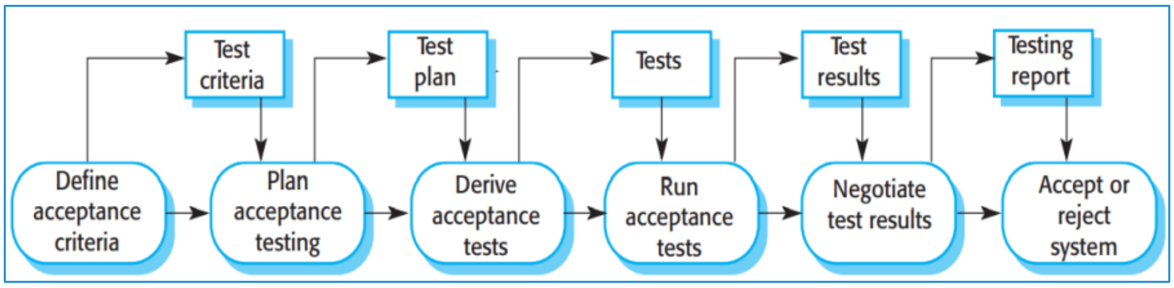


Figure 6: Acceptance testing process

We only constructed the test cases in the release testing stage, meaning we completed only the first three steps of the acceptance testing process.

[12] (Sommerville, 2016, p. 232)
[13] (Sommerville, 2016, p. 232)
[14] (Sommerville, 2016, p. 232)
[15] (Sommerville, 2016, p. 250)

7.6.3. User Testing

The final stage of testing done was user testing. Here we completed the last three steps of the acceptance testing process. This involved having a peer review group member test the implementation according to the test cases defined. A developer was there to supervise the testing, helping the peer review group member properly execute the test cases as specified in the testing report.

In this manner the different functionalities of the system were able to be evaluated and checked by an unbiased third party to verify that they fulfilled the requirements given. The results of this user testing can be seen in the acceptance test, which can be found in the Appendix.

8. Results/analysis

In the following sections the results and findings of our work will be explained and commented.

8.1. Requirements Specification

Following the requirements engineering process described in section 7.3 a requirement specification has been produced by the development team. This has been created following the criteria described in the project specification[16]. In the following section a walkthrough of the most essential parts of the document will be represented, but the full requirement specification[17] will be in the Appendix. The requirements will be stated using the use case method, in which a series of actors will be referenced. These actors can be split into two groups, which are primary and secondary actors. The primary actor is the user, who will be interacting with the secondary actors, which are the movement sensors, lights, power plug and the database.

[16] (Wagner, Week 1 - Introduction, 2022, p. 17)
[17] (Lauritsen, Matzen, Michaelsen, & Schønberg, 2022)

8.1.1. List of use cases

A list of the use cases with their associated names and short descriptions are listed in Table 1. This table and descriptions are meant to establish an overview of the use cases which will be described further in the following sections.

UC ID	UC name	Description
UC1	Monitor Stove	The sensor detects if the user is in the kitchen and sends events to the controller,
UC2	Start Timer	A timer is started and after 5 minutes the user is alerted in the room he occupies if no movement is detected. If the user returns before 5 minutes, the timer is not started. And no lights are affected.
UC3	Alert user	The system tries to find the user and alert him/her by turning on the light in the room the user is in. If the user reenters the kitchen or the user is not found, then no lights turn on.
UC4	Search for User	Detect if the user is in one of the monitored rooms and informs the controller.
UC5	Detect User Return	An event is received informing of the user presence in the kitchen after an absence of more than 10 seconds with the stove turned on.
UC6	Turn stove off	The user has been absent from the kitchen with the stove turned on for more than 20 minutes, the stove and alerting light is turned off, while an indicating light is turned on in the kitchen.
UC7	View Data	Allows the user to see the data from the database.

Table 1- Use case overview.

8.1.2. UML Use Case diagram

The use cases from Table 1 can be seen on Figure 7 as a UML Use Case Diagram. On this diagram the different use cases are depicted with dependencies to actors and other use cases. We can see which use cases are dependent on which actors by the solid lines connecting the use case bubbles to the figures depicting the different actors. Here actors are differentiated by whether the actor is a device or a person, identified by the head shape of the figure. Round for a person and square for a device.

The dependencies between different use cases can also be seen depicted by dashed lines between the use case bubbles. This indicates that a use case is dependent on an earlier use case being completed or started for the current use case to be able to execute. A final thing depicted on the figure is which use cases are handled by the Kitchen Guard program on the raspberry pi. These are the use cases contained within the box marked “Kitchenguard.py”. As an alternative we have use UC7 which is handled by the web API instead of the controller.

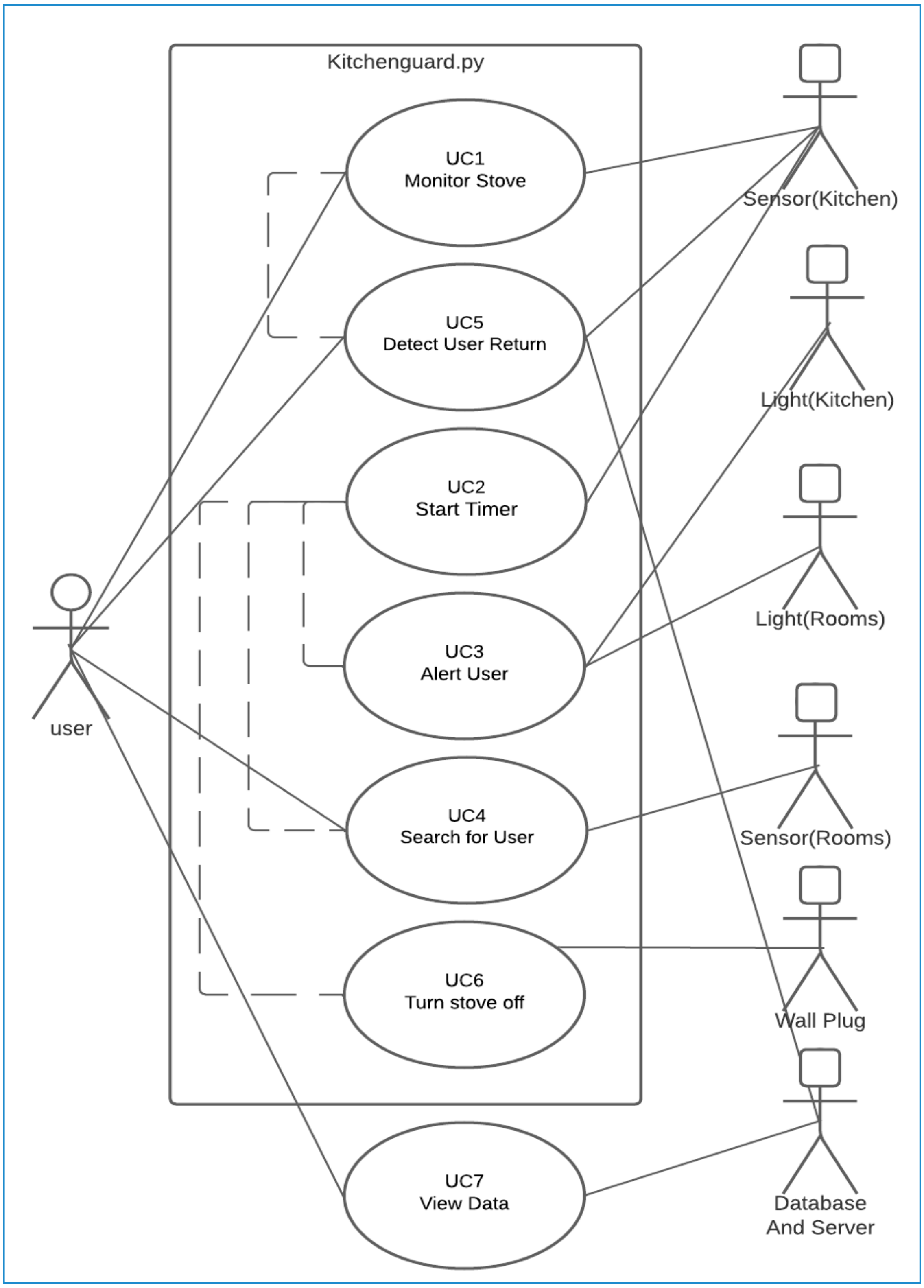


Figure 7. UML Use-Case diagram of the Kitchen guard.

8.1.3. Functional Requirements

Each use case is presented in detail, with important information such as use case id, pre- and postconditions, main scenarios, primary actors, and stakeholders. Here the primary actor will be the actor initiating the use case and the stakeholders will be the actors affected by the use case. Furthermore, the preconditions are the conditions which needs to be met for the use case to be initiated and the postconditions are the conditions which needs to be met after the use case has executed. At last, the main scenario is the series of events done by the use case, and the extensions is a possible divergence from said main scenarios.

Name	**Monitor Stove**
Use case	UC1
Primary actor	User
Stakeholders	User & sensor(kitchen)
Precondition	Stove is turned on
Postcondition	Controller receives the event from the sensor
Main scenario	1. Sensor in the kitchen scans for movement 2. The sensor sends an event to the controller
Extension	If the Sensor in the kitchen is broken and fails to send an event, the controller will assume that there is no movement in the kitchen.

Name	**Start Timer**
Use case	UC2
Primary actor	Sensor (Kitchen)
Stakeholders	Sensor (Kitchen)
Precondition	The controller has not received events confirming that the user is still in the kitchen for more than 20 seconds.
Postcondition	The timer is started
Main scenario	1. Start a 20-minute timer
Extension	None

Name	**Alert User**
Use case	UC3
Primary actor	Alerting lights
Stakeholders	Alerting lights
Precondition	UC2 + 5 minutes has passed
Postcondition	Light is turned on where the user is
Main scenario	1. UC4 to find the room the user is located in 2. Turn on the light in that room 3. Turn off the light in other rooms
Extension	None

Name	**Search for User**
Use case	UC4
Primary actor	Sensors
Stakeholders	User, sensors
Precondition	UC3
Postcondition	The controller is told where the user is
Main scenario	1. Search for movement in one of the monitored rooms 2. Tell the controller where and if movement is detected
Extension	None

Name	**Detect User Return**
Use case	UC5
Primary actor	Sensor
Stakeholders	User, sensor(kitchen) and database
Precondition	The user is not in the kitchen and the stove is on
Postcondition	The timer is reset, and the event is saved on the database
Main scenario	1. Controller receives event “User reenters the kitchen” from UC1 2. Sensor(kitchen) senses movement 3. Timer is stopped 4. All alerting lights are turned off 5. Report event to server and database
Extension	None

Name	**Turn stove off**
Use case	UC6
Primary actor	Power plug
Stakeholders	Power plug, lights
Precondition	UC2 + 20 minutes has passed
Postcondition	The stove is off and the light in the kitchen is showing a colored light indicating the stove was left on for too long
Main scenario	− Turn off the stove − Turn off the alerting lights − Turn on light in kitchen indicating the stove was left on − Report to server and database
Extension	None

Name	**View Data**
Use case	UC7
Primary actor	User
Stakeholders	User & database
Precondition	None
Postcondition	The user has access to the information from the database
Main scenario	− The user opens a browser − The user enters the web page − The user can watch the information from the database
Extension	None

8.1.4. Non-functional Requirements

Listed beneath are the non-functional requirements of the system. These are what the system should contain and not what it should be able to do. These are as the non-functional requirements derived from the project description[18].

NFR ID	NFR name	Description
NFR1	Must support at least 5 rooms including kitchen	The system must be able to support identification of user presence in at least 5 rooms including the kitchen.
NFR2	Database	The system should have a database for logging information.
NFR3	Distributed user interface server and client	A distributed user interface server and client must be developed for personal computer, tablet, web, and/or smartphone, for accessing the "monitoring" data. The server should be based on web technologies, either NodeJS, PHP, ASP.NET or Blazor (.NET)

Table 2- Non-functional requirements.

8.2. Architecture and design specification

In this section we will cover the architectural and design specification that were derived from the functional and non-functional requirements in section 8.1. In this section we will define how the system will be structured, what components it should contain and how these components should interact with each other and the user.

8.2.1. Architectural design method

To design the architecture of the system we used Krutchens 4+1 model which was also mentioned in section 7. Methods. In the following sections we will look at each of the four main views and describe what we designed for each of them.

8.2.2. Physical view

In line with the description in section 7.4.5, we have defined what hardware and software components the system will consist of and how they interact with each other. This can be seen on Figure 8.

We see that the connections between hardware is shown by the lines, which also show the technology they are connected with. Some of the lines are directed, which means data is restricted in some cases. The connection between software components inside each hardware components will be described later in section 8.4.

[18] (Wagner, Week 1 - Introduction, 2022, p. 17)

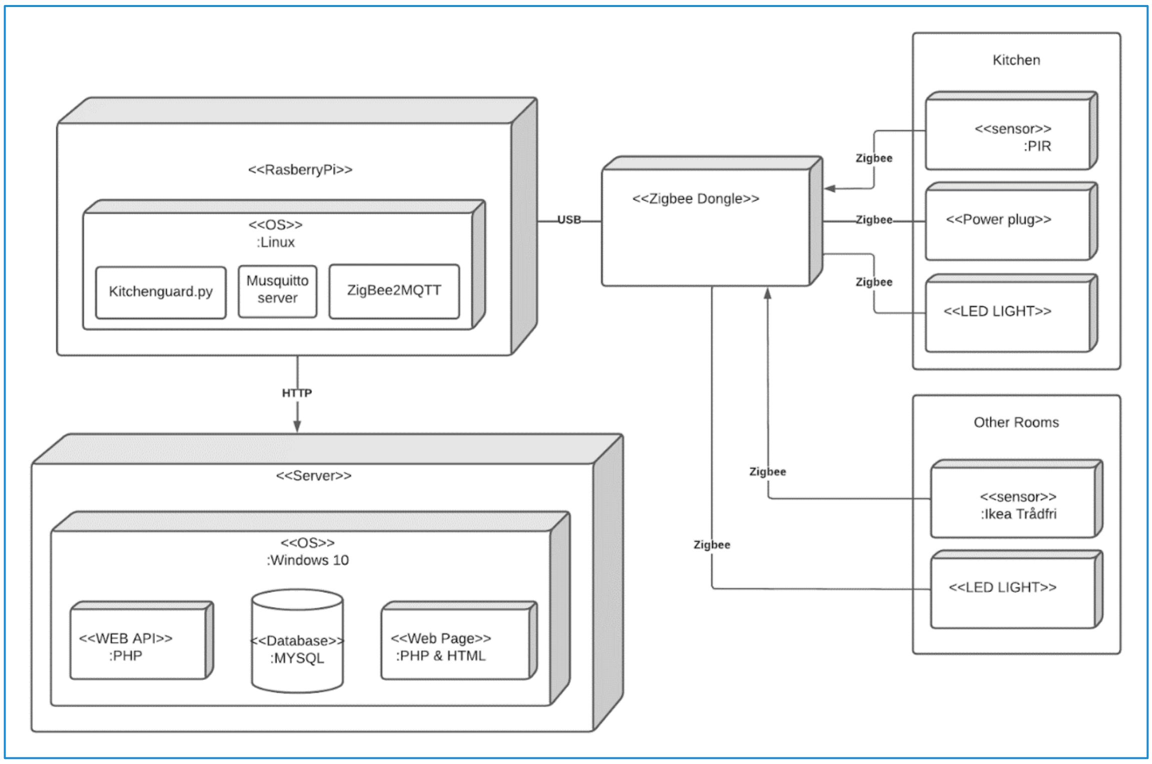


Figure 8:UML Deployment Diagram

In Figure 8 we also see the broker pattern that we described in section Design patterns Here zigbee2mqtt acts as the broker between actors and sensors and the controller.

A list of hardware components can be seen in the table below. The name, model, and quantity our system will support is listed for each item.

Components		
Component Name	Model	Quantity
Xiaomi Aqara Motion Sensor	RTCGQ11LM	Up to 5 (Collectively with Ikea sensor)
IKEA Trådfri Motion Sensor	E1525/E1745	Up to 5 (Collectively with Aqara sensor)
GLEDOPTO LED Light Strip	GL-MC-001PK	Up to 5
Immax NEO power unit Power Plug	07048L	1
Raspberry Pi4		1
Zigbee Controller USB Dongle	CC2531	1
Windows 10 Server	Any	1

Table 3: Components

Below is a list of the technologies used for the connections in the UML Deployment diagram and their full name.

Connections	
Name	Full name
HTTP	Hypertext Transfer Protocol
USB	Universal Serial Bus
Zigbee	

Table 4: Connections

For a detailed description of the components and their deployment can be seen in the Appendix.

8.2.3. Logical view

To give an overview of the functions and attributes of the different classes in the controller of the system we use a UML Class diagram as seen in Figure 9. It is a simple way to get an understanding of the different software components of the program.

We have based our controller implementation on the code given as part of tutorial 6[19]. This means that most classes have remained untouched, and we will therefore focus on the one class we did change. In the diagram below we can see that the functions and attributes marked with green are what we have created or changed the functionality of. In other classes, we have changed minor things like variables for addresses of the devices and web server.

To see what has and has not been changed, we refer to the top left box of Figure 9. Items marked with green (or grey if this is printed in greyscale) has been changed from the template or has been created by us.

The __zigbee2mqtt_event_received function has been changed and the rest of the “changed or created” functions/attributes have been created by us.

[19] (Wagner, Tutorial 6, 2022)

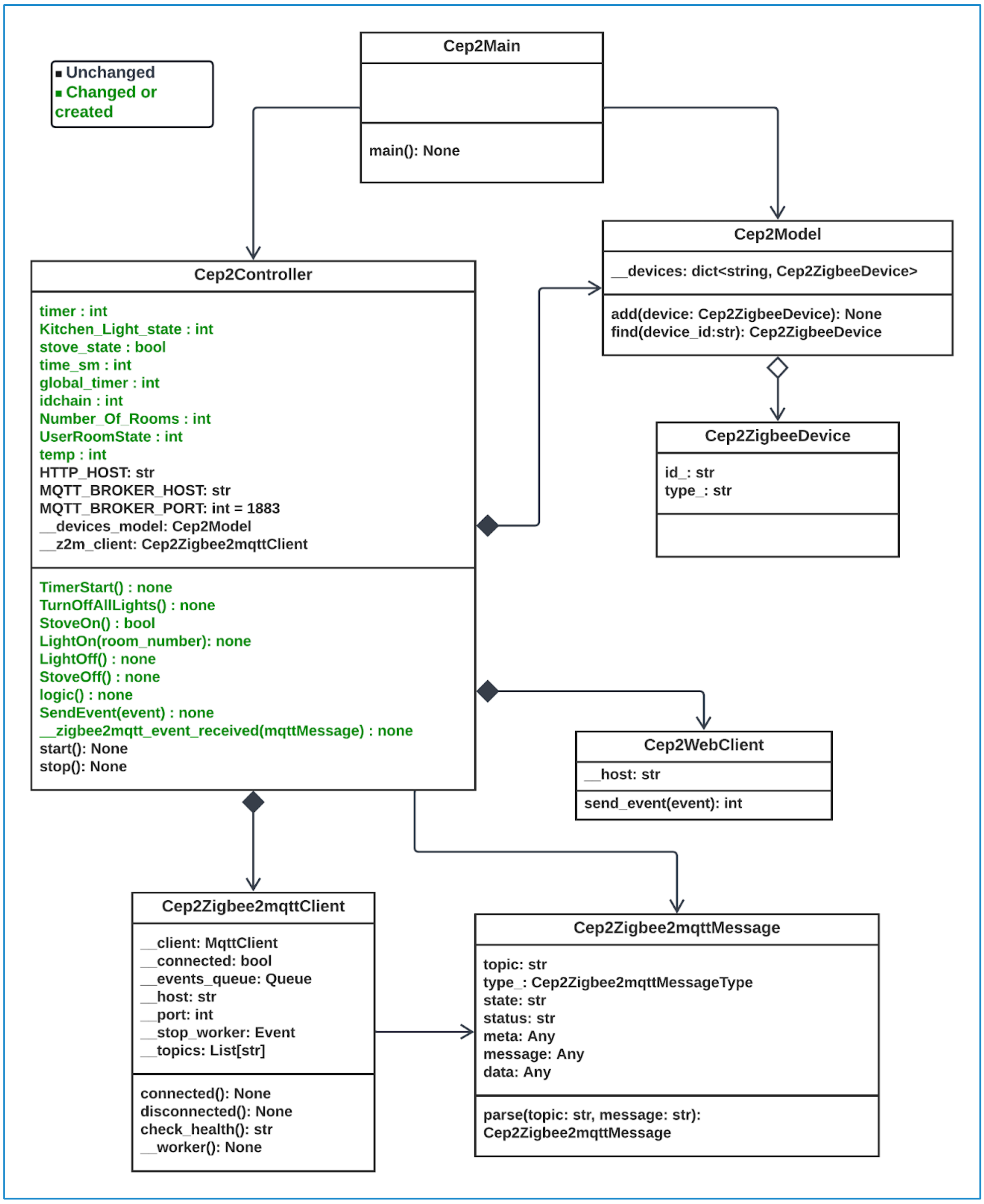


Figure 9: UML Class Diagram

In Figure 9 we also see the patterns disused in section 7.5. In terms of the model-view controller pattern we see a separation of model, view and controller. This is elaborated in tutorial 6[20]. We have also created a UML State diagram to show how these functions from the UML Class diagram are used.

[20] (Wagner, Tutorial 6, 2022)

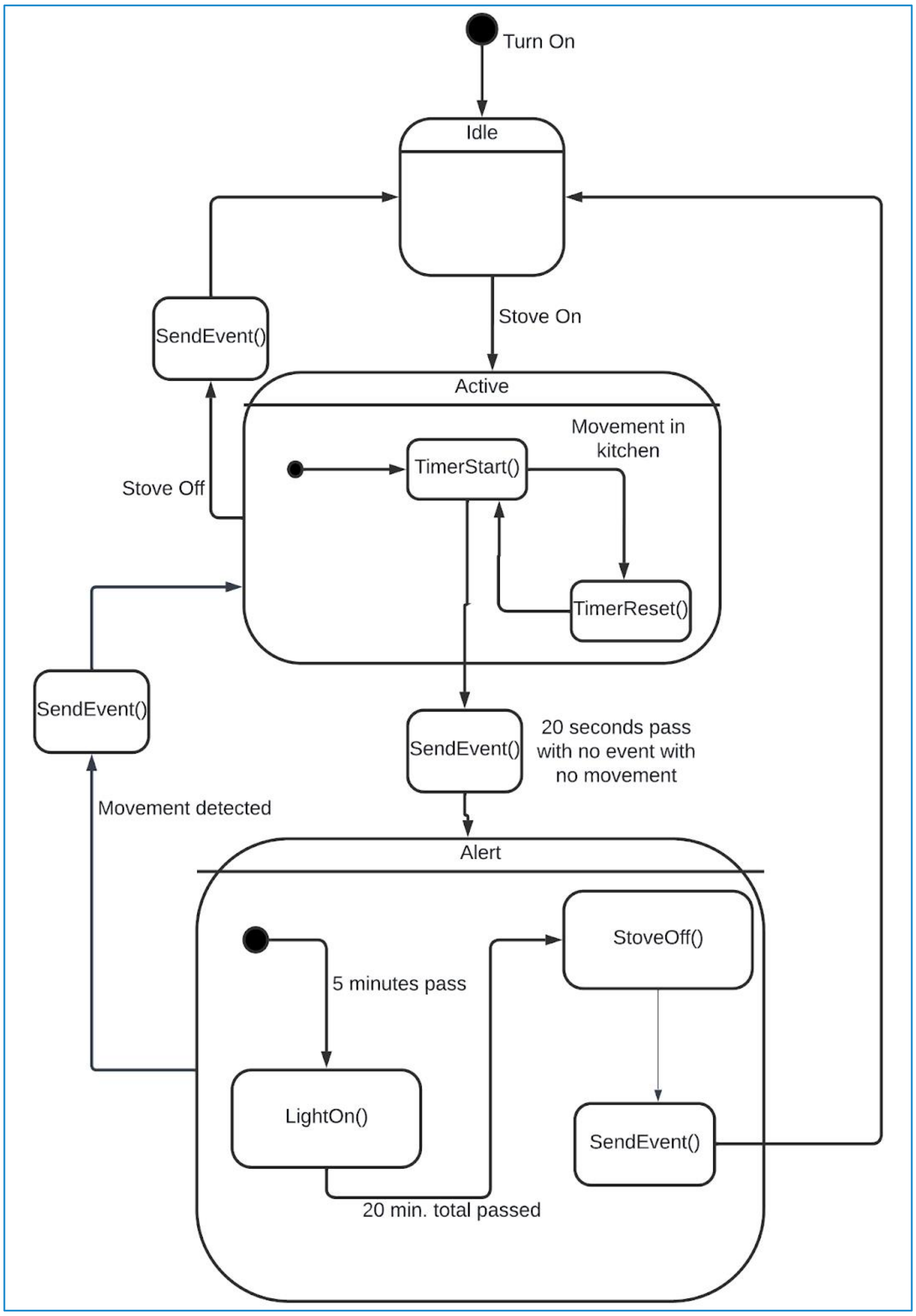


Figure 10: UML State Diagram

In the state diagram we see the three different states of the system. The system is idle when the stove is turned off. It then switches to active when the stove is turned on, from which it will monitor the kitchen for movement and reset a timer each time it receives confirmation that the user is in the kitchen. When the timer passes a certain threshold the system switches to alert state. Here the system detects that the user has left the kitchen and will alert the user with lights after 5 minutes and ultimately turn off the stove if the user does not return within 20 minutes. When leaving the alert state the system will log the data in the database for a third party to monitor.

8.2.4. Process view

To show the design of key functions in the program, we will use UML sequence diagrams, which show how the functions execute and how they call other functions depending on the state of the program. In the following UML sequence diagrams, we are seeing functions where classes would normally be in a sequence diagram. This is because the main functionality of the system is contained within the Kitchenguard class. To illustrate the actual functionality, we have changed the scope of the diagrams to focus on two functions within the Kitchenguard class and how they interact with other functions in the system.

On the following two pages we see Figure 11 and Figure 12, which show UML sequence diagrams of the logic and __zigbee2mqtt_even_received functions. The boxes tagged with “Alt” are alternative routes the function can take depending on the conditions stated in the top left of each “Alt” box. The lines indicate actions and only the ones named “function call” are function calls. This means that the text next to the curved lines is the actions performed by the function.

In Figure 11 we see the sequence diagram for the __zigbee2mqtt_event_received function. This function was part of the given code and has been changed. The unchanged part is shown in the beginning and only the last part of the function, which we have changed, is shown. In the diagram, we see how the function determines what component sent the event and determines what to do accordingly.

The diagram seen on Figure 12 handles the information the __zigbee2mqtt_event_received function has interpreted. It is responsible for turning off the stove and turning on the lights when needed. It also handles two timers, one small timer to determine if the user has left the kitchen and a larger timer to determine the time the user has spent away from the stove. This function is called periodically once every second, which means that every second it uses the latest information and can therefore make decisions using updated information.

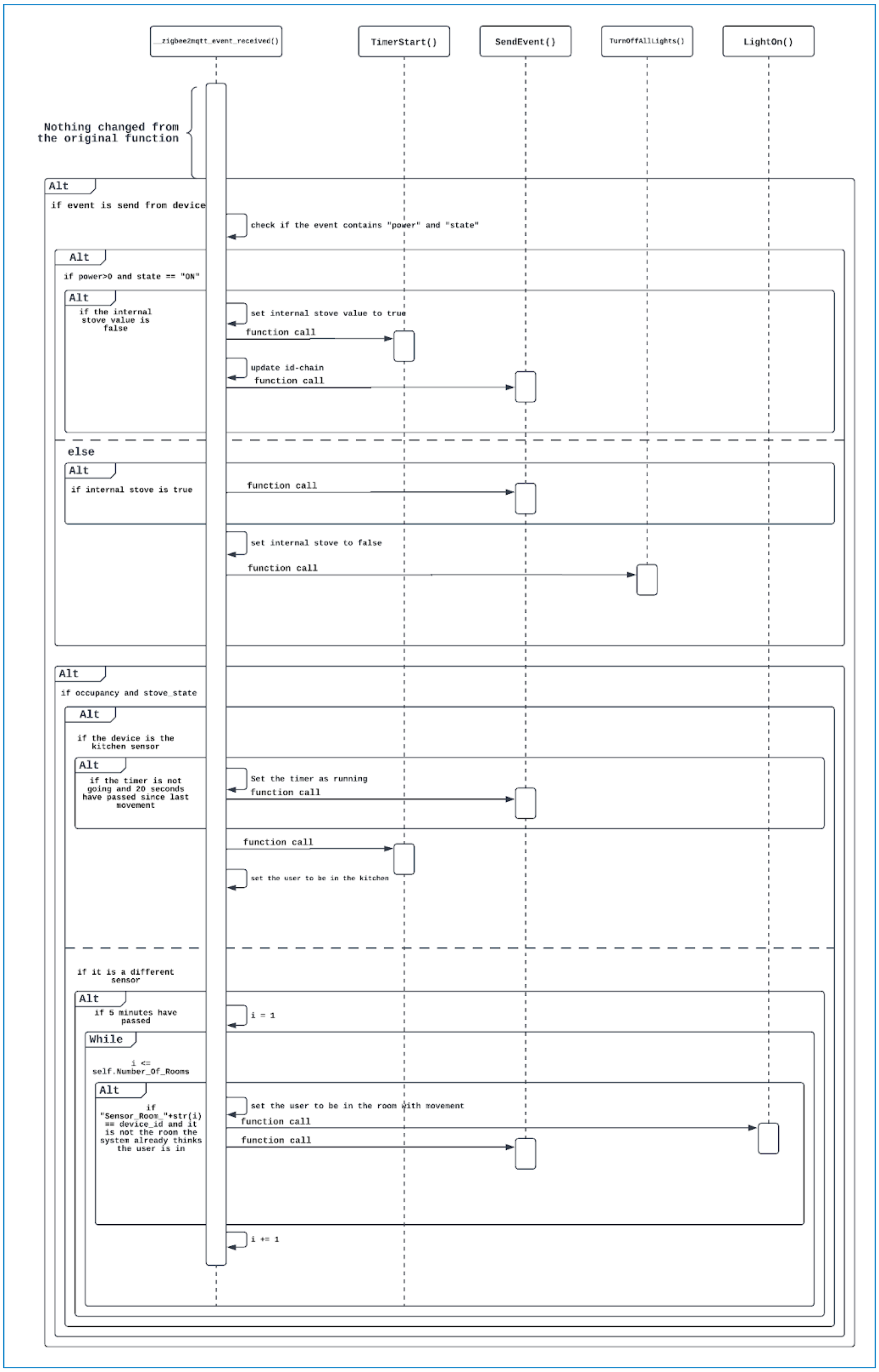

__zigbee2mqtt_event_received()
TimerStart()
SendEvent()
TurnOffAllLights()
LightOn()
Nothing changed from the original function
Alt
if event is send from device
check if the event contains "power" and "state"
Alt
if power>0 and state == "ON"
Alt
if the internal stove value is false
set internal stove value to true
function call
update id-chain
function call
else
Alt
if internal stove is true
function call
set internal stove to false
function call
Alt
if occupancy and stove_state
Alt
if the device is the kitchen sensor
Alt
if the timer is not going and 20 seconds have passed since last movement
Set the timer as running
function call
function call
set the user to be in the kitchen
if it is a different sensor
Alt
if 5 minutes have passed
i = 1
While
i <= self.Number_Of_Rooms
Alt
if "Sensor_Room_"+str(i) == device_id and it is not the room the system already thinks the user is in
set the user to be in the room with movement
function call
function call
i += 1

Figure 11: UML Sequence Diagram

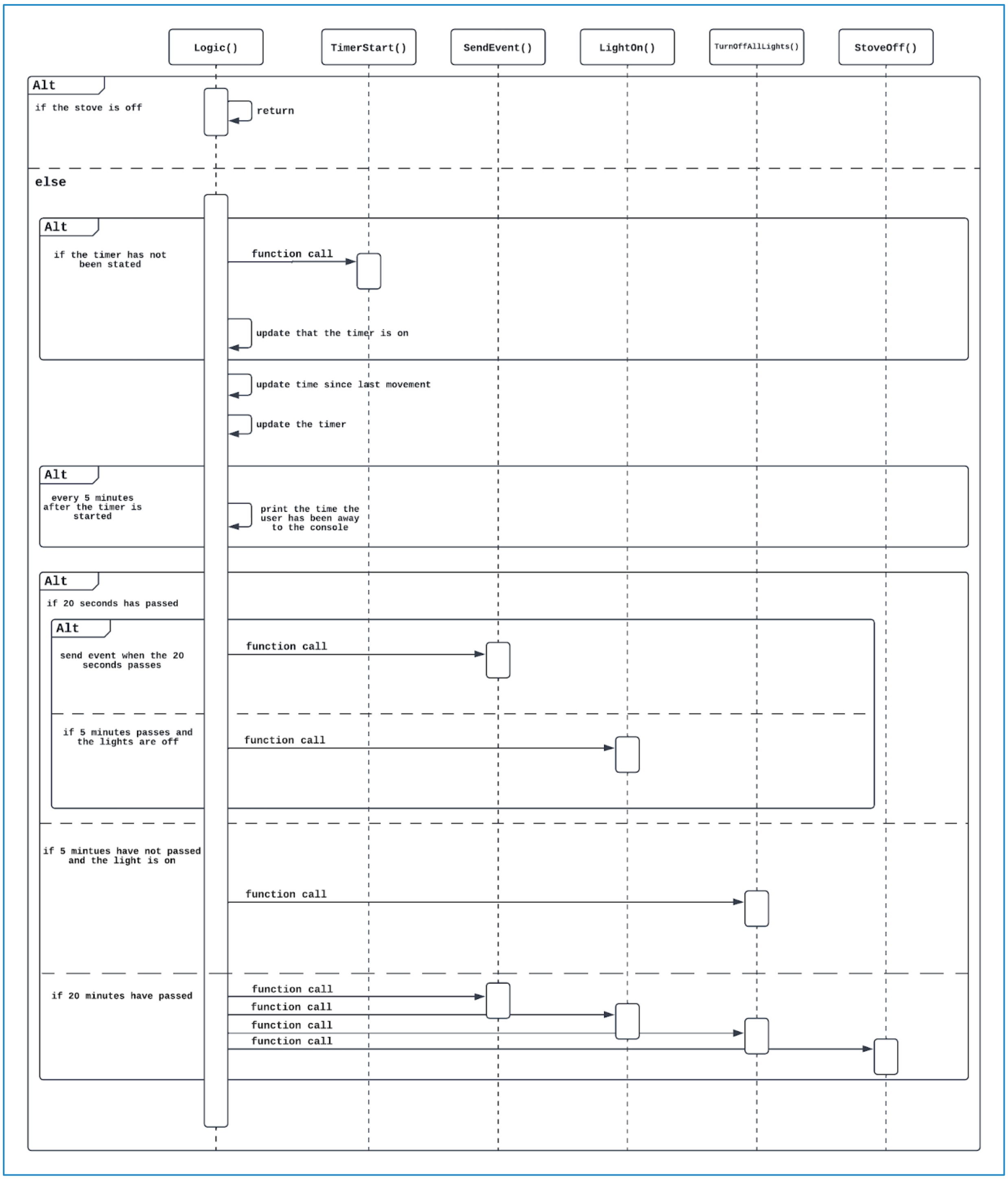

Logic()
TimerStart()
SendEvent()
LightOn()
TurnOffAllLights()
StoveOff()
Alt
if the stove is off
return
else
Alt
if the timer has not been stated
function call
update that the timer is on
update time since last movement
update the timer
Alt
every 5 minutes after the timer is started
print the time the user has been away to the console
Alt
if 20 seconds has passed
Alt
send event when the 20 seconds passes
function call
if 5 minutes passes and the lights are off
function call
if 5 mintues have not passed and the light is on
function call
if 20 minutes have passed
function call
function call
function call
function call

Figure 12: UML Sequence Diagram

To illustrate the runtime characteristics of the system we have created an UML Activity diagram that shows how the use cases have been used to structure the behavior of the system while in use.

For the activity to begin it is required that the stove is turned on. When this precondition has been satisfied, the controller will start monitoring for movement in the kitchen (see UC1). What this means is that the controller will wait for events received from the sensors. If it receives an event from the kitchen sensors within 20 seconds, it will continue monitoring. If it has not, however, it will start a timer (see UC2). If the user returns to the kitchen before the timer reaches 5 minutes, it will go back to monitoring the stove. If not, the controller will start waiting for other sensor events indicating which room the user is in (see UC4). If it does not receive any event, it simply does nothing, however, if it receives an event from one of the sensors in one of the rooms, the controller will alert the user by turning on an LED in the corresponding room (see UC3). If movement is detected in the kitchen before 20 minutes pass, the controller will go back to UC1. Throughout this activity flow chart, the controller will at some points send events to the web API, this is indicated by the hexagon figure (see UC7).

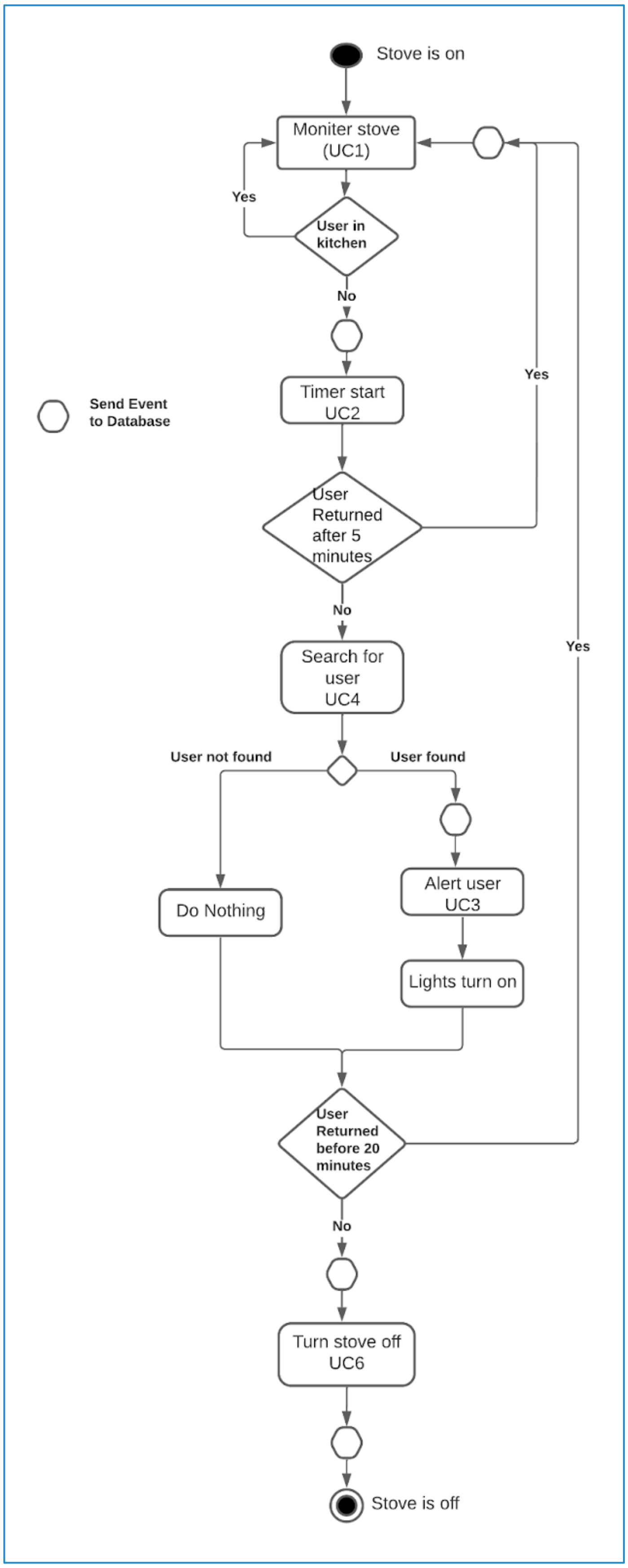


Figure 13: UML Activity Diagram

8.2.5. Development view

Error! Reference source not found. shows a UML package diagram of the software of the whole system. The dotted lines symbolize communication. The arrow shows directed communication. That means that Kitchenguard.py can communicate with the Web API but the Web API cannot communicate with the Kitchenguard controller.

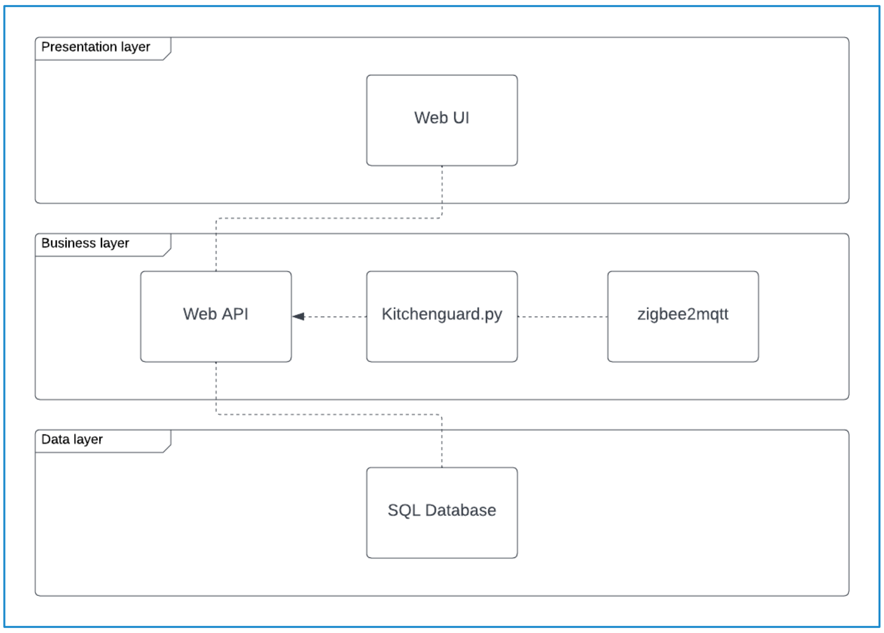


Figure 14: UML Package Diagram

8.3. Test Specification

The test specification describes how we designed and performed different test cases for our functional requirements described in the Requirement specification to view it refer to the Appendix. The system design centers around home use, with different visual cues throughout the home indicating that the stove has been left on and after an extended period of time, the stove will be turned off. The system will also store data for later review by the user on a web page.

An example of one of the test-cases can be seen below. To gain complete insight of the test specification refer to the Test specification Appendix.

Name	Monitor Stove
Use case	UC1
Primary actor	User
Precondition	Stove is turned on
Test input.	1. The user will wave his hand in front of the kitchen sensor to indicate movement in the kitchen.
Expected result	The controller receives an event, and prints it in the controller terminal.
Test functionality	To test if the sensor can send events to the controller and to test if the controller can receive said events.
Result (Pass/fail)	
Note:	

Table 5: Testcase 1 - Monitor Stove

In Table 5 we see TC1 Monitor stove which tests UC1. A closer inspection of TC1 also reveals a *Result* box. This box is the one that is later used to indicate whether a test has passed or failed the acceptance test described in 0

Acceptance Testing.

When developing the kitchen guard controller, component testing was used to verify that the logic, __zigbee2mqtt_event_received and SendEvent functions were working together with the Web API as intended. However, because of a delay when sending events to the web API using HTTP, there was a bug in the __zigbee2mqtt_event_received function where it would fail to finish executing because of the while loop running the logic function every second and thereby changing the state of the timer variables that the __zigbee2mqtt_event_received function used. This bug would have been near impossible to notice without proper testing, but because of the systematic testing utilized it was swiftly found and a fix implemented to ensure that running the logic function would not implicate the __zigbee2mqtt_event_received methods functionality.

8.4. Implementation Documentation

This section entails a detailed explanation of some crucial methods of our kitchen guard implementation. To view the full implementation documentation, refer to the Appendix and the implementation on git[21]. There are many comments in the implementation to increase readability, however, we thought it was necessary to explain certain parts in depth in this report as well.

The code blocks in this document will include line numbers on the left part of the code block and will also show the file's name at the top of the code block.

The explanation will cover some crucial methods and variables we have added to and modified from the Cep2 kitchen guard system received in tutorial 6[22].

We will also be covering our developed Web Application Programming Interface or Web API for short. This handles processing events sent from the controller, storing them in a database and displaying them in a graphical user interface in the form of a web client. It does this using the WAMP stack, which is an acronym for Windows, Apache, MySQL, and PHP.

8.4.1. The Kitchen Guard Controller

The event handling functionality of the kitchen guard system can be found in the controller. The controller consists of the python files running on the raspberry pi.

The first thing the controller waits for is for the stove_state to be True, as in the stove to be turned on. This is monitored in the __zigbee2mqtt_event_received function as seen below:

```
File: Kitchenguard.py
184:         if device:
185:             try:
186:                 #This checks if the event recieved includes the variable power, as in is the event from the plug.
```

[21] Schønberg, 2022: CEP2
[22] jmiranda-au, 2021: cep2app

```
            power = message.event["power"]
            state = message.event["state"]
            #If the device is not the kitchen plug then ignore the event. We only use that one plug.
            if device_id != "Kitchen_Plug":
                pass
        except KeyError:
            pass
        else:
            #If the plug is powering something as in the stove is on, check if it was off previously, if it was do the following.
            if power>0 and state == "ON":
                if self.stove_state == False :
                    self.stove_state = True
                    self.TimerStart()
                    self.idchain += 1
                    self.SendEvent("Stove Turned On")
            else: # Otherwise if the stove is off
                if self.stove_state == True: #check if it previously was on, if it was do the following.
                    self.SendEvent("Stove Was Manually Turned Off")
                    if self.global_timer < 1200:
                        self.LightOff(0)
                self.stove_state = False
```

The reason for both checking the power and state of the kitchen plug, is that when the stove is turned off it takes some seconds for the power output to reach 0. At the same time the state of the kitchen plug can be ON but if the stove is not powered on the power will remain at 0. So, the only time when the stove is truly turned on is when both the state is ON and the power is greater than 0. When the stove is turned on the logic function will start operating. The logic function is what changes the lights and kitchen plug depending on the value of the different timers. It also updates the values of the timers.

```
File: Kitchenguard.py
    def logic(self) -> None:
        if self.stove_state == False: # If the stove is off return
            return

        if self.temp != 0: # This is a failsafe used to handle the delay that sometimes comes with sending an event using HTTP
            self.TimerStart()
            self.temp = 0

        self.time_sm = time.time() - self.timer # The first timer
        self.global_timer = self.time_sm - 20 # The global timer
        """
        Used for testing purposes
        if round(self.global_timer) % 5 == 0 and self.global_timer >=0:
            print("User has left the kitchen for " + str(round(self.global_timer)) + " seconds.")
        """

        if self.time_sm >= 20:
            if  self.global_timer <= 1:
                self.SendEvent("User Has Left The Kitchen")
        """
        This part is for testing purposes
            if self.global_timer > 300 and self.Kitchen_Light_State != 1:
                self.LightOn(0)
        """
        if self.global_timer <= 300 and self.Kitchen_Light_State != 0: # This would be changed once we added more lights to the system
            self.TurnOffAllLights()

        if self.global_timer >= 1200: # If 20 minutes pass with no event recieved from the kitchen sensor with occupancy == true do the following
            self.TurnOffAllLights() # Turn off all alerting lights.
            self.LightOn(0) # Turn on the kitchen LED light.
            self.__z2m_client.change_color("Kitchen_Light",{"r":2,"g":0,"b":0}) # Make the light
```

```
red to signify that the user left the stove on for more than 20 minutes.
                self.SendEvent("User Has Not Returned After 20 Minutes") # Send an event to the web API
                self.StoveOff() # Turn off the stove
            return
```

The logic function often uses the method TurnOffAllLights, this methods implementation can be seen below:

```
File: Kitchenguard.py
    def TurnOffAllLights(self) -> None:
        i = 0 # Variable used to shift through all rooms.
        new_state = "OFF"
        while i <= self.Number_Of_Rooms: # Shift through the number of rooms in the house
            #self.__z2m_client.change_state("Light_Room_"+str(i), new_state) # This should work however is untested
            i +=1
        self.LightOff(0)
```

As the name suggests this method turns off all LED lights in the kitchen guard system. The functionality of the while loop is currently commented out because additional LED lights have not been connected. Because of this we are unable to test if this piece of code works as intended.

A snippet of a part of the __zigbee2mqtt_event_received method can be seen below:

```
File: Kitchenguard.py
            try:
                # Check if the event includes the variable occupancy, as in is it from a sensor.
                occupancy = message.event["occupancy"]
            except KeyError:
                pass
            else:
                # If there is movement and the stove is on
                if occupancy and self.stove_state:
                    #If the movement is in the kitchen
                    if device_id == "Kitchen_Sensor":
                        if self.time_sm > 20 and self.temp == 0:
                            self.temp = 1 # Used to handle HTTP delay
                            self.SendEvent("User Returned To The Kitchen")
                        self.TimerStart() # Resets the timer
                        self.UserRoomState = 0 # Used to indicate that the user is in the kitchen
                        #self.__z2m_client.change_color("Kitchen_Light",{"r":2,"g":5,"b":2}) # Used for testing purposes

                    elif device_id != "Kitchen_Sensor": # If the movement is not in the kitchen
                        if self.global_timer > 300: # and 5 minutes have passed
                            i = 1 # used to keep track of the rooms
                            while i <= self.Number_Of_Rooms: # Check which room the movement oc-cured in
                                if "Sensor_Room_"+str(i) == device_id and self.UserRoomState != i:
                                    #self.LightOn(i)
                                    #When the user enters another room the kitchen light will change colour. This is because we only have one light.
                                    #self.TurnOffAllLights()
                                    self.UserRoomState = i
                                    self.LightOn(0) # Here we would light up room i, but as we only have 1 led light this has not been implemented
                                    self.__z2m_client.change_color("Kitchen_Light",{"r":i+2,"g":i,"b":i+1}) # We instead change the color depending on i.
                                    self.SendEvent("User is in Room "+str(i)) # Send an event to the web api.
                                i+=1
```

This is where events from PIR sensors are handled.

The variable UserRoomState is used when the event received is not from the kitchen sensor. The value of the variable corresponds to the room the user is in. Running a while loop it checks if the sensor in any of the room's device id matches with the sender of the event. For further explanations refer to Appendix.

The SendEvent method, used in the logic and event receiving function, implemented can be seen below:

```
File: Kitchenguard.py
    def SendEvent(self, event) -> None:
        """
        This is for testing purposes
        print("Event("+event+") was sent to the database.")
        """
        # This is the code used to send a json package to the web API to store in the database.
        conn = http.client.HTTPConnection('kitchenguard.ddns.net')
        event_kitchen_occupancy = heucod.HeucodEvent()
        event_kitchen_occupancy.Timestamp = time.time()+7200
        event_kitchen_occupancy.Event = event
        event_kitchen_occupancy.IdChain = self.idchain
        data = event_kitchen_occupancy.to_json()
        conn.request('POST', '/a.php', data)
        """
        This is for testing purposes
        r1 = conn.getresponse()
        print(r1.status, r1.reason)
        while chunk := r1.read(200):
            print(repr(chunk))
        """
```

This method uses the heucod.py file, that was received in tutorial 9[23], to create a JSON string which contains three variables Timestamp, Event and IdChain. This JSON string is then sent using HTTP to the webserver 'kitchenguard.ddns.net' using the HTTP request method.

8.4.2. The Web API

The Web API is what handles storing events received from the controller in the MySQL database and displaying them on a web client as a GUI for the end user to see.

We will cover how the events are processed in a chronological order. Firstly, the implementation that receives the events and stores them in the database. This functionality can be found in the a.php file, which processes events received from the controller using HTTP. The important part of the file's functionality:

```
File: a.php
// Load the POST.
$data = file_get_contents("php://input");

// ...and decode it into a PHP array.
$jsondata = json_decode($data);

// Do whatever with the array.
//PrintObj($jsondata);

$servername = "localhost";
$username = "root";
$password = "cep2";
$dbname = "kitchenguearddb";

// Database connection
$conn = new mysqli($servername, $username, $password, $dbname);
// Insert data Query
```

[23] Jmiranda-au, 2022: cep2heucod

```
$sql = "INSERT INTO events ( Timestamp, Eventtype, IdChain)
VALUES ('$jsondata->Timestamp', '$jsondata->Event' , '$jsondata->IdChain')";
```

The method first receives the JSON string sent through HTTP by the controller. It then decodes the JSON string. Sets up variable parameters that correspond to our kitchen guard database, connects to the MySQL database, and using Structured Query Language, or SQL for short, inserts the data into the database. There are three variables or columns in the database that are created from data received through the controller, those being Timestamp, Eventtype and Id-Chain. Timestamp is the number of seconds that have passed since 1970 the 1st of January 00:00 UTC therefore it is an integer. Eventtype is a string that entails what event was received by the controller, it describes what the user has done. The IdChain variable is simply used to separate a ‘session’ of events, it is an integer that increments whenever the stove is turned off and on again.

The web client’s functionality can be found in the index.php file that handles both fetching from and displaying the database in the web client for the end user to see. The PHP part of the file can be seen below:

```
File: index.php
        <?php
            $servername = "127.0.0.1";
            $username = "root";
            $password = "cep2";
            $dbname = "kitchenguearddb";

            // Database connection
            $conn = new mysqli($servername, $username, $password, $dbname);
            // Check connection
            if ($conn->connect_error) {
            die("Connection failed: " . $conn->connect_error);
            }

            $sql = "SELECT Timestamp, Eventtype, IdChain, id FROM events ORDER BY id DESC";
            $result = $conn->query($sql);

            if ($result->num_rows > 0) {
            // output data of each row
            while($row = $result->fetch_assoc()) {
                $time = gmdate("Y-m-d H:i:s", $row["Timestamp"]);
                echo "<tr>";
                echo "<td>" . $row['id'] . "</td>";
                echo "<td>" . $time . "</td>";
                echo "<td>" . $row['Eventtype'] . "</td>";
                echo "<td>" . $row["IdChain"] . "</td>";
                echo "</tr>";
            }

            }
            else {
            echo "0 results";
            }
            $conn->close();
        ?>
```

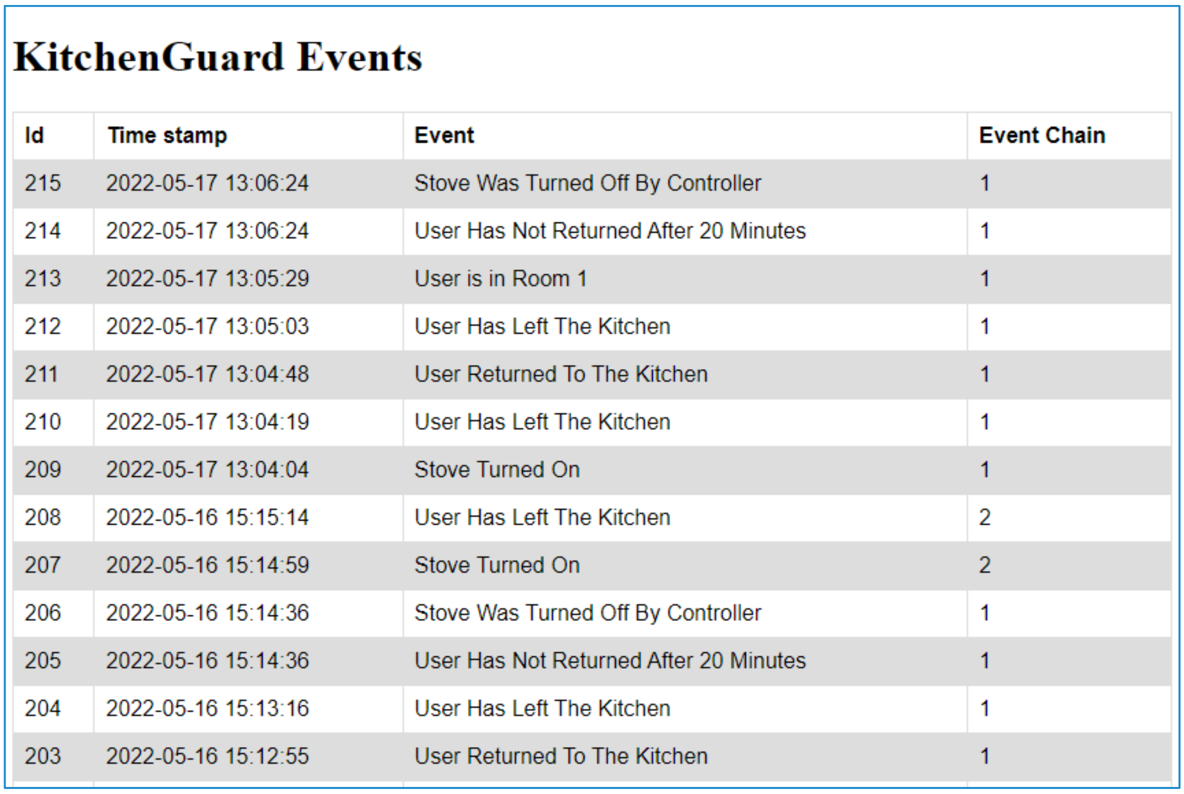
This PHP program also connects to the MySQL database using the database parameters. It then uses SQL to select the variables Timestamp, Eventtype, IdChain and id, from the table events in the database, ordered by the id in a descending manner. Here the id is simply an integer given to all events stored in the database to help keep track of what event is which. The results from this query are then saved and placed in an HTML table created prior, to be seen by the end user in the web client.

8.4.3. WAMP

Our database and webserver are both run using WAMP which stands for Windows, Apache, MySQL, and PHP. Note that the 'M' can also refer to MariaDB and the 'P' can also refer to Perl and Python. For our implementation we have chosen WAMP since we are running the server on a Windows 10 system, but this is not a must. A similar server could be set up using LAMP, which is for Linux or MAMP which is for MacOS.

8.4.3.1. The Database

Our database is an SQL database which runs on a MySQL server. The database is set up using the application ADMINER, which was used to set up the different columns of the database. The database data can be seen through the address Kitchenguard.ddns.net which is hosted using Apache. The database and web server are connected through the PHP scripts that are described in section 8.4.2. A view of the database can be seen in Figure 15, which displays the GUI web client:



KitchenGuard Events

Id	Time stamp	Event	Event Chain
215	2022-05-17 13:06:24	Stove Was Turned Off By Controller	1
214	2022-05-17 13:06:24	User Has Not Returned After 20 Minutes	1
213	2022-05-17 13:05:29	User is in Room 1	1
212	2022-05-17 13:05:03	User Has Left The Kitchen	1
211	2022-05-17 13:04:48	User Returned To The Kitchen	1
210	2022-05-17 13:04:19	User Has Left The Kitchen	1
209	2022-05-17 13:04:04	Stove Turned On	1
208	2022-05-16 15:15:14	User Has Left The Kitchen	2
207	2022-05-16 15:14:59	Stove Turned On	2
206	2022-05-16 15:14:36	Stove Was Turned Off By Controller	1
205	2022-05-16 15:14:36	User Has Not Returned After 20 Minutes	1
204	2022-05-16 15:13:16	User Has Left The Kitchen	1
203	2022-05-16 15:12:55	User Returned To The Kitchen	1

Figure 15: View of Database

8.4.3.2. The Web Server

The web server is used to store the data in the database and host our web page. This is hosted through the application Apache. The process of storing the data is done by the "a.php" file, which is called by the Kitchenguard python file's SendEvent function, as described previously. The website is also used to view the data from the database. Apache also hosts the web page, which's code can be found in the file "index.php".

8.5. Acceptance Testing

We did an acceptance test of our system to ensure that all the requirements were fulfilled. We did this by having a member of our peer group come by. The peer group member then was given a copy of our acceptance test to complete. The acceptance test was completed by the peer member without any nodes. The result of the test can be found in the Appendix.

9. Discussion

9.1. SCRUM

During this project we were using the SCRUM managing method. We did though deviate from the role structure in SCRUM in the way we did not have one person which filled the SCRUM leader role. Each member of the group was calling meetings and delegating assignments. During development we found this, to not be of great importance, which we think is mostly because of our small group size. Where in a larger group a single leader would perhaps be more essential.

9.2. Requirements Specification

During the requirement specification process, we deviated from the process in the way we did not have a specific client to discuss and construct the requirements with. Instead, we had the case description to follow, but a client to discuss what and how the requirements should be, could have given a different process all together.

In the requirement specification we did not take equipment failure or other general errors into consideration. This could be errors like sensor malfunction, wall plug malfunction or network errors. This is something we would have liked to include given another chance. This entails that our system can encounter an error with the equipment and therefore will operate in an unknown way. This means that we have not included all functionality of the system in the requirements, and the system is therefore not completely covered.

9.3. Architecture and design specification

During our design process we took advantage of design patterns that would suit our system well. This can be seen in section 7.5. Here we had a look at implementing the heartbeat pattern. The heartbeat pattern has all actuators and sensors send out a heartbeat signal in a given time interval to indicate that they have not failed. The heartbeat pattern would have helped us improve functionality of our system since it would help us detect whether an element in our system had failed. Something which we currently have no way of doing.

As seen in

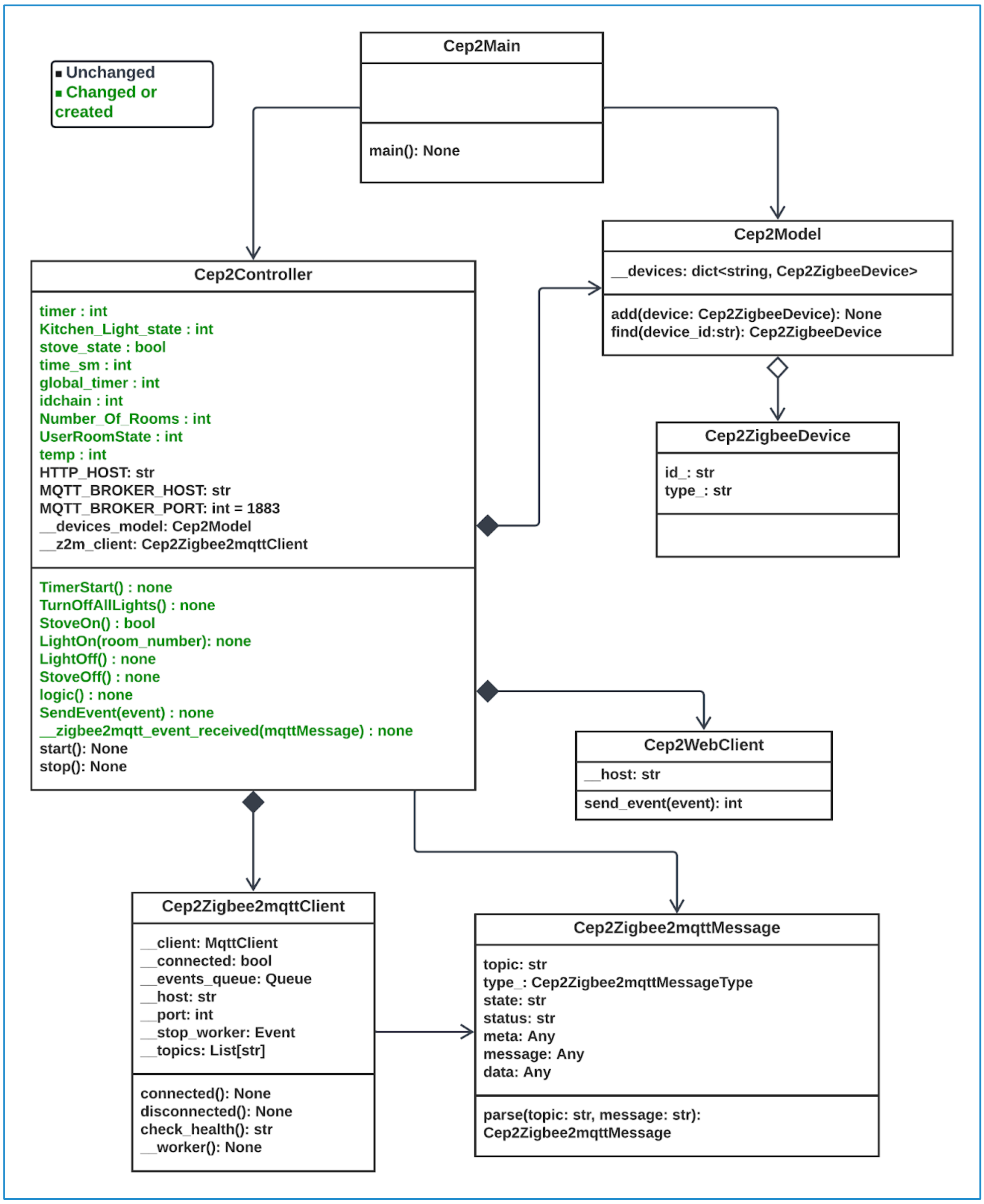


Figure 9 and as explained for the sequence diagrams, the class architecture of the system is not ideal. Almost all functionality is located in the controller class. This reduced the possibility of concurrency in our system and thus made our system slower. A solution would be to distribute the control flow by splitting the functionality into smaller classes that have their own responsibilities. An example of where we could have done this is the function calls to control the lights. These functions could have been moved to their own class that controlled the lights. Then the controller would only have to call the function and could then carry on with the logic. We could also have made a separate class for the timer, the stove and the event handler. Changing this would not make a big difference in this project since it is small, there are not any slow functions and the fact that the system does not get compromised by the small delays that could be caused by the lack of concurrency.

Staying on the topic of distributing functionality, we use a local MQTT server to communicate with the Raspberry Pi. This would mean that if we were to implement this system in multiple homes there would need to be a MQTT server in each home too. But if we moved the MQTT server to where our HTTP server is, then all local systems could communicate to the one single MQTT server and then the data would be saved on the database after. This would save money and energy by reducing hardware need for each system. From a functional perspective this would not change how the system works it would only change the technology used to send data to the server hosting the database and the hardware cost mentioned.

9.4. Test Specification

During the testing process we performed an acceptance test with another group which can be viewed in the Appendix. This acceptance test was performed in a similar environment as our video[24]. This meant the sensors were not tested in sperate rooms, and that we used a jacket to cover the sensor to represent the lack of movement. These things meant that the acceptance test did not properly reflect reality. The fact that it did not reflect reality meant that some tests maybe passed that should not have passed. It would be desirable to set up the system in environment that more closely reflected reality.

Another problem with having a testing environment that did not properly reflect the reality were that non-functional requirement 1 was not actually tested. Since we were unable set up our system with enough sensors and lights. This requirement would need to be tested in a better testing environment.

[24] (Schønberg, Acceptance Test Video)

9.5. Implementation

There were some parts of the implementation that had issues. For example, the controller logic function could have been implemented in a better manner. At line 117-119 a situation where more than one second has passed since last the timers were updated could occur leading to the first timer jumping from less than 20 seconds to above 21 seconds meaning the global timer would be above 1 second and therefore never satisfy the condition required to send the event. This did not turn out to be an issue practically, but the issue could lead to problematic situations. To avoid it one could implement some variable to signify what state the controller was in, using this variable one could change what functionality would execute in the different methods.

Because of lack of devices and components some parts of the design were not implemented as originally intended and also lacking the scalability desired. For example, the if statement in the logic function line 125-126 is not a part of the final implementation with the devices in all the rooms in the house connected. However, as no more than one LED light and two sensors were connected this is what was implemented. This could have been changed to what might work with all the devices. As it is not possible to test it, however, this would likely lead to many errors and bugs.

Additionally, there were some issues with a delay happening using HTTP, the reason why this happened was that it would usually take multiple seconds for an event to be sent using HTTP and likely because of the while loop running the logic function every second leading to the thread running the event received function to stop executing. The logic function would then change the timer values leading to the event received function not functioning as expected. This was made up for in the logic function, where the timer would be reset, and the temporary variable set to 0.

A general issue comes with the setup of the implementation. It is a complex process to set up the MQTT server, Zigbee adapter and configure all the devices and components correctly. The reason for this being that these ports, id’s and IP’s are all hard-coded manually. Without an expert available it is infeasible to setup. A solution to this could be to allow the user to add parameters upon startup given the specifications on the devices used. The reason for lacking this is also in part due to the implementation not being able to be run headlessly, as in without it being connected to a monitor. One could achieve a more systematic and automatic setup of the system if it was headless.

10. Conclusion

10.1. Objective 1: Requirements Specification

We have created a requirement specification which covers most of the functionality of the system using the requirements engineering process. In this we have constructed use case based requirements and ensured that these were testable for later test design. We have made a UML use case diagram showing the connections and dependencies between the use cases. As mentioned in the discussion, we could have included some equipment failure requirements which would have ensured full coverage of the system behavior. The requirement specification in its entirety can be seen in the Appendix. In conclusion objective 1 has been accomplished successfully.

10.2. Objective 2: Architecture and design specification

We designed the architecture of the system using the standards given by Kruchtens 4+1 model view as stated in the method section. We used the use cases from requirement specification as the scenarios and designed the system within the constraints given to us. All views were designed by closely following the standards given by the method even though we ended up making a few poor choices in regards to distributing control flow, scaling by having a local MQTT server and detecting component failure. The first two would have little to no impact on the functionality in terms of the goal of safety. Looking beyond these flaws, the architecture and design specification we created did allow us to implement a satisfactory system in the end. For the entire architectural and design specification refer to Appendix. In conclusion objective 2 has been accomplished successfully.

10.3. Objective 3: Implementation

The requirements-, architecture- and design specifications have been implemented as a system. This system consists of a kitchen guard controller made with python running using a MQTT server and a Zigbee adapter. The Web API is made with PHP, SQL and HTML running through the use Apache and MySQL. Events are sent and received across the controller and Web API with HTTP.

In this manner we have successfully managed to implement the designed and specified kitchen guard system, albeit with some issues such as lacking automated configuration and setup, as well as some suboptimized software. The entire implementation can be seen in the Appendix. In conclusion objective 3 has been accomplished successfully.

10.4. Objective 4: Test Specification

We tested the system by developing testcases from the use case based requirements that is described in section 8.1. These testcases can be seen in the test specification appendix. We then tested these testcases in an acceptance test which also can be seen in the Appendix. The test cases passed successfully without notes. However as described in section 9.4 the environment in which we tested this did not properly reflect reality. In addition to this the non-functional requirement 1, that we would want to test as well. These things however are not a major flaw, and we conclude that we were still able to test most aspects of the functionality of our system. In conclusion objective 4 has been accomplished somewhat successfully.

11. Sources

Jmiranda-au. (2021). *cep2app*. Retrieved from https://github.com/jmiranda-au/cep2app

Jmiranda-au. (2022). *cep2heucod*. Retrieved from https://github.com/jmiranda-au/cep2heucod

Lauritsen, E. H., Matzen, A. G., Michaelsen, M., & Schønberg, K. V. (2022). Requirements Specification.

Schønberg, K. V. (2022). *CEP2*. Retrieved from https://github.com/kevinschoenberg/CEP2

Schønberg, K. V. (n.d.). *Acceptance Test Video.* Retrieved from https://www.youtube.com/watch?v=2kqWTvpPbT0

Sommerville, P. (2016). *Software engineering 10th edition.*

Wagner, S. R. (2022). Totorial 9.

Wagner, S. R. (2022). Tutorial 6.

Wagner, S. R. (2022). Week 1 - Introduction.

12. Appendix

In the appendix folder the following documents can be found:

Requirement Specification

Architectural Design Specification

Implementation Document

Test Specification

Acceptance Test

Individual Contribution Documentation

Peer Review Received

Peer Review Performed