Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelor Thesis

Süleyman Berk Çemberci

Resource Aware Concept and Implementation of a PC Pool Status Indicator

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Süleyman Berk Çemberci

Resource Aware Concept and Implementation of a PC Pool Status Indicator

Bachelor Thesis based on the study regulations
for the Bachelor of Engineering degree programme
Information Engineering
at the Department of Information and Electrical Engineering
of the Faculty of Engineering and Computer Science
of the Hamburg University of Aplied Sciences

Supervising examiner : Prof. Dr. rer. nat. Henning Dierks
Second Examiner : Prof. Dr.-Ing. Lutz Leutelt

Day of delivery 15. Februar 2016

Süleyman Berk Çemberci

Title of the Bachelor Thesis

Resource Aware Concept and Implementation of a PC Pool Status Indicator

Keywords

PC Pool Status Indicator, Raspberry Pi, Resource Aware Concept and Implementation, Self-maintained

Abstract

Inside this thesis, the design process and implementation of the self-maintained PC Pool Status Indicator are described. The PC Pool Status Indicator, with its resource aware concept, is a software application running on a Raspberry Pi. It displays the availability of PC pools that are located on the 13th floor of Berliner Tor 7.

Süleyman Berk Çemberci

Titel der Arbeit

Ein Ressourcen schonendes Konzept und Implementierung einer PC-Pool Belegungsanzeige

Stichworte

PC Pool Belegungsanzeige, Raspberry Pi, Ressourcen schonend, Implementierung, Wartungsarm

Kurzzusammenfassung

In dieser Bachelor Theses wird der Entwurfsprozess und die spätere Implementierung einer wartungsarmen PC Pool Belegungsanzeige beschrieben. Die PC Pool Belegungsanzeige soll Ressourcen schonen und ist daher als Programm auf einem Raspberry Pi realisiert worden. Es zeigt die Verfügbarkeit der verschiedenen Räume des PC Pools im 13.ten Stockwerk im Berliner Tor 7 der HAW Hamburg an.

Contents

List of Tables

List of Figures

Listings

1. Introduction

On the 13th floor of Berliner Tor 7, there is a PC pool composed of 7 rooms which are available for either lectures and laboratories or individual study unless they are occupied. A lot of students are faced with the challenge of wandering around from room to room to find an available space, which causes disturbances to ongoing lectures and laboratories. In order to avoid these and direct students to the available rooms in a quick and efficient way, a new system is to be created. The PC Pool Status Indicator, composed of a Raspberry Pi and a display, is a self-maintained system that provides information on the availability of rooms located on the 13th floor of Berliner Tor 7.

In order to demonstrate the implementation of certain processes, the Unified Modeling Language - UML - diagrams are used throughout this thesis [22].

1.1. Requirements

As mentioned earlier, the PC Pool Status Indicator needs to display information regarding the availability of certain rooms; but in order to do so, it first needs to have access to that information. The necessary calendar files containing the availability descriptions of individual rooms are stored at an online location. Therefore, Internet access needs to be acquired first. Since the main goal is to achieve a self-maintained system, it is necessary for the PC Pool Status Indicator to be able to check its network connection status. In case of no Internet access, it needs to be able to establish Internet connection; which also makes it necessary for the system to be able to reset and restart its own hardware modules such as a wireless adapter. Having a wireless adapter is a necessity due to the location decided for the PC Pool Status Indicator, which lacks a network socket.

Assuming that the system has the ability to acquire its own Internet access by controlling some hardware modules, it would also need the ability to download the mentioned calendar files and store them at a specific location that is accessible offline, such as an internal memory. Once this is also made possible, the PC Pool Status Indicator would need to read the contents of the calendar files and parse the necessary availability information so that it can be displayed in a user friendly manner. Since these calendar files are updated frequently, they also need to be downloaded periodically.

Another requirement is that the system should have a long life span, meaning it should be easily customized when needed. If the calendar file locations are changed or certain rooms are renamed, it should have the ability to adapt to these changes easily. This can be achieved by having the ability to allow secure connections from remote to its internal memory, so that certain changes can be made when necessary.

Last but not least, the system should be low cost. This can be easily achieved through low power consumption and the acquisition of low priced hardware.

As an overview, all the aforementioned requirements can be listed as follows:

- Automated process of establishing Internet access by controlling a wireless adapter
- Ability to download calendar files periodically and to retrieve the necessary information by parsing them
- Acquiring a long life span by being able to adapt to changes easily
- Ability to allow secure connections from remote
- Low cost and low power consumption
- Ability to display room information in a user friendly manner

2. Analysis

2.1. Hardware

In order to keep the hardware costs low and avoid high levels of power consumption, it is decided to target a low power computer system; more precisely a barebone. Barebones are computer systems that contain only the components that are vital for the system to function, which keeps their cost and power consumption low [14]. Usually, barebones come with motherboards that have integrated graphics, which is crucial for this project considering the requirement of displaying room availability information on a graphical user interface.

Due to their cutting-edge price, computation power and very low power consumption, Beagle Bone Black (BBB) from Texas Instruments, Edison from Intel, CI20 from MIPS and Raspberry Pi model B revision 2 are considered for the PC Pool Status Indicator.

2.1.1. Raspberry Pi Model B Revision 2

Considering the barebones available on the market, Raspberry Pi - the credit card sized linux box, with its more than enough computation power - is by far the best seller and the most popular [31]. This may appear to be irrelevant in the decision making process of choosing a barebone; however, considering the impact its popularity has on finding necessary information about the device such as documentation and tutorials, it is quite important to have a wide user community. Only this aspect considered, with its enormous user community and unlimited number of open source projects, as well as documentation and tutorials, the Raspberry Pi is ahead of its competitors. Other aspects that are evaluated are power consumption and computation power offered in respect to the price.

Power Consumption

The size of its user community is, of course, not the only criterion on which the choice of the barebone is based; power consumption is also quite an important aspect. According to its official website, the Raspberry Pi model B revision 2 is recommended to be used with a 5V DC power supply of 1.8A current capacity [28]. Based on this recommendation only, assuming a constant full current draw at all times, the Raspberry Pi would consume 9 Watts at most (Equation 2.1).

$$P_{[W]} = V_{[V]} * I_{[A]} \tag{2.1}$$

Power calculation [13]

Clearly, the scenario above is only possible in theory; in practice, it is impossible for the Raspberry Pi to consume that amount of current. As demonstrated in Figure 2.1, the peaks of current draws with and without a WLAN dongle during boot up are 500mA and 400mA respectively.

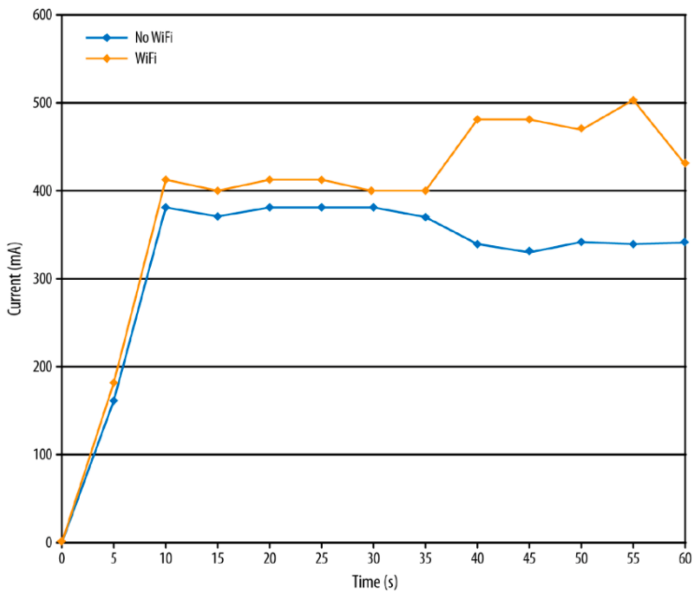


Figure 2.1.: Raspberry Pi model B revision 2, current consumption during boot up [21]

Considering the assumption of the device constantly drawing 500mA, which is the peak current draw with a WLAN dongle during boot up (Figure 2.1), from the equation 2.1 the power consumption would be 2.5 Watts. Comparing this to the average power consumption of a desktop computer - which is around a minimum median of 86.89 Watt at Idle state [27] - a barebone is indeed the right choice to keep the power consumption at a minimum.

Power consumption of Raspberry Pi model B revision 2 is compared to its competitors in Table 2.1. All the devices require a 5V DC power supply, hence the demonstrated power values.

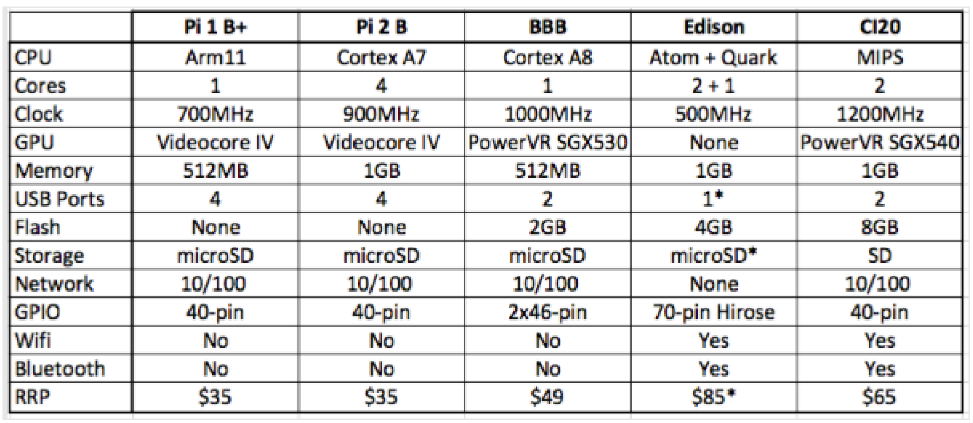
	IDLE Current	MAX Current	IDLE Power	MAX Power
Raspberry Pi [21]	380mA	500mA	1.9W	2.5W
Beaglebone Black [1]	310mA	520mA	1.6W	2.6W
Edison [3]	200mA	500mA	1W	2.5W
CI20 [6]	210mA	750mA	1.1W	3.8W

Table 2.1.: Power consumption comparison

As demonstrated, power consumptions of the devices are almost identical and very low at the same time.

CPU Performance and the Price

Since the Raspberry Pi has competitors that are able to match similar power consumption rates, the size of its user community is so far the only area where it provides significant difference. However, what is unique and cutting-edge about Raspberry Pi is its price together with the computation powered offered. A price comparison between Raspberry Pi and its competitors is demonstrated in Figure 2.2. Prices are shown in the bottom row and BBB stands for Beaglebone Black.



	Pi 1 B+	Pi 2 B	BBB	Edison	CI20
CPU	Arm11	Cortex A7	Cortex A8	Atom + Quark	MIPS
Cores	1	4	1	2 + 1	2
Clock	700MHz	900MHz	1000MHz	500MHz	1200MHz
GPU	Videocore IV	Videocore IV	PowerVR SGX530	None	PowerVR SGX540
Memory	512MB	1GB	512MB	1GB	1GB
USB Ports	4	4	2	1*	2
Flash	None	None	2GB	4GB	8GB
Storage	microSD	microSD	microSD	microSD*	SD
Network	10/100	10/100	10/100	None	10/100
GPIO	40-pin	40-pin	2x46-pin	70-pin Hirose	40-pin
Wifi	No	No	No	Yes	Yes
Bluetooth	No	No	No	Yes	Yes
RRP	$35	$35	$49	$85*	$65

Figure 2.2.: Raspberry Pi price comparison in respect to offered hardware specifications [16]

Just by looking at the prices, Raspberry Pi model B revision 2 is indeed the cheapest; but this does not make it inferior when it comes to the computation power offered. As shown in Figure 2.2, it is shipped with 1GB of RAM and an ARM Cortex-A7 quad core processor clocked at 900Mhz [2]. A benchmarking test known as *Sysbench*, is performed on all the competitors shown in Figure 2.2. Basically, *Sysbench's* CPU test verifies prime numbers by dividing each number by all numbers between 2 and the square root of the number. If any number gives a remainder of 0, it continues with the next number, until it reaches the maximum number defined by the user [11]. With this in mind, Figure 2.3 demonstrates the results of *Sysbench* test with the maximum number set to 20000.

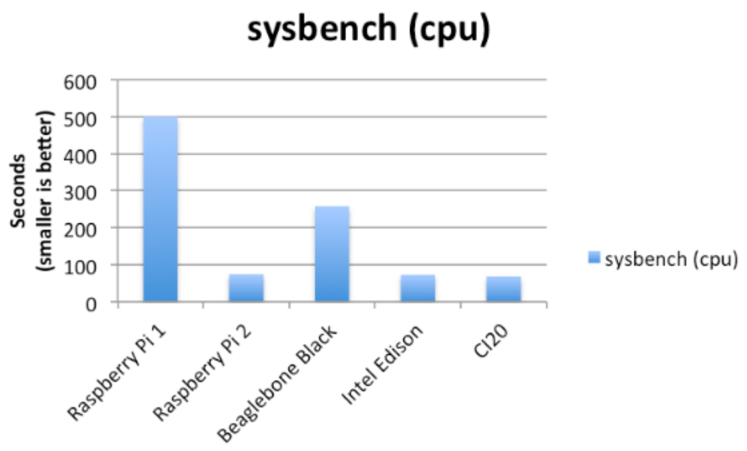


Figure 2.3.: Sysbench results of Raspberry Pi and its competitors [16]

For the requirements of this project, the quad core ARM Cortex-A7 - which is able to verify prime numbers up to 20000 in less than 100 seconds - is proven to offer the required computation power necessary.

Considering the facts demonstrated, with its cutting-edge price, computation power and very low power consumption, Raspberry Pi model B revision 2 is decided to be used for the PC Pool Status Indicator.

2.1.2. WLAN Dongle - RT5370 Chipset

The decision on the Raspberry Pi makes the decision on the WLAN dongle easier. Ralink RT5370 Chipset is shown to be working on the Raspberry Pi revision 2 out of the box [9]. Considering this fact, the decision is made to purchase a LB-LINK wireless USB adapter with a Ralink RT5370 Chipset [18].

Unfortunately, due to the WLAN Dongle not being well known, a detailed datasheet containing information regarding its power consumption in terms of current draws could not be found. For this reason, the linux bash command *lsusb* is used. When run with the argument *-v*, the command *lsusb* lists all the USB devices detected by the operating system with their determined characteristics [30]. From the stdout of the *lsusb -v*, the defined maximum current draw for the RT5370 wireless adapter is found out to be 450mA (the complete stdout of *lsusb -v* can be found in Appendix A). Even though the maximum current draw of 450mA does not seem high, the default allowed maximum total USB peripheral current on the Raspberry Pi 2 model B of 600mA proves otherwise [28]. 600mA are set to be distributed among the four USB ports of Raspberry Pi, which increases the likelihood of the WLAN dongle not receiving the required amount of current. Due to these facts, this limit is changed from its default value to the maximum allowed 1.2A, so that the WLAN dongle gets the current it needs at all times.

2.2. Software

Once the hardware decisions are made based on the knowledge gained through hardware analysis, software decisions can be made and the possible outcomes can be analyzed.

2.2.1. Operating System - Raspbian GNU/Linux 8

Regardless of its hardware based capabilities, without an operating system to drive its hardware, Raspberry Pi is nothing but a circuit board with a piece of silicon on it. Fortunately, due to its wide user community and the enormous development support, there are several options when it comes to deciding for an operating system.

Based on Debian - which is one of the original Linux distributions - Raspbian is an open source operating system modified specifically for the Raspberry Pi [15]. Due to it being the recommended operating system for the Raspberry Pi, as well as it being shipped with Java and Python installed, the decision is made upon Raspbian. In order to keep things up to date and have a stable system with the most current bug fixes, the latest available version of Raspbian is used. *Raspbian Jessie*, also referred to as *Raspbian GNU/Linux 8*, released in November 2015, is the latest version available.

Java and Python

Raspbian already comes preinstalled with two very important programming languages for this project, Java and Python. By executing *java –version* and *python –version* in the terminal of the Rasbian, it is discovered that the versions are *1.8.0* and *2.7.9* respectively. The Java version *1.8.0* corresponds to JDK8 and JRE8, which are up to date [25]. As any unix based operating system, Raspbian is capable of running several different scripting laguages, such as bash, tcsh etc. However, since dealing with a more advanced, object-oriented programming language is easier than programming scripts based on bash or tcsh, Python is decided to be used. Therefore the decision is made upon using Python as the main scripting language responsible for performing certain tasks in the background, with Java as the programming language for the user interface.

Secure Shell - SSH

In order to fulfill the requirements that are mentioned in Section 1.1, remote connections need to be granted to the system. To allow these connections to the Raspberry Pi, the Raspbian operating system comes with a prebuilt SSH server [29]. The easiest way of granting secure connections to the device is to configure and use this prebuilt server. Therefore the decision is made upon using SSH.

Cron Command Scheduling

As in any Unix based operating system, Raspbian is also capable of running cron jobs. Cron jobs are processes or commands that are executed by the operating system at designated times defined by the user [17]. As previously mentioned in the requirements (Section 1.1), the system requires the ability to check its Internet connectivity and establish WiFi connection, as well as download certain files periodically. Since the operating system is already capable of running scheduled commands by cron jobs, instead of trying to schedule these tasks on the software level by timers or timer equivalent processes, it is decided to use cron jobs.

WPA Supplicant

In order to automate the mechanism of establishing Internet connection, a very basic command line based network utility is required. One of the preinstalled programs on Rasbian, called *wpa_supplicant*, is able to control wireless adapters and achieve exactly what is needed [19]. The main purpose of this open source program is to run in the background as a *daemon* and control the wireless connections. It is able to extract its stdout into a file when needed, as well as read wireless network configurations from a file defined by the user. By the help of the mentioned configuration file option, *wpa_supplicant* also allows the wireless settings to be easily changed when desired. Therefore it is a great fit to achieve longer life span for the system, allowing it to adapt to changes in the wireless network. Therefore, in order to avoid searching for an alternative that might fit the mentioned requirements, the decision is made upon using the wpa_supplicant.

2.2.2. Java GUI

As mentioned earlier, Java is decided to be the programming language for the graphical user interface. Following this decision, a GUI development library needs to be selected. In order to avoid downloading and installing additional packages, Swing and AWT libraries can be used. These libraries are present in the Oracle JDK which is preinstalled in Raspbian.

There are certain things that are to be displayed on the GUI:

- Floor plan of the PC Pool
- Names and labels of each room
- Emergency exits on the floor
- Current date and time
- Available or occupied status of the seven rooms of interest

In order to start with this task, a floor plan needs to be drawn. In addition, the room labels on the 13th floor need to be collected manually so that they can be displayed on the GUI. In order to mark the emergency exits in a nice way, an icon for these exits need to found. Since it is safe to assume that the location of the restrooms on the floor is already known by the students and since this area is more or less located right in the center of the floor plan, the restroom area can be used to place the current date and time information instead. At the locations where the seven rooms of interest are placed, a basic schedule information should be shown. This can be done in a way that the current ongoing event and its remaining time are displayed together with the upcoming events, including their start and end times.

Additionally, the final GUI should be realized in full screen mode so that the entire screen is dedicated to the application. This way, a more professional look can also be established.

2.2.3. Python Scripts

As mentioned earlier, there are some tasks that need to be executed periodically by the system. These tasks are as follows:

A. Checking Internet connectivity and establishing WiFi connection

B. Downloading ICS files of the rooms of interest, keeping the availability information up to date

C. Checking the list of running processes for the Java GUI and starting it if not found

In order to achieve a periodic execution of tasks A and B, cron command scheduling is used. The easiest way to schedule tasks using cron jobs is to create them as executable scripts and therefore the decision is made to realize the mentioned tasks in Python.

On the other hand, task C needs to be realized with a shorter period compared to the other tasks. The reason behind this is that if the Java GUI crashes or quits for some reason, it is better to restart it as soon as possible. Therefore, the mentioned task needs to have a period in seconds and since the smallest form of a period for a cron job is in minutes, a different approach needs to be taken [17]. Therefore, instead of it being a cron job, task C should be realized in the form of a Python wrapper, responsible for starting or restarting the Java GUI when necessary.

2.2.4. ICS File Format

The availability information of the PC Pool rooms of interest is accessible in the form of ICS files. Therefore, the system should be capable of parsing the mentioned files and getting the necessary availability information. To do so, a parsing mechanism needs to be created. An example of a certain event from an ICS file of interest is demonstrated in Listing 2.1.

```
BEGIN:VEVENT
SUMMARY:B-EE1-PR1 Tutorium
LOCATION:1381  Stand 01-12-2015
UID:151201135425.100962@etech.haw-hamburg.de
DTSTART;TZID=Europe/Berlin:20151028T155500
DTEND;TZID=Europe/Berlin:20151028T172500
END:VEVENT
```

Listing 2.1: Event definition from the ICS file of the room 1381

As can be seen in lines 1 and 7, an event in an ICS file is surrounded by "*BEGIN:VEVENT*" and "*END:VEVENT*". Therefore, the system should be able to detect occurrences of these keywords and mark them as the beginning and end of an event respectively. Each event has a descriptive field marked with the keyword "*SUMMARY*", which explains what the event is about (Line 2). The mentioned field is also of interest, since the Java GUI needs to display the name of the lecture or laboratory taking place if the room is occupied. Therefore, the system needs the ability to mark occurrences of the keyword "*SUMMARY*" within an event description. Since each room has its own ICS file and the ICS file creation date is irrelevant, line 3 only contains redundant information and can be skipped. A similar situation is valid for line 4 as well; the UID of the event is not of interest and therefore line 4 can be ignored. Lines 5 and 6, which are marked with the keywords "*DTSTART*" and "*DTEND*", are of high interest. These are the fields defining the start and end of an event with the date format "*yyyyMMdd'T'HHmmSS*". Therefore, once again the system should be capable of retrieving this information by the occurrences of the aforementioned keywords.

3. Design

Following up on the hardware and software analysis, there are certain things to be covered in the detailed description of the design process:

- Operating system installation and configuration
- WLAN dongle setup and creation of the *wpa_supplicant* configuration file
- Detailed descriptions and work flows of the Python scripts
- Java GUI design including layout and class hierarchy
- Project directory setup and overview

3.1. Raspbian Installation & Setup

In order to install Rasbian and set it up, the latest Raspbian image needs to be obtained first. For this reason, the Rasbian Jessie image with the following credentials is downloaded:

- Version: November 2015
- Release date: 2015-11-21
- Kernel version: 4.1
- SHA-1: ce1654f4b0492b3bcc93b233f431539b3df2f813
- Download URL: https://www.raspberrypi.org/downloads/raspbian/
- Home URL: http://www.raspbian.org
- Support URL: http://www.raspbian.org/RaspbianForums
- Bug report URL: http://www.raspbian.org/RaspbianBugs

The SHA-1 key shown in the image credentials above is obtained from the download URL. The purpose of this key is to uniquely identify the image to be downloaded, so that it can be verified if the retrieved image is indeed the one which is displayed on the download page. Therefore, before installing the downloaded image, its SHA-1 is obtained by the Linux command demonstrated in Listing 3.1 and compared to the one from the download URL. The command *openssl sha <path-to-image>* runs the software *openssl* and with the help of the argument *sha*, commands it to create the SHA-1 key of the given image [23].

```
> $ openssl sha 2015−11−21−raspbian−jessie.img
> SHA(2015−11−21−raspbian−jessie.img)=
    ce1654f4b0492b3bcc93b233f431539b3df2f813
```

Listing 3.1: SHA-1 key of the downloaded image created by a Linux terminal

As demonstrated, SHA-1 keys are matching. Therefore, the image is marked as safe and it is to be installed on the Raspberry Pi.

3.1.1. SD Card Partitioning and OS Image Installation

Since the Raspberry Pi does not have any internal memory, the only way to boot it from a disk is to use an SD card. For this purpose, a MicroSDHC card of 16GB is used.

In order to be able to boot Raspbian from the SD card, it is sufficient to write the image without changing its contents. Therefore, the Linux command *dd* [17] is used. Using this command to write the image to the SD card requires the device name of the SD card to be known in advance. Additionally, before the execution of *dd*, the SD card should be unmounted so that the writing process can proceed safely. For these purposes, the *df* Linux command is used. Once executed, this command lists all the mounted drive partitions available [17]. By executing *df* before and after connecting the SD card (Listings 3.2 and 3.3 correspondingly), the device name of the card is discovered to be */dev/sdb*.

```
> $ df
> Filesystem     1K−blocks     Used  Available  Use% Mounted on
  /dev/sda5      103075104 19744516   82265692   20% /
  ...
  /dev/sda7      681071328  5723612  673947208    1% /home
```

Listing 3.2: Stdout of *df* before connecting the SD card

```
> $ df
> Filesystem      1K-blocks      Used  Available  Use% Mounted on
  /dev/sda5       103075104  19744516   82265692   20% /
  ...
  /dev/sda7       681071328   5723612  673947208    1% /home
  /dev/sdb1        15052040      1648   15050392    1% /media/boot
```

Listing 3.3: Stdout of *df* after connecting the SD card

As demonstrated, instead of */dev/sdb1*, */dev/sdb* is taken as the device name. The reason behind this is that the */dev/sdb1* points to the mounted partition of the drive */dev/sdb*. Since the target is not just a partition of the SD card, but the SD card itself, device name is found to be */dev/sdb*.

Once the device name is determined, its mounted partition is unmounted by the Linux command demonstrated in Listing 3.4.

```
> $ umount /dev/sdb1
```

Listing 3.4: Unmounting mounted partition of the SD card

After the partition is unmounted, the writing process can start with the *dd* command. This command expects 3 arguments in order to be able to write the given source image to the given target disk. These arguments are as follows:

- bs - used to set the block size of the writing process. Defined as read and write up to the number of bytes at a time: *bs=<#bytes>*
- if - used to set the source path, in other words the path of the image to be written. Defined as: *if=<image-path>*
- of - similar to the *if* argument, used to set the target path, in this case the path to the SD card: *of=<SD-card-path>*

The resulting *dd* command and its execution to write the downloaded image to the SD card is demonstrated in Listing 3.5.

```
> $ dd bs=1M if=2015-11-21-raspbian-jessie.img of=/dev/sdb
>  3752+0 records in
   3752+0 records out
   3934257152 bytes transferred in 794.670148 secs (4950805 bytes/
      sec)
```

Listing 3.5: Execution of the dd command and its stdout

There is one additional thing worth mentioning about the *dd* command shown in Listing 3.5. From the documentation on the SD card setup for Raspberry Pi [7], it is discovered that the block size of 4M usually works; however the documentation suggests using 1M when it does not work. Therefore, in order to avoid a possible repetition of the process, block size 1M is used instead of 4M.

Once the writing process of the image is done, the SD card is plugged into the Raspberry Pi. This also marks that the Raspberry Pi reached a bootable state (Figure 3.1).

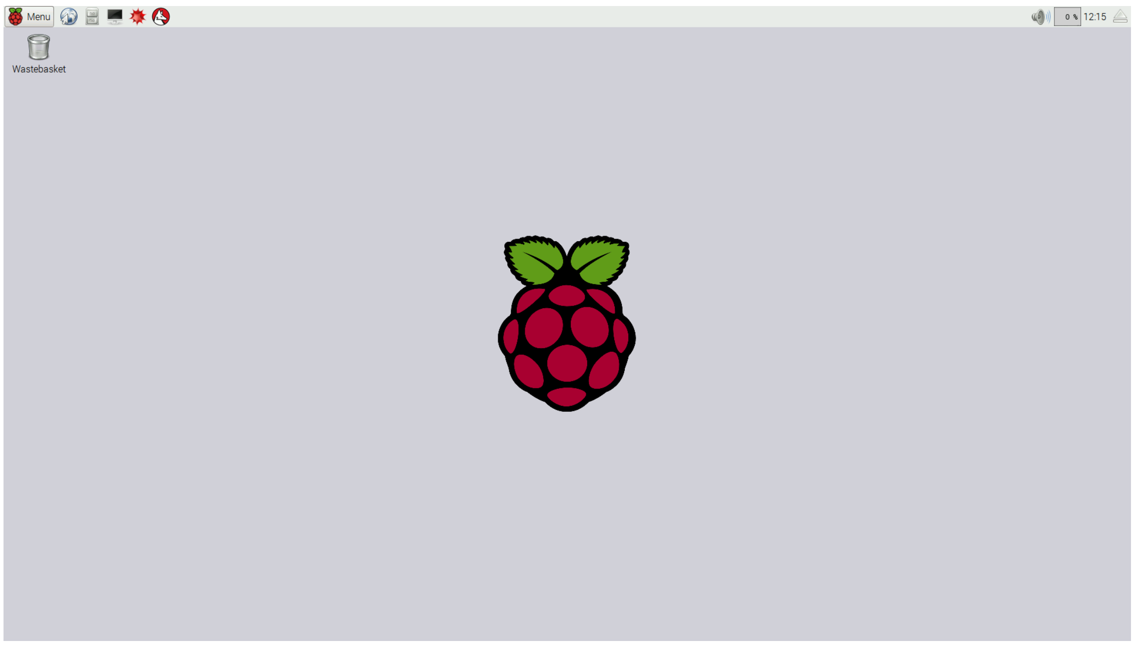


Figure 3.1.: Raspbian desktop view from its first ever boot

3.1.2. Configuration

Once the installation process of the Raspbian OS is done, it is configured according to the needs of the system.

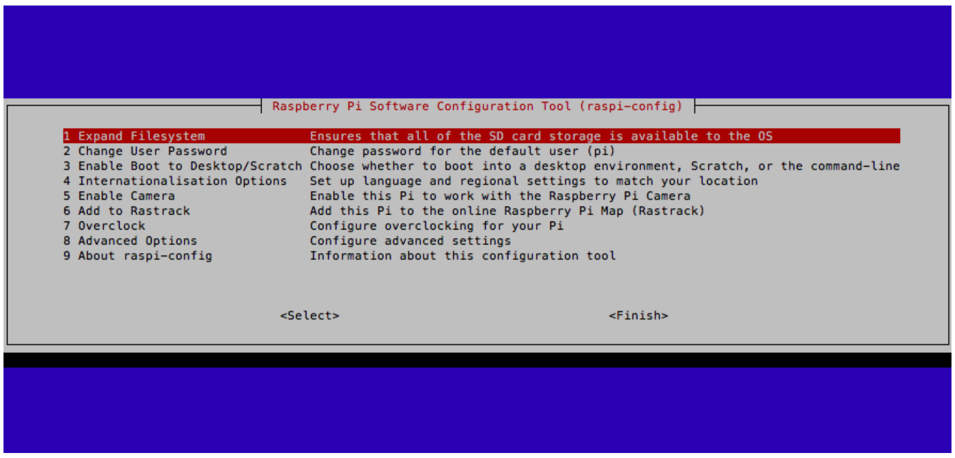
Secure Shell - SSH

As previously mentioned in Section 2.2.1, remote connections to the system can be securely granted through an SSH server. In order to do so, the SSH server needs to be enabled and for this purpose the raspi-config tool is used. Raspi-config, which is a bash script in its core, is designed to configure several operating system related settings on the Raspberry Pi [8]. By executing the command demonstrated in Listing 3.6 the tool is started.

```
> $ sudo raspi−config
```

Listing 3.6: Command to start raspi-config

Since the tool is capable of changing operating system related settings, it requires root priviledges, hence it is executed with *sudo* [17]. Once the raspi-config is started, the window demonstrated in Figure 3.2 is displayed.



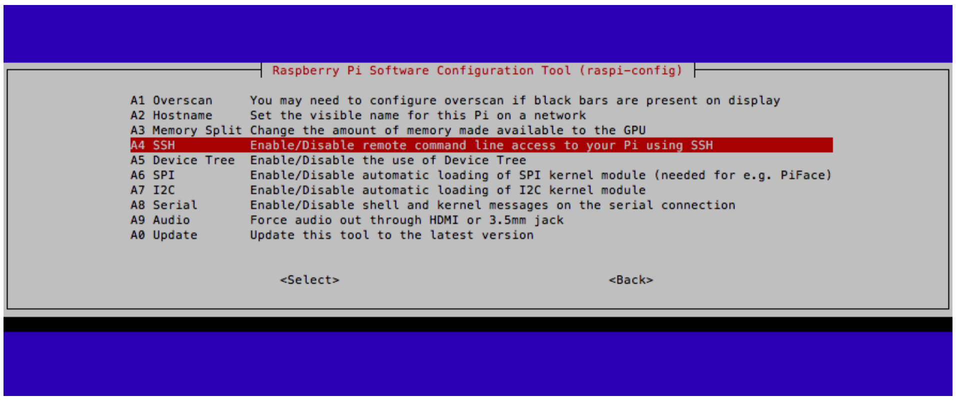
```
             Raspberry Pi Software Configuration Tool (raspi-config)

1 Expand Filesystem                Ensures that all of the SD card storage is available to the OS
2 Change User Password             Change password for the default user (pi)
3 Enable Boot to Desktop/Scratch   Choose whether to boot into a desktop environment, Scratch, or the command-line
4 Internationalisation Options     Set up language and regional settings to match your location
5 Enable Camera                    Enable this Pi to work with the Raspberry Pi Camera
6 Add to Rastrack                  Add this Pi to the online Raspberry Pi Map (Rastrack)
7 Overclock                        Configure overclocking for your Pi
8 Advanced Options                 Configure advanced settings
9 About raspi-config               Information about this configuration tool

                   <Select>                                   <Finish>
```

Figure 3.2.: Main window of the tool raspi-config

The available configuration options can be seen in Figure 3.2. By navigating to *Advanced Options*, which is number 8 on the list, some additional options can be displayed (Figure 3.3).



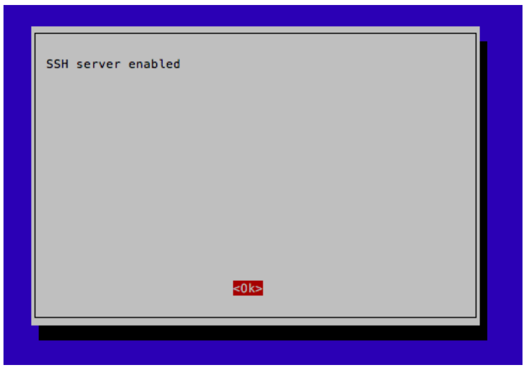
```
Raspberry Pi Software Configuration Tool (raspi-config)

A1 Overscan     You may need to configure overscan if black bars are present on display
A2 Hostname     Set the visible name for this Pi on a network
A3 Memory Split Change the amount of memory made available to the GPU
A4 SSH          Enable/Disable remote command line access to your Pi using SSH
A5 Device Tree  Enable/Disable the use of Device Tree
A6 SPI          Enable/Disable automatic loading of SPI kernel module (needed for e.g. PiFace)
A7 I2C          Enable/Disable automatic loading of I2C kernel module
A8 Serial       Enable/Disable shell and kernel messages on the serial connection
A9 Audio        Force audio out through HDMI or 3.5mm jack
A0 Update       Update this tool to the latest version

        <Select>                                    <Back>
```

Figure 3.3.: Advanced options window with the entry SSH

As the description suggests, number A4 from the list on the advanced options window is used to enable / disable the SSH server. By navigating to that number on the list and pressing Enter, the SSH server is enabled (Figure 3.4).



```
SSH server enabled

                         <Ok>
```

Figure 3.4.: SSH server enabled success message

For security purposes, the SSH connection is secured by a password. Login credentials are demonstrated in Listing 3.7.

```
username: pi
host: 141.22.80.211
port: 22
password: cQnbtps979@
```

Listing 3.7: SSH login credentials

Cron Commands

Assigning scripts as cron jobs to the operating system is done by executing the *crontab* command with the argument *-e* [26]. This command opens the system's default command-line based text editor so that the crontab can be edited (the default version of the crontab from Raspbian OS can be found in Appendix B). Once the crontab is opened for editing, the following two lines demonstrated in Listing 3.8 are added to the end of the crontab.

```
1 */10 * * * * /home/pi/pcp_si/scripts/establishWifiConnection
2 0 0 * * * /home/pi/pcp_si/scripts/downloadIcsFiles
3 @reboot /home/pi/pcp_si/scripts/keepRunning PcpSI
4 @reboot /home/pi/pcp_si/scripts/hideMouse
```

Listing 3.8: Cron jobs added to the crontab

There are 6 parameters (columns) to be set in a crontab entry: minute, hour, day, month, day of the week and the path to an executable. Each of these columns is separated by an empty space. A star means "any" in this context and the character "/" after a star is used for splitting the corresponding column into intervals. Line 1 in Listing 3.8 is used to register the Python script *establishWifiConnection* as a cron job that is executed every 10 minutes. As its name suggests, this Python script is used for establishing a connection to the wireless network HAW.1X, which is described in detail in Sections 3.3.1 and 4.1.2.

Additionally, line 2 of the Listing 3.8 is used to register the Python script responsible for downloading ICS files, *downloadIcsFiles*. The execution period is defined to be every day exactly at 00:00. The Python script *downloadIcsFiles* is described in detail in Sections 3.3.2 and 4.1.3.

As can be seen in line 3 and 4 two other Python scripts with the names *keepRunning* and *hideMouse* are registered as cron jobs. The script *keepRunning* expects a command or an executable as an argument and it is responsible of ensuring that the process is running by checking its status every 5 seconds. If it detects that process is terminated, it restarts the process. The other script, *hideMouse*, as its name suggests is responsible for hiding the mouse cursor when the system boots up. Both of the mentioned scripts are scheduled to run once at every boot. Detailed descriptions of the scripts can be found in Sections 3.3.3, 4.1.4 for *keepRunning* and Sections 3.3.4, 4.1.5 for *hideMouse*.

USB Ports Max Current

As previously mentioned in Section 2.1.2, by default Raspberry Pi model B revision 2 allows a total of 600mA of current to be drawn by the USB peripherals. In order to ensure that the WLAN dongle gets the necessary amount of current at all times, this limit is pushed to the maximum allowed. Since the power supply is able to deliver in total up to 2A, considering the current draw of the Raspberry Pi itself, directing a maximum of 1.2A to the USB peripherals is in the safe limits. Therefore, the following line demonstrated in Listing 3.9 is executed with root privileges. Once executed, the boot configuration file is appended by the line defined in quotation marks. This allows the USB peripheral current limit to be changed to 1.2A next time the device boots [10].

```
> $ echo "max_usb_current=1" >> /boot/config.txt
```

Listing 3.9: Command to append /boot/config.txt

Additional Fonts

Since the Swing and AWT GUI libraries use the fonts defined by the operating system, regardless of how nice the GUI design would be, its appearance would eventually be limited by the system fonts. In order to make the labels and texts on the GUI look nicer, the font *Arial* is added to the system. As a first step, all the *Arial* font files are copied to the directory */usr/local/share/fonts*. Afterwards, by executing the command *fc-cache* with the argument *-fv*, the system is forced to rebuild its font cache [4]. Once all these steps are completed, the font *Arial* becomes available to the operating system.

3.1.3. WLAN Dongle Setup

The wireless adapter is automatically recognized by the operating system once plugged in. First tests are carried out before updating the maximum allowed current on the USB peripherals and it is found that the WLAN dongle does not function in a stable manner with the USB peripheral current limitation of 600mA. Connections to wireless networks are possible; however it often disconnects and shows instable behavior. All these problems are discovered to be fixed once the maximum current is pushed up to 1200mA.

WPA Supplicant Configuration

As previously mentioned in Section 2.2.1, *wpa_supplicant* is an open source command-line based tool that allows to control wireless adapters and establish connections to wireless networks. In order to work with this tool, a configuration file needs to be created with a specific syntax. Based on the example file obtained from the official page of the tool, the following configuration file is set (Listing 3.10).

In the configuration file the desired wireless network for connection is defined by *network={}*, with all the network specifics going inside the curly brackets. As can be seen in line 2 of Listing 3.10, the SSID of the desired wireless network for acquiring Internet access is defined to be "HAW.1X". Line 4 with the keyword *key_mgmt* is used to define the authenticated key management protocol used by the network "HAW.1X". By setting the *key_mgmt* to "WPA-EAP", *wpa_supplicant* is configured to use *LEAP* authentication, which is required by the "HAW.1X" network [32]. Once the authentication method is set, the corresponding authentication credentials needs to be defined and lines 3 and 4 are used for this purpose. The keyword *identity* and *password* are used to identify the user name and password combination required to connect to the wireless network.

```
1 network={
2         ssid="HAW.1X"
3         key_mgmt=WPA-EAP
4         identity="abk284"
5         password="cQnbtps979@"
6 }
```

Listing 3.10: WPA Supplicant configuration file

After the creation of the configuration file, *wpa_supplicant* need to be tested. The tool requires two specific arguments: the identifier for the network adapter to be used and the path to a config file (Listing 3.11).

```
wpa_supplicant -i<network-adapter-identifier> -c<config-file-path>
```

Listing 3.11: WPA Supplicant arguments

The identifier of the WLAN dongle is retrieved from the operating system by the Linux command *ifconfig* [17]. This command is used to list all the available network adapters with their specifications as well as their identifiers.

```
> $ ifconfig
>   ...
   wlan0     Link encap:Ethernet  HWaddr ac:a2:13:7f:9e:2d
             UP BROADCAST MULTICAST  MTU:1500  Metric:1
             RX packets:0 errors:0 dropped:0 overruns:0 frame:0
             TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
             collisions:0 txqueuelen:1000
             RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Listing 3.12: WLAN dongle specifications obtained by the Linux command *ifconfig*

As demonstrated in Listing 3.12, the identifier for the wireless adapter is found to be *wlan0*. With this information at hand, *wpa_supplicant* is executed with root priviledges and the Raspberry Pi is successfully connected to the wireless network "HAW.1X" (Listing 3.13).

```
> $ sudo wpa_supplicant -iwlan0 -cconfig/wpa_supplicant.conf
>   Successfully initialized wpa_supplicant
   wlan0: SME: Trying to authenticate with 00:26:cb:d2:23:61 (SSID
      ='HAW.1X' freq=2412 MHz)
   wlan0: Trying to associate with 00:26:cb:d2:23:61 (SSID='HAW.1X
      ' freq=2412 MHz)
   wlan0: Associated with 00:26:cb:d2:23:61
   wlan0: CTRL-EVENT-EAP-STARTED EAP authentication started
   wlan0: CTRL-EVENT-EAP-METHOD EAP vendor 0 method 21 (TTLS)
      selected
   wlan0: CTRL-EVENT-EAP-PEER-CERT depth=3 subject='/C=DE/O=
      Deutsche Telekom AG/OU=T-TeleSec Trust Center/CN=Deutsche
      Telekom Root CA 2'
   wlan0: CTRL-EVENT-EAP-PEER-CERT depth=1 subject='/C=DE/O=
      Hochschule fuer Angewandte Wissenschaften Hamburg/CN=HAW
      Hamburg CA - G02'
   wlan0: CTRL-EVENT-EAP-PEER-CERT depth=0 subject='/C=DE/ST=
      Hamburg/L=Hamburg/O=Hochschule fuer Angewandte
      Wissenschaften Hamburg/OU=IT Service Center/CN=radius.haw-
      hamburg.de/emailAddress=dialup@haw-hamburg.de'
   wlan0: CTRL-EVENT-EAP-SUCCESS EAP authentication completed
      successfully
   wlan0: WPA: Key negotiation completed with 00:26:cb:d2:23:61 [
      PTK=CCMP GTK=TKIP]
   wlan0: CTRL-EVENT-CONNECTED - Connection to 00:26:cb:d2:23:61
      completed [id=0 id_str=]
```

Listing 3.13: Execution of *wpa_supplicant* and its stdout

3.2. Project Directory

For future reference it is important to demonstrate the project directory hierarchy with the respective subdirectory names. In order to keep configuration files, Java classes and Python scripts in an organized fashion, the following directory hierarchy is created under */home-/pi/pcp_si/* with the corresponding subdirectories:

- classes - Java source code is stored under this directory
- config - All the configuration files are located in this directory including the wpa_supplicant configuration file
- fonts - A copy of the installed fonts are placed under this directory
- ics_files - This is a directory for the downloaded ICS files
- images - Images for the graphical user interface are kept under this directory
- logs - Stdout of the wpa_supplicant as well as a table in text format which contains all the displayed information on the Java GUI are located in this directory
- scripts - This is a directory for the Python scripts

3.3. Python Scripts

As for any operating system, there are only certain things that can be handled by the Raspbian OS. The tasks that are out of the operating system's scope need to be implemented manually. These tasks are mainly to do with the automation of certain processes, which can be considered as the backend of the system.

3.3.1. Establish WiFi Connection

Establishing WiFi connection is one of the aforementioned back-end tasks that need to be automated. For this purpose a script is designed. As can be seen in the activity diagram demonstrated in Figure 3.5, the script first needs to check Internet connectivity. This is achieved by pinging *http://www.google.com*. The reason for selecting *google.com* is the very low probability of this domain name changing in the near future. As demonstrated, if the script discovers that the system has Internet access, it should simply terminate. Otherwise it should restart the wireless adapter and try to connect to the wireless network "HAW.1X" with the help of *wpa_supplicant*. If the connection to the network fails, it should keep trying by restarting the wireless adapter. Once the network connection is established, Internet access should be tested by pinging *google.com*. This ping step should be repeated for a maximum of 50 times separated by 0.2 seconds of sleep until it succeeds; if all 50 attempts are unsuccessful (approximately after 10 seconds), then the whole process should be repeated from the restart of the wireless adapter. If at any time pinging *google.com* succeeds, then the script should terminate.

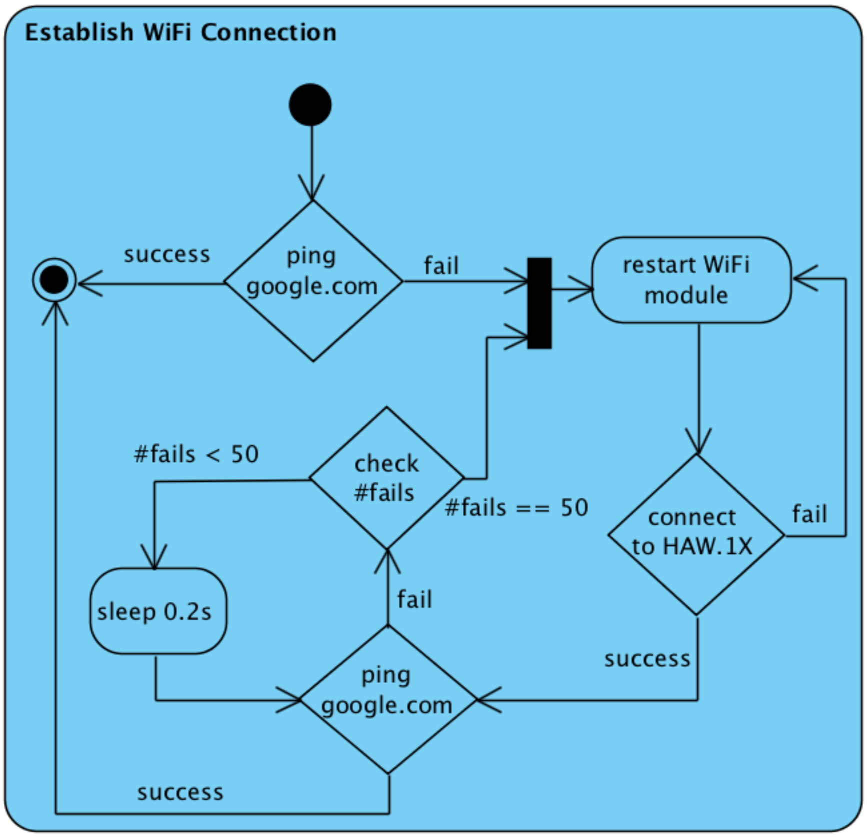


Figure 3.5.: Activity diagram of the Python script establishWifiConnection

3.3.2. Download ICS Files

Downloading ICS files is another back-end task that requires automation and this is achieved with the help of a script (Figure 3.6). The python script *downloadIcsFiles* should first execute the previously demonstrated *establishWifiConnection* (Section 3.3.1), through which it can ensure that the system has Internet access. Afterwards, links for the ICS files should be retrieved from a configuration file. As previously mentioned, all the PC Pool rooms of interest have their own ICS files located at an online location. In case the location of these files or their names change in the near future, these links should not be hard coded and should be realized in the form of a configuration file, which is parsed by this script. Once the links are determined, the script should download the files one by one and store them in the directory called *ics_files*.

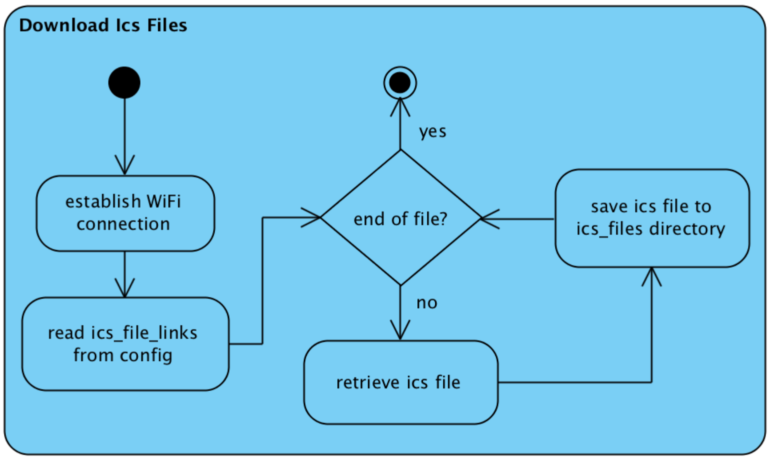

Download Ics Files
establish WiFi connection
read ics_file_links from config
end of file?
yes
no
retrieve ics file
save ics file to ics_files directory

Figure 3.6.: Activity diagram of the Python script downloadIcsFiles

3.3.3. Keep a Process Running

As previously mentioned, a wrapper is required to periodically check whether the graphical user interface is running and restart it if not. This wrapper is designed in the form of a Python script that does the mentioned check every 5 seconds and its corresponding activity diagram can be found in Figure 3.7.

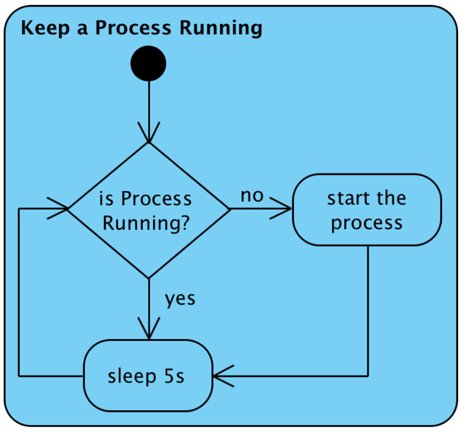


Figure 3.7.: Activity diagram of the Python script keepRunning

3.3.4. Hide Mouse

As previously mentioned, the idea is to realize the Java GUI application in full screen mode. The challenge here is to make the mouse cursor displayed on the GUI invisible. Initially, the idea was to use Java to set the cursor of the GUI application invisible, however this was unsuccessful. As an alternative, the window system of Raspbian OS (X by XOrg Foundation) is discovered to be offering this functionality through the optional argument *-nocursor* [33]. Unfortunately, this option is found to be non-existent for versions older than 1.7. Once the version being used by the Raspbian OS is checked, it is found out that the installed and used version is 1.6. The workaround offered for the earlier versions , however, only makes the root window cursor invisible, meaning the Java GUI application would still have its own cursor visible. For the aforementioned reasons, it is decided to implement another workaround by using a tool called *xdotool* [12]. This tool offers an option to move the cursor to the given (x, y) coordinate on the screen, hence setting the coordinates somewhere outside of the screen would make the cursor invisible. Therefore, considering that the screen size is going to have a full HD resolution of 1920x1080, moving the cursor to the position (2000, 2000) would ensure that the cursor is not visible, hence in theory the execution of the command *xdotool mousemove 2000 2000* should be sufficient. However, since the idea is to execute this command at the start up of the system in the form of a cron job, the display has to be specified for the shell where the cron command is executed. Otherwise, since there is no display known to the *xdotool*, it terminates with an error message not being able to locate the cursor. Therefore, the final command to be executed also specifies the display which can be seen in Listing 3.14. Considering that there is going to be only one screen attached to the Raspberry Pi, the index number for the display is 0. In a unix system, displays detected by the operating system are indexed starting from 0 and accessed via *DISPLAY=:<index>* [17].

```
DISPLAY=:0 xdotool mousemove 2000 2000
```

Listing 3.14: The xdotool command to be executed to move the cursor outside of the screen

3.4. Java GUI

So far the back-end offers the ability to maintain background tasks, which involves establishing an Internet connection and downloading the ICS files that are essential for the Java GUI. As previously mentioned, room schedules need to be parsed from the corresponding ICS files and displayed in a user friendly manner. Since the ICS files are only downloaded at midnight, precisely at 00:00, accessing the files once a day right after midnight should be sufficient. In addition, since the displayed schedule is only dependent on date and time, it can only change with the smallest possible time unit of one minute. Therefore, in order to save power and avoid unnecessary computations, the GUI should be active for updating the displayed information once every 60 seconds. In order to achieve the aforementioned functionality one timer with a period of 60 seconds is sufficient. With the help of this timer, the GUI can be updated and access to the ICS files can be triggered right after midnight. Additionally, the ICS files should be accessed only after the download process is completed. Therefore, the files should be read a couple of minutes after midnight just to ensure that the download process is over and new files are available.

3.4.1. Layout Design

In order to provide not only the availability information of the PC Pool rooms, but also information regarding their locations including orientation, it is decided to display a floor plan with the focus on the PC Pool rooms of interest as a base for the GUI (Figure 3.8).

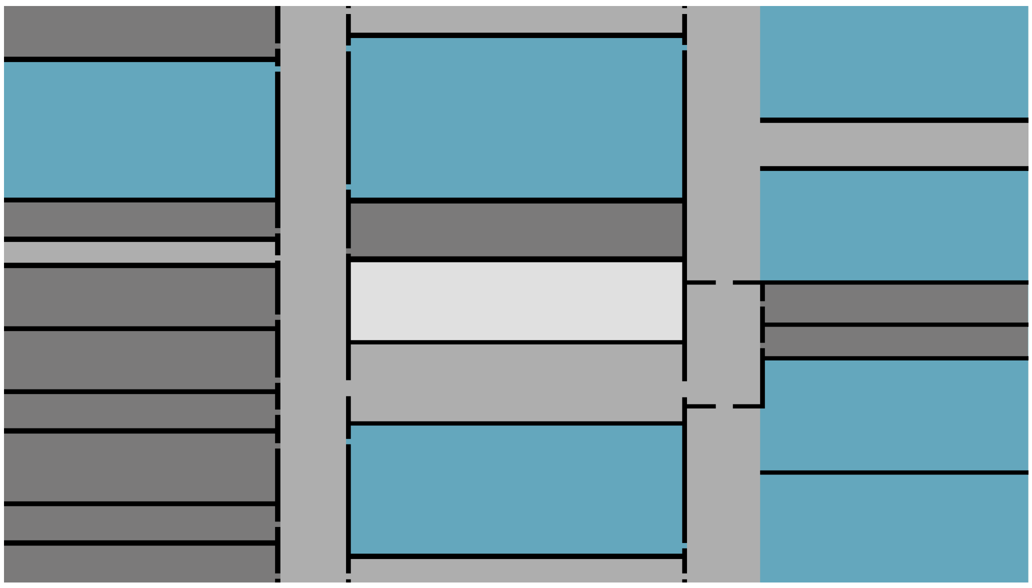


Figure 3.8.: Floor plan with rooms of interest in blue - GUI background

Room Labels

In order to have room for future changes, the background image is kept free of text and the room labels are decided to be set by the GUI. For this purpose a configuration file is placed in the *config* directory with the name *labels*. The configuration file contains a line for each room in the PC Pool and the labels are defined as demonstrated in Listing 3.15.

```
<current_room_name>=<label_to_be_displayed_on_the_GUI>
```

Listing 3.15: Label definition for the configuration file *labels*

It is worth mentioning that the GUI can only know where to place these labels if the *<current_room_name>* is set correctly. In other words, the left hand side of the equal sign defines the location of a given label and therefore it should not be changed. For example, if in the future the room 1381 starts to be called 1390 and the label needs to be updated, the entry in the file should look as demonstrated in Listing 3.16. The default version of the *labels* configuration file with its effect on the GUI background can be found in Appendix C.

```
1380=1390\nThis is an example of changing the label for the room
    1380 and making a new line within the label itself
```

Listing 3.16: Example for changing a room label

In order to avoid restarting the GUI when the configuration file *labels* is edited, the GUI should access this file once a day at midnight as well and update its labels accordingly.

Labels Regarding Availability

For displaying availability information of a room, rectangular shaped labels are used. These labels contain event summary including start-end times and the remaining time if applicable. The idea is to place a label for each event, sorting them by their start times within a box that symbolizes the room (locations of the PC Pool rooms on the floor plan can be seen in Appendix C).

The label of the ongoing event is set to be larger than the others in order to draw attention and focus on the ongoing event. Ongoing events should also display the remaining time together with an image of a stopwatch. Green is used to indicate that the room is available and red is used for occupied. For the upcoming events a third color option is present; blue is used for an event that is going to take place between the shown start-end times. If there are no events present for a given room for that day, then a large green label is displayed stating that the room is available (Figure 3.10). In order to make it easier to modify the labels according to the changes in events, it is decided to assign stationary labels to the rooms. According to changes in events during a given day, the visibility setting of the labels is altered, making it easier to apply changes without moving the labels around. For this purpose each room is assigned a large *available* label as well as a different sized label for the top event. In addition, depending on the space that the room occupies on the GUI, smaller labels for upcoming events are created and up to 3 labels are assigned to a given room. As an example, room 1381 is demonstrated in Figure 3.9. Even though there are a minimum of 4 events scheduled for the room, only 3 are visible. This is due to the maximum number of labels fitting in the room space being 3. Once the top event's remaining time runs out, labels move upwards and the 4th event becomes visible.

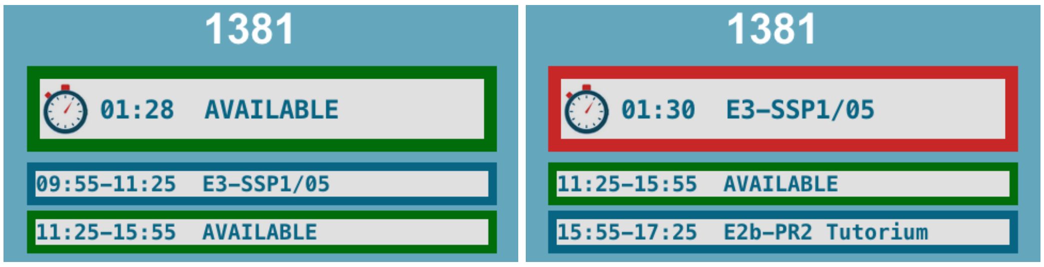


Figure 3.9.: PC pool room 1381 with its availability information

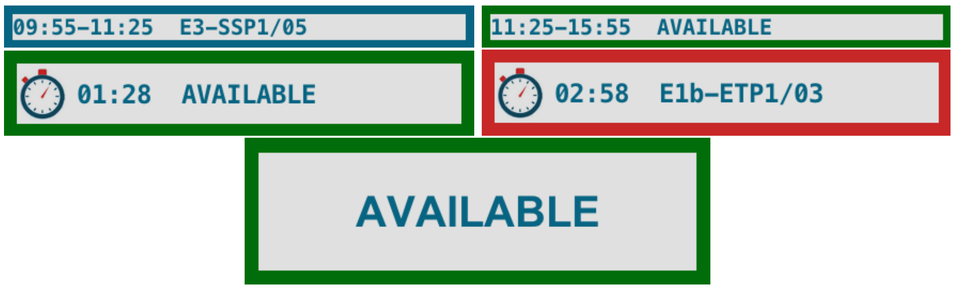


Figure 3.10.: Variety of different labels showing availability

3.4.2. GUI Update Timer Task

As previously mentioned, updating labels on the GUI is to be done by a timer, so that the update would take place precisely every 60 seconds. In addition, as also mentioned earlier, before the regular update process starts, it is also required to detect if it is after midnight. In order to give enough buffer for the download process to be completed, the time for parsing the new ICS files and also refreshing stationary room labels is defined to be 00:02. The given buffer of 2 minutes is more than enough for the ICS files to be downloaded, which is completed within seconds (Table 4.2).

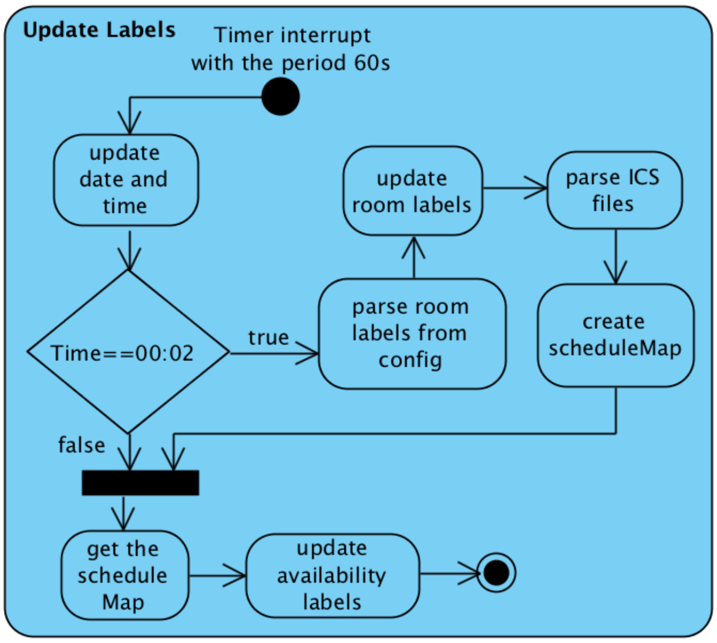


Figure 3.11.: Activity diagram of the GUI update timer task

As demonstrated in Figure 3.11, first the time and date displayed on the GUI are updated. Afterwards, the current time is compared to 00:02 and if it is a match, then first the *labels* configuration file is parsed, which is followed by the parsing of the new ICS files including the creation of a schedule map. A schedule map in this context refers to a map containing *room name : event list* pairs. Once the mentioned tasks are completed, the GUI updates the availability labels of each room according to the schedule map.

3.4.3. OOP View

Considering the activity diagram demonstrated in Figure 3.11, the process including the timer can be distributed among the following classes:

- UpdateGuiTimerTask - Timer class which extends the Java class TimerTask [24]
- PcpSI_Gui - Contains the frame, labels, images etc. In other words anything related to the display is part of this class
- JComponentWithBackground - A class that extends Java class JComponent [24] to add background image support
- JImage - Extends Java class JLabel [24] to add background image support

- StatusLabelHandler - Responsible for updating the availability labels of rooms
- DateHandler - A class for handling all the necessary tasks in regard to date and time
- FileHandler - Read / write operations from / to files are done by this class
- ImageHandler - Similar to FileHandler, reads images from files
- RoomInfoCollector - A class that creates the schedule map from the ICS files
- Controller - Handles all the classes except PcpSI_Gui, JComponentWithBackground, JImage and StatusLabelHandler

An overview of the aforementioned classes is demonstrated in Figure 3.12 with a simplified class diagram. A detailed explanation of why this specific OOP design is used can be found in Section 4.2.

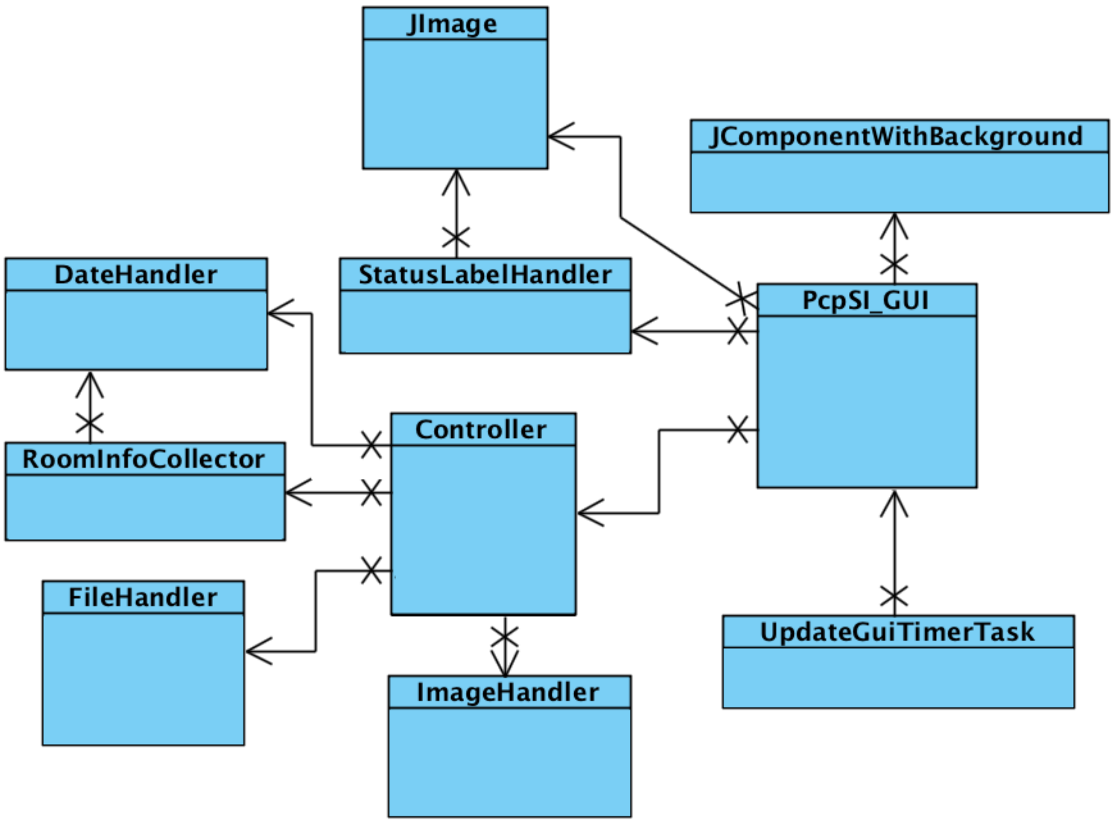


Figure 3.12.: A simplified class diagram as an overview

4. Realization

Following up on the design, Python scripts and the Java GUI are implemented. This chapter focuses on the actual implementation and the reasoning for the taken steps as well as the test cases.

4.1. Python Scripts

In order to avoid having redundant code, all the methods required by the Python scripts are put into a separate file called *functions.py*, which is located together with the scripts under the *scripts* directory. Each Python script is tested and the evaluation of these tests can be found in the corresponding sections.

4.1.1. Common Methods

There are in total 7 methods implemented and put into the *functions.py*, which is located in *scripts* directory.

runCommand(cmd, blocking=False)

The method *runCommand* is implemented to execute a given command and buffer its stdout and stderr so that they can be accessed by a Python module. Executed commands can be as basic as a Linux command or they can be scripts or even programs written in another language such as Java. In order to achieve the mentioned functionality, there are two Python libraries providing useful methods: *os.system()* and *subprocess.Popen()* [20]. Both of these methods allow executing commands that can be executed within a Linux shell. However, the method *Popen* from the library *subprocess* is a better fit for the task. Unlike the *system* method, it allows access to stdout and stderr, which are just echoed by the *system* method to the shell where the parent Python process is started. Therefore, a decision is made to use the library *subprocess* allowing the stdout and stderr of an executed command to be retrieved.

The implementation uses the method *Popen* with 3 arguments:

- An array of strings containing the command to be executed as its first element and arguments to this command as the other elements.
- Either a file to direct the stdout, or *PIPE* object from the *subprocess* library to buffer it.
- Similar to stdout, either a file to direct the stderr, or again *PIPE* object from the *subprocess* library for buffering.

Once executed with the corresponding arguments, *Popen* returns an object referring to a subprocess started by the parent Python process itself. The method *runCommand*, in its simplest form, expects an array of strings defining a command with its arguments so that it can be executed by *Popen*. When provided, *Popen* is executed with the given array storing the stdout and stderr inside its buffers by using *PIPE*s from the subprocess library. Once buffered, stdout and stderr can be accessed through the object returned by *Popen*, which is a reference to the subprocess started. The method *runCommand* returns this object without any alteration. As an addition to this functionality, a flag is defined, which is an optional argument to *runCommand* and set to *False* by default. If this argument is set to true, the object referring to the subprocess is not returned until the subprocess is fully executed, which is achieved with the help of the reference object's method *wait*. This way the program can wait on that function call until the subprocess terminates. This functionality is useful when a Linux command is used. In most Linux commands, stdout and stderr only become available once the command has run its course. This is why if the flag is set, the object referring to the executed Linux command is not returned until the stdout and stderr become available. The corresponding functionality is demonstrated by an activity diagram in Figure 4.1.

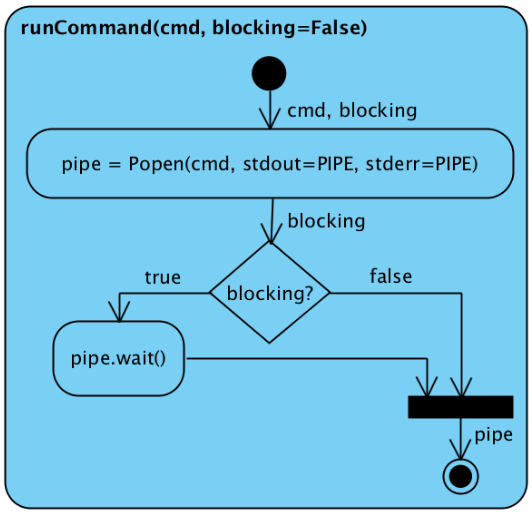

runCommand(cmd, blocking=False)
cmd, blocking
pipe = Popen(cmd, stdout=PIPE, stderr=PIPE)
blocking
true
blocking?
false
pipe.wait()
pipe

Figure 4.1.: Activity diagram - runCommand(cmd, blocking=False)

getPath()

In order to make the Python scripts executable from any directory, a method is required to inform the Python script about its own path so that the script can point to the right relative path inside the project. This scenario might sound straightforward, which is the case when the script is executed from a shell with its *pwd* set to the script's directory. No problems occur with the relative paths in this case, since accessing a file called *temp* located in a directory above with the path *../temp* simply works. However, things change if the shell has a different *pwd* [17] and executes the script from a different directory by just pointing to the script's location. For example, assuming that the script is located in the directory *sub* within *parent* inside *grandparent*. For the sake of simplicity, this is how the directory looks like: *grandparent/parent/sub/script.py*. Now, considering that the required file *temp* is located inside *parent* and is accessed by the script with the relative path *../temp*. Let's assume that the script is executed from the *grandparent* by *parent/sub/script.py*. Then the relative path *../temp* would point to *grandparent/../temp*, which is the directory above, hence not *parent*; resulting in the script being not able to locate the file *temp*. In order to avoid such problems and make the scripts fully executable by cron command scheduling, the *getPath* method is implemented. This method returns the absolute path to the script where it is executed. For this purpose, the Python library *os* and its methods *path.dirname* and *path.realpath* are used [20]. The method *os.path.realpath()* returns the absolute path to the *File* object passed as an argument. Python attribute *__file__* points to the path of the script where it is accessed. For example, if a script with the name *test.py* is located in a directory with the following path: */home/user/*, executing *print __file__* inside the script would print */home/user/test.py* to the console. In other words, executing *os.path.realpath(__file__)* returns the absolute path of the script. However, the path to the directory where the script is located is the path that is required. Therefore, the other *os* method, *os.path.dirname()* is used. Given path to a file as an argument, this method returns the path to the directory where the file is located. The method *getPath* uses both of these methods and returns the absolute path to the *functions.py*'s directory. The corresponding functionality is demonstrated by an activity diagram in Figure 4.2.

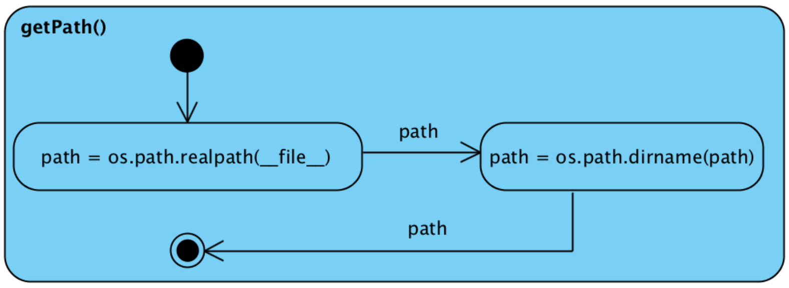

getPath()
path = os.path.realpath(__file__)
path
path = os.path.dirname(path)
path

Figure 4.2.: Activity diagram - getPath()

ping()

The method *ping()* is implemented to test Internet connectivity. By using the *runCommand* method, the linux command *ping* [17] is executed with root privileges (Listing 4.1).

```
sudo ping -c 4 google.com -W 1
```

Listing 4.1: Ping command with root privileges

There are two things worth mentioning from the *ping* command demonstrated. The argument *-c* is used to define the number of packets to be transferred during the ping process; which is set to 4. The reason why not 1 but 4 packets are transferred is the intention to test not only the connectivity but also the stability of the Internet connection. For the same purpose, the timeout is set to be 1 second by the *-W* option. By these settings, it is ensured that the connectivity is stable enough for downloading the ICS files.

The method *runCommand* is called with the optional argument *blocking* set to *True*, so that the program waits until the *ping* command runs its course and terminates. Afterwards with the help of the reference to the subprocess, stderr is checked for errors and the method returns a boolean indicating the success or failure of the execution of *ping*. The corresponding functionality can be seen in the activity diagram demonstrated in Figure 4.3.

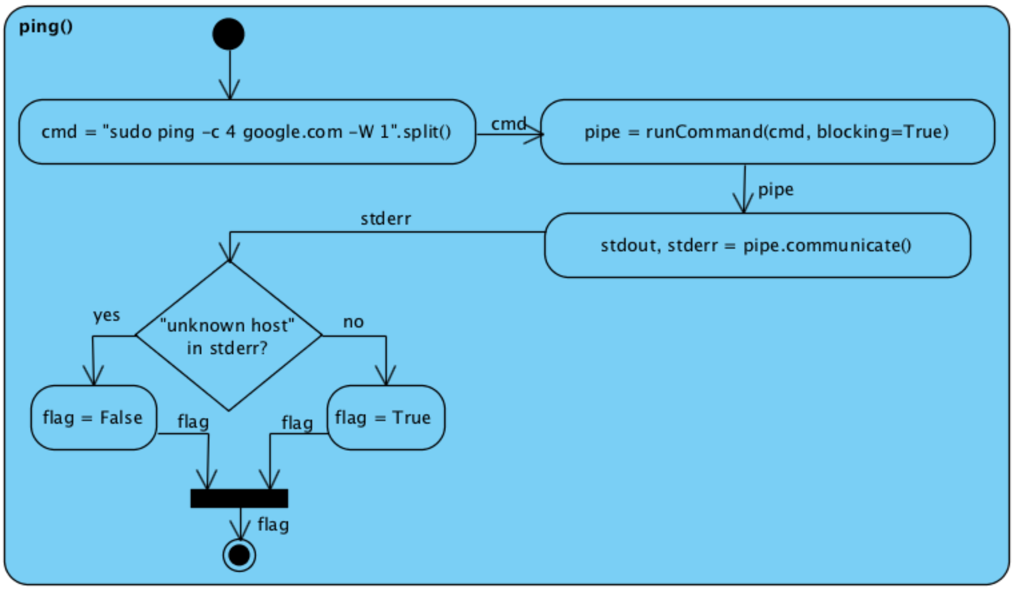

ping()
cmd = "sudo ping -c 4 google.com -W 1".split()
cmd
pipe = runCommand(cmd, blocking=True)
pipe
stdout, stderr = pipe.communicate()
stderr
"unknown host" in stderr?
yes
no
flag = False
flag = True
flag
flag
flag

Figure 4.3.: Activity diagram - ping()

An alternative to using *ping* would have been directly trying to download the ICS files. The reason why *ping* is called beforehand is the way it allows checking the stability of the connection. If the connection is not stable, procedure might get interrupted while a file is being downloaded. No problems would occur if this is detected; the download process can simply be restarted. However if missed, one of the ICS files would end up being corrupt. With the help of the *ping* call, such errors are less likely to occur.

getPIDs(pName)

The method *getPIDs* is implemented to retrieve process ids for a given process name (pName). For this purpose, again the method *runCommand* is used. The linux command *ps* is executed with the argument *-C* [17]. This is used to retrieve information regarding running processes from the operating system. The argument *-C* followed by a process name is used to filter out processes by the given name. After executing the *runCommand*, the returned reference is used to parse the stdout for the process ids. First, the stdout is split into lines, then the leading and trailing empty spaces are stripped from each line. This is followed by the line getting split by the remaining empty spaces, resulting into an array of strings containing the process id in its first element. Afterwards, all the process ids are packed into an array and the method returns the aforementioned array before it terminates. The activity diagram of the method *getPIDs(pName)* can be found in Figure 4.4.

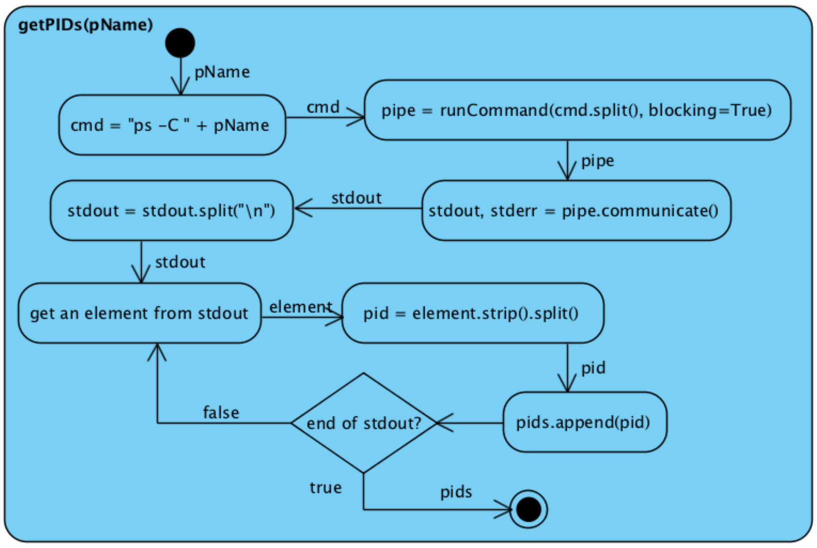

getPIDs(pName)
pName
cmd = "ps -C " + pName
cmd
pipe = runCommand(cmd.split(), blocking=True)
pipe
stdout, stderr = pipe.communicate()
stdout
stdout = stdout.split("\n")
stdout
get an element from stdout
element
pid = element.strip().split()
pid
pids.append(pid)
end of stdout?
false
true
pids

Figure 4.4.: Activity diagram - getPIDs(pName)

killProcess(pName)

The method *killProcess* kills all the processes by the given name with the help of the *runCommand* and the *getPIDs* methods. First, the process ids for the given *pName* are retrieved by the *getPIDs(pName)* call. Afterwards, by iterating through the list, for each process id the linux command *kill* is executed by calling the *runCommand* [17]. Blocking is set to ensure that the program waits until the termination of one process before calling the *kill* command for the next one. The corresponding functionality is demonstrated with the help of an activity diagram (Figure 4.5).

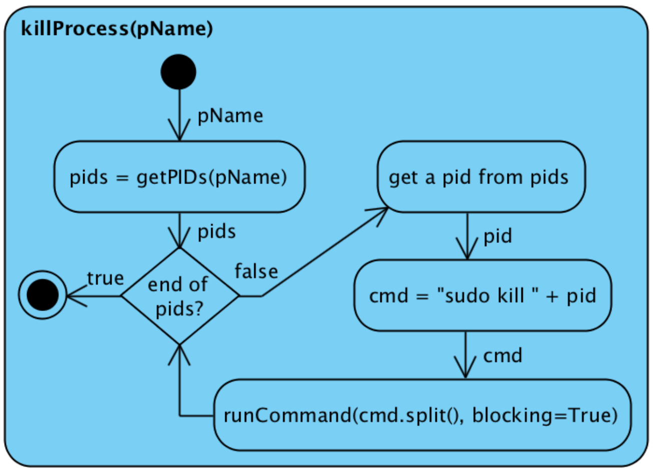


Figure 4.5.: Activity diagram - killProcess(pName)

restartWifiModule()

In order to restart the WiFi module, first all running wpa_supplicant processes need to be killed. For this exact reason, the method *killProcess* is implemented. When called, the first thing that is done by the method *restartWifiModule* is to call the previously mentioned method *killProcess* with the argument *"wpa_supplicant"*. Once it is ensured that all the running sessions of *wpa_supplicant* are killed, the linux command *ifconfig* is used to disable and enable the wireless adapter [17]. This command stands for "interface configuration" and it is used to display and configure network interfaces. Given a network interface as an argument, by passing keywords *up* or *down* as an argument, the given network interface is enabled or disabled respectively. Therefore, it is executed with root privileges using the arguments *wlan0* and *down* first. The method *runCommand* is called with the corresponding command for this purpose. Afterwards, the same command with *up* instead of *down* is executed, which ensures that the previously disabled *wlan0* interface is enabled again. For both cases the *runCommand* is called with the *blocking* set to *True*, which ensures that the enable command is not called before the adapter is disabled. The corresponding activity diagram is demonstrated in Figure 4.6.

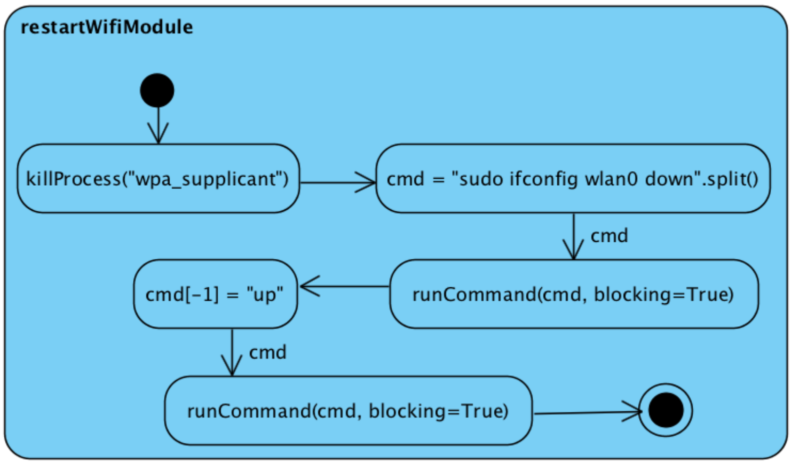

restartWifiModule
killProcess("wpa_supplicant")
cmd = "sudo ifconfig wlan0 down".split()
cmd
runCommand(cmd, blocking=True)
cmd[-1] = "up"
cmd
runCommand(cmd, blocking=True)

Figure 4.6.: Activity diagram - restartWifiModule()

connectToWPAwifi()

The final common method is called *conenctToWPAwifi*. As its name suggests, purpose of this method is to connect to the wireless network specified in the *wpa_supplicant.conf* file located in the *config* directory. By using the *runCommand* method, *wpa_supplicant* is executed with the arguments shown in the previously demonstrated Listing 3.11. Additional to the given arguments, for debug purposes *-f* with a path to a log file is added and the log file is stored under the *logs* directory. The reason for directing the stdout to a file instead of the subprocess itself is that a buffer with a constant size is used by the subprocess otherwise. Since wpa_supplicant runs in the background as long as the wireless connection is in the established state, the stdout can grow up to a size that violates the buffer size, which can cause a crash. To avoid problems of this sort, usage of a log file is preferred and the stdout of the wpa_supplicant is directed there.

In order to define the log file and additionally to get the directory of the network configuration file (*wpa_supplicant.conf*), the method *getPath* is called first. Once the log and the configuration file paths are defined, the existence of a previous *wpa_supplicant* log file is checked and it is cleared by using the method *runCommand* and calling the linux command *rm* [17]. After this step, wpa_supplicant is executed again with the help of the *runCommand* method. Since the program wpa_supplicant runs as long as the wireless connection is active without terminating, the blocking flag of the *runCommand* is set to *False*. Afterwards, the program sleeps for 0.1 seconds and starts checking the log file line by line continiuously looking for an error or a success message. A boolean is returned as an indicator (Figure 4.7).

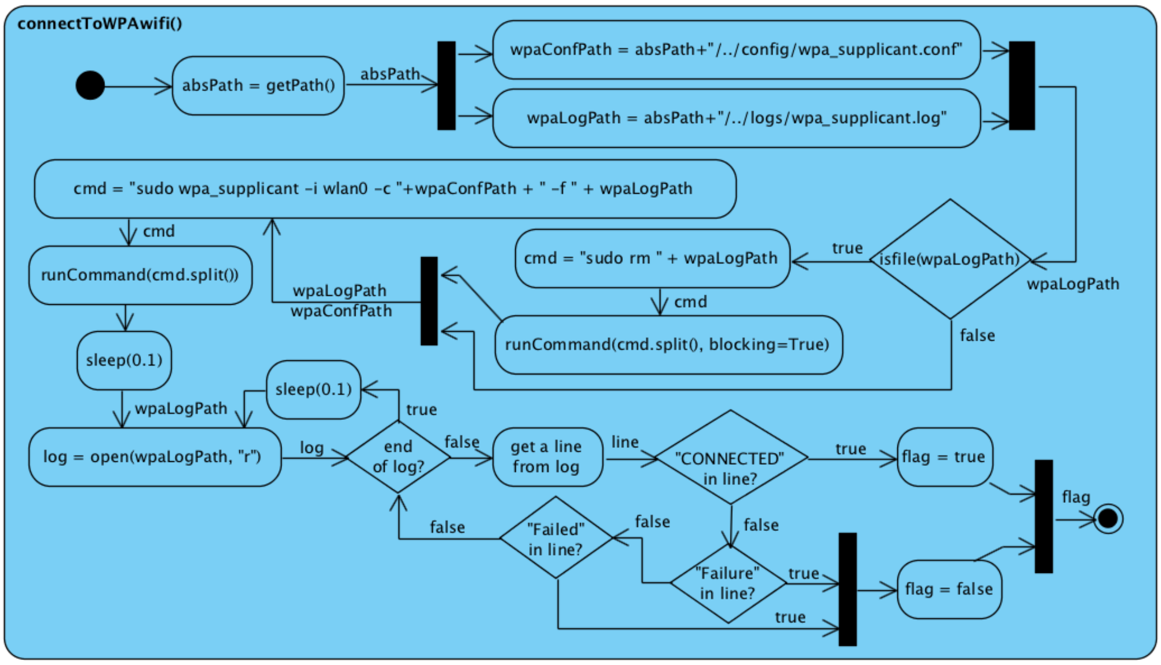


Figure 4.7.: Activity diagram - connectToWPAwifi()

4.1.2. establishWifiConnection

As previously explained in Section 3.3.1, the purpose of the script *establishWifiConnection* is to connect to the "HAW.1X" wireless network. In order to implement the necessary functionality, methods *ping()*, *restartWifiModule()* and *connectToWPAwifi()* are used (Section 4.1.1). The method *ping* is used to test the Internet connectivity and the script exits if *ping* is successful. As previously mentioned, if *ping* fails the wireless adapter is restarted. For this purpose, as its name suggests, the method *restartWifiModule* is executed. This method call is followed by establishing a connection to the wireless network with the help of the method *connectToWPAWifi*. This process is repeated until the connection is established and the method *ping* is successful. Since in Python everything, including functions themselves, is an object, the corresponding implementation is demonstrated with a sequence diagram (Figure 4.8).

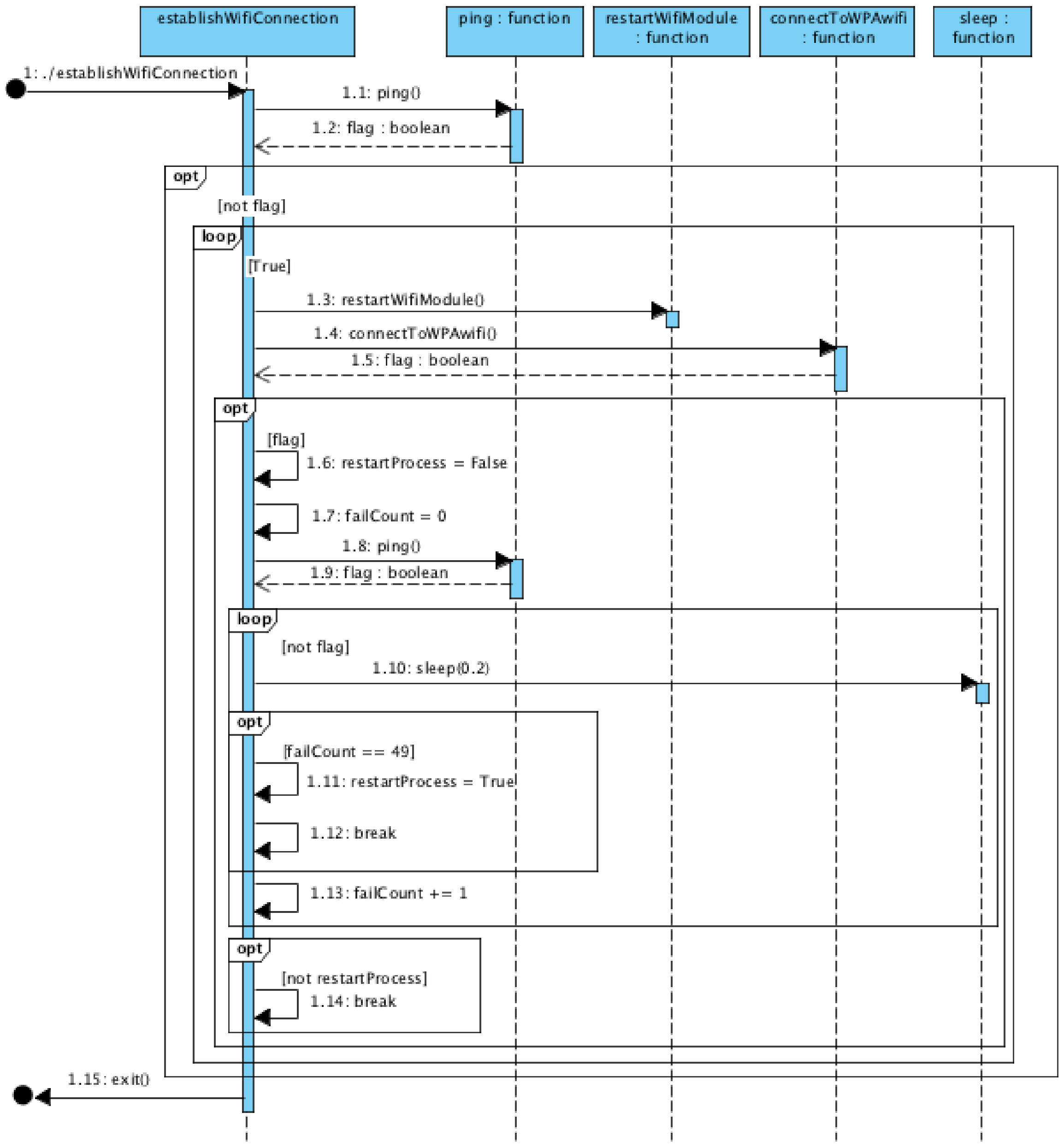


Figure 4.8.: Sequence diagram - establishWifiConnection

Test Cases

There are 3 different scenarios that need to be tested:

A. Internet connection is present and the script terminates

B. Internet connection is not present and the script establishes connection to "HAW.1X"

C. Internet connection is not present and the script establishes connection to "HAW.1X", however it fails to acquire Internet access

Scenario A is tested first by executing the script *establishWifiConnection* with Internet connection being present. After 10 consecutive time measurements, the mean value for this scenario is found out to be 4.08 seconds. As expected, the script has run its course by just pinging google.com and terminating once the Internet connection is detected.

As the next step, scenario B is tested. For this purpose the system is forced to disconnect from the wireless network "HAW.1X" and the script is called afterwards. Similarly to scenario A, the total run time is measured. As expected, the script first detected that the system has no internet access and restarted the wireless adapter, followed by the connection to the wireless network "HAW.1X". After 10 consecutive time measurements, the mean value of this scenario is found out to be 12.37 seconds.

As the last step, scenario C is tested. In order to simulate the scenario, first the system is connected to a wireless network without Internet access. Once the connection is established, *wpa_supplicant* configuration file is reverted back to its original so that the system connects to "HAW.1X" in its second attempt. The reason for testing this scenario is the possibility of the system connecting to the network "HAW.1X" without being able to acquire Internet access. Usually such issues are fixed once the wireless adapter is restarted. Therefore, the system is allowed access to the correct network in its second attempt. After 10 consecutive time measurements, the mean value is found out to be 24.41 seconds.

Once all the tests are completed, it is found out that the execution of the script *establishWifiConnection* takes 4.08 and 24.41 seconds best and worst cases respectively (Table 4.1).

	Scenario A	Scenario B	Scenario C
Execution Time [s]	4.08	12.37	24.4

Table 4.1.: Execution time of scenarios A, B and C for the script establishWifiConnection

4.1.3. downloadIcsFiles

The purpose of the script *downloadIcsFiles*, as explained in Section 3.3.2, is to download the ICS files from the links provided in the configuration file *ics_file_links*. In order to achieve the given task, the script uses the following methods from the given Python libraries [20]:

- re.sub(): Python library *re* allows operations on strings by regular expressions. The method *sub()* expects 3 arguments: pattern to be replaced, replacement string and the target string.
- urllib.urlretrieve(): The library *urllib* has useful methods for accessing the Internet from Python. The method *urlretrieve()* allows to download a file from a given link (first argument) and save it to the given path (second argument).

Additional to the libraries mentioned, the script uses the methods *getPath* and *runCommand*. As explained in Section 4.1.1, absolute paths are required while accessing configuration files so that the scripts can be executed as cron jobs. The *getPath* method is used for this purpose, returning the absolute path to the directory where the Python scripts are located. The method *runCommand* on the other hand, is used to call the other Python script *establishWifiConnection* to ensure the system has internet access before it starts to download the ICS files.

Once executed, the script retrieves the absolute path to its own directory and defines the relative paths to the file *ics_file_links* and to the directory *ics_files* where the downloaded ICS files are stored. Afterwards, by calling the script *establishWifiConnection* it ensures Internet connectivity. Once all the aforementioned steps are successfully executed, it downloads each ICS file by iterating through the links that are read from the file *ics_file_links*. Links are stored with "file name = link" pairs. Therefore, each read line is split into the file name and the link before the method *urlretrieve* is called. The corresponding implementation is demonstrated with a sequence diagram in Figure 4.9.

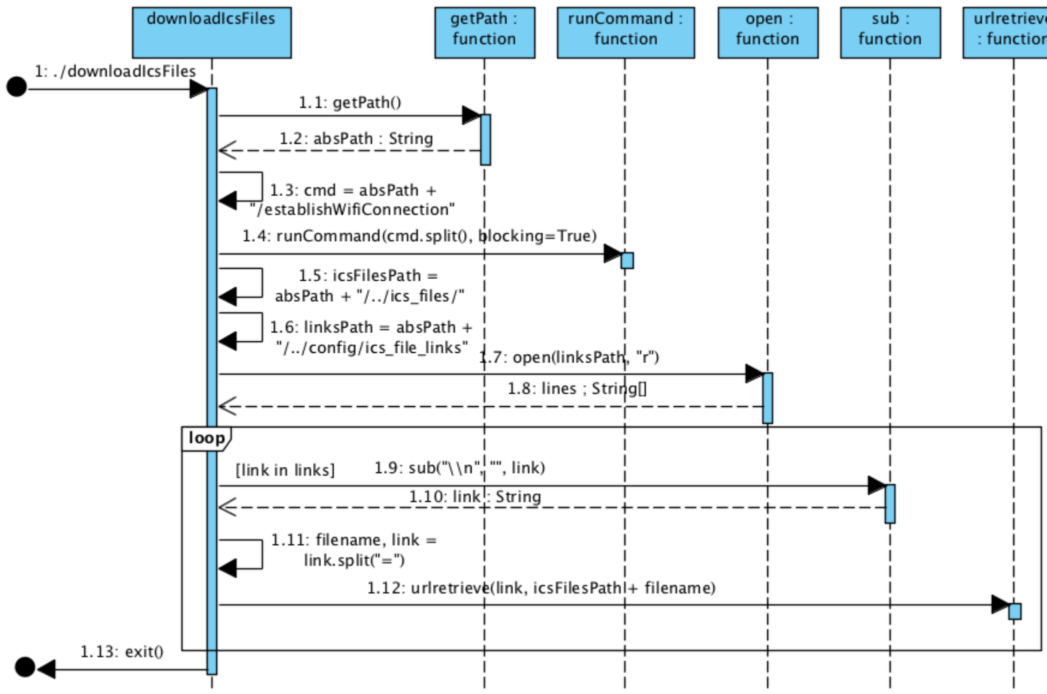

downloadIcsFiles
getPath : function
runCommand : function
open : function
sub : function
urlretrieve : function
1: ./downloadIcsFiles
1.1: getPath()
1.2: absPath : String
1.3: cmd = absPath + "/establishWifiConnection"
1.4: runCommand(cmd.split(), blocking=True)
1.5: icsFilesPath = absPath + "/../ics_files/"
1.6: linksPath = absPath + "/../config/ics_file_links"
1.7: open(linksPath, "r")
1.8: lines ; String[]
loop
[link in links]
1.9: sub("\\n", "", link)
1.10: link : String
1.11: filename, link = link.split("=")
1.12: urlretrieve(link, icsFilesPath + filename)
1.13: exit()

Figure 4.9.: Sequence diagram - downloadIcsFiles

Test Cases

There are 2 scenarios to be tested on the script *downloadIcsFiles*:

A. Internet connection is present and the script is called

B. Script is called when the system has no Internet access

The reason for testing both cases is to determine exactly how long it takes to download the ICS files in the best and worst case scenarios. In the best case scenario (scenario A), the script only pings google.com and starts downloading the ICS files. After 10 consecutive time measurements, this scenario is found out to take 4.82 seconds in average with the download process only taking 0.61 seconds.

The worst case scenario (scenario B) is dependent on Internet access. As it is found out from the test cases run on the script *establishWifiConnection* the best and worst cases for this script to execute are 4.08 and 24.41 seconds respectively (Table 4.1). Bearing this in mind, it can be assumed that the script *downloadIcsFiles* takes 4.7 to 25.1 seconds as best and worst cases respectively (Table 4.2).

	Scenario A	Scenario B
Execution Time [s]	4.7	25.1

Table 4.2.: Execution time of scenarios A and B for the script downloadIcsFiles

As previously mentioned in Section 3.4.2, from the Table 4.2 it can be concluded that the given buffer time of 2 minutes for the ICS files to be downloaded before the GUI to start parsing is more than enough.

4.1.4. keepRunning

The purpose of the script *keepRunning*, as previously mentioned, is to ensure that a given command is restarted if it terminates. The implementation of this functionality is demonstrated with a sequence diagram in Figure 4.10.

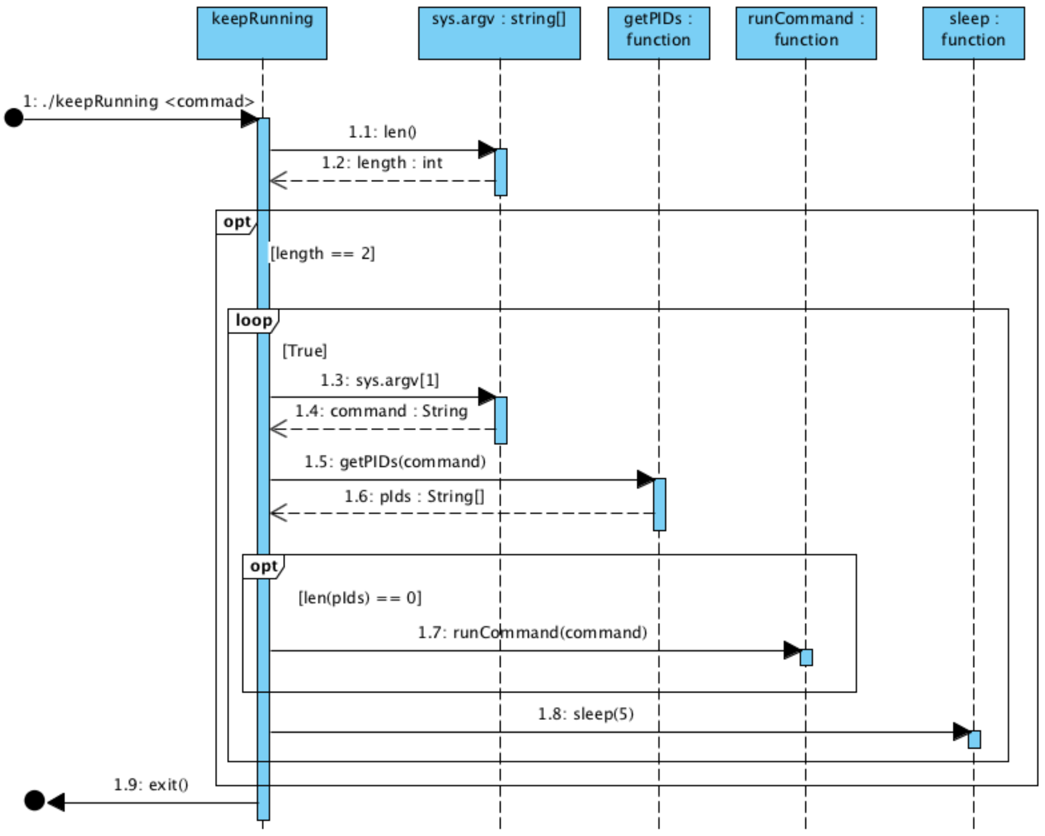


Figure 4.10.: Sequence diagram - keepRunning

As demonstrated, the script expects the command to be passed as an argument and if a command is not provided, it terminates itself. In order to perform the required check, it uses the Python library *sys* [20]. The library allows access to the arguments array containing the relative path used to execute the string as the first element and each provided argument as an element located at the following indexes starting from 1. In other words, if the length of the arguments array is not equal to exactly 2, either a command is not specified or more than one command is given, hence the script should terminate. If the mentioned check is passed, then the script starts checking if the given command is running every 5 seconds. In order to achieve the summarized functionality, the script uses the methods *getPIDs* and *runCommand*. By calling the method *getPIDs* with the command as the argument, it retrieves an array of strings containing the process ids for the given command. If the returned array is empty, then it simply calls the method *runCommand* to execute the given command. Afterwards, it sleeps for 5 seconds and repeats the same procedure again.

Test Cases

In order to test the script *keepRunning*, it is executed with the bash executable called *PcpSI* as its argument 4.3. Once the script is executed, the stdout of the linux command *ps -C PcpSI* [17] is checked and the list of processes demonstrated in Listing 4.2 is retrieved.

```
PID   TTY        TIME      CMD
8810  pts/0     00:00:00  PcpSI
```

Listing 4.2: Stdout of the command ps -C PcpSI

As can be seen, the script successfully started the process *PcpSI*. Afterwards, by killing the process with the id 8810 (process id of *PcpSI*), the behavior of the script is tested. As expected, within 5 seconds a new process with the name *PcpSI* is started having a different process id (Listing 4.3).

```
PID   TTY        TIME      CMD
8948  pts/0     00:00:00  PcpSI
```

Listing 4.3: Stdout of the command ps -C PcpSI with the new process

With the aforementioned test, it is found out that the implemented script *keepRunning* works as desired.

4.1.5. hideMouse

As mentioned earlier, this script is responsible for moving the mouse cursor to a location outside of the visible screen by using the tool *xdotool*. Unfortunately, the implemented function *runCommand* cannot be used for this purpose. Since the command needs to be executed together with the display definition, a shell is required. The Python library *os* offers a method called *system* for executing shell commands, which is used for this purpose [20]. However, it is found out that the cron command scheduled to run at the system start up gets executed before the X window system is fully established. Therefore, the *xdotool* fails to move the cursor since the cursor does not exist at the time of the execution. For this reason a sleep of 10 seconds is introduced before attempting to move the cursor. The corresponding functionality is demonstrated by a sequence diagram in Figure 4.11.

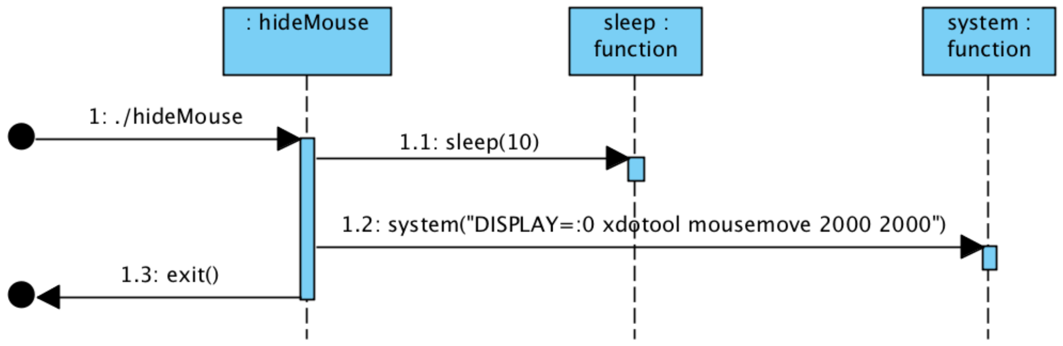


Figure 4.11.: Sequence diagram - hideMouse

4.2. Java GUI

An overview for the OOP design is demonstrated in Section 3.4.3. The reason behind this design is the desire to keep the GUI class as minimalistic as possible. The aim is to make room for possible changes on the GUI without having the necessity of altering a lot of code. With the current design of classes, the GUI is working just as a display and all the work is done by other classes. The interactive labels of the GUI are controlled by the class *StatusLabeLHandler*, while the *Controller* ensures that all the room availability information is collected with the help of the classes *RoomInfoCollector*, *DateHandler* and *FileHandler*. Even the images used inside the GUI are loaded through the *ImageHandler* class over the *Controller*. All of these make it possible for the GUI to be redesigned or even completely removed while keeping the rest of the program functional by changing very little of code. In other words, the core functionality of the front-end is independent of the class *PcpSI_GUI*.

In order to give a detailed overview of the OOP design, two class diagrams are created: one focuses on the GUI related classes such as *PcpSI_GUI*, *JComponentWithBackground*, *JImage* and *StatusLabelHandler*, while the other focuses on the *Controller* and its associations such as *DateHandler*, *FileHandler*, *ImageHandler* and *RoomInfoCollector* (Figures 4.12 and 4.13).

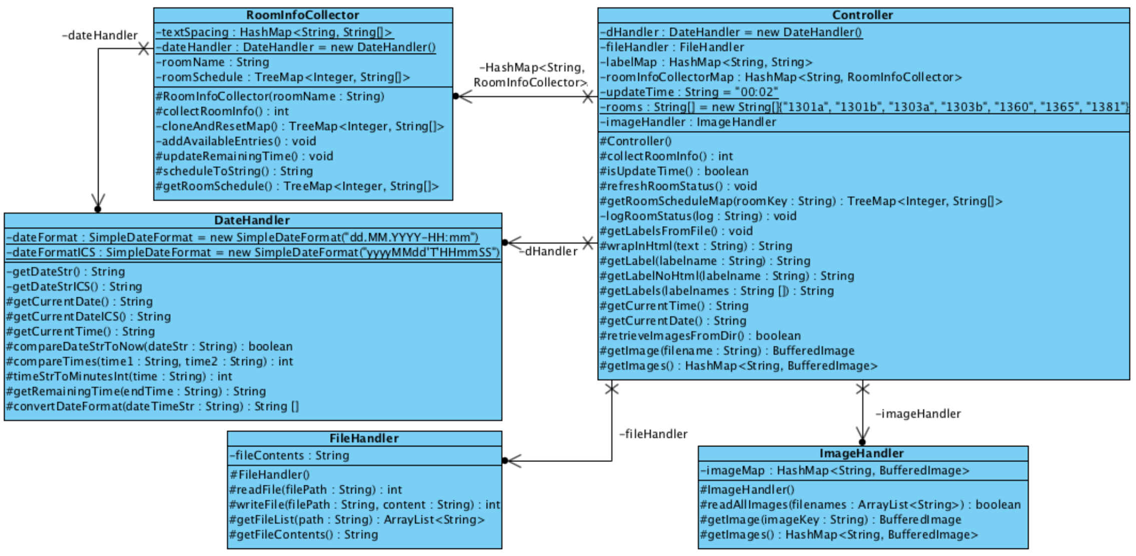

RoomInfoCollector
-textSpacing : HashMap<String, String[]>
-dateHandler : DateHandler = new DateHandler()
-roomName : String
-roomSchedule : TreeMap<Integer, String[]>
#RoomInfoCollector(roomName : String)
#collectRoomInfo() : int
-cloneAndResetMap() : TreeMap<Integer, String[]>
-addAvailableEntries() : void
#updateRemainingTime() : void
#scheduleToString() : String
#getRoomSchedule() : TreeMap<Integer, String[]>
-dateHandler
-HashMap<String, RoomInfoCollector>
Controller
-dHandler : DateHandler = new DateHandler()
-fileHandler : FileHandler
-labelMap : HashMap<String, String>
-roomInfoCollectorMap : HashMap<String, RoomInfoCollector>
-updateTime : String = "00:02"
-rooms : String[] = new String[]{"1301a", "1301b", "1303a", "1303b", "1360", "1365", "1381"}
-imageHandler : ImageHandler
#Controller()
#collectRoomInfo() : int
#isUpdateTime() : boolean
#refreshRoomStatus() : void
#getRoomScheduleMap(roomKey : String) : TreeMap<Integer, String[]>
-logRoomStatus(log : String) : void
#getLabelsFromFile() : void
#wrapInHtml(text : String) : String
#getLabel(labelname : String) : String
#getLabelNoHtml(labelname : String) : String
#getLabels(labelnames : String []) : String
#getCurrentTime() : String
#getCurrentDate() : String
#retrieveImagesFromDir() : boolean
#getImage(filename : String) : BufferedImage
#getImages() : HashMap<String, BufferedImage>
DateHandler
-dateFormat : SimpleDateFormat = new SimpleDateFormat("dd.MM.YYYY-HH:mm")
-dateFormatICS : SimpleDateFormat = new SimpleDateFormat("yyyyMMdd'T'HHmmSS")
-getDateStr() : String
-getDateStrICS() : String
#getCurrentDate() : String
#getCurrentDateICS() : String
#getCurrentTime() : String
#compareDateStrToNow(dateStr : String) : boolean
#compareTimes(time1 : String, time2 : String) : int
#timeStrToMinutesInt(time : String) : int
#getRemainingTime(endTime : String) : String
#convertDateFormat(dateTimeStr : String) : String []
-dHandler
-fileHandler
-imageHandler
FileHandler
-fileContents : String
#FileHandler()
#readFile(filePath : String) : int
#writeFile(filePath : String, content : String) : int
#getFileList(path : String) : ArrayList<String>
#getFileContents() : String
ImageHandler
-imageMap : HashMap<String, BufferedImage>
#ImageHandler()
#readAllImages(filenames : ArrayList<String>) : boolean
#getImage(imageKey : String) : BufferedImage
#getImages() : HashMap<String, BufferedImage>

Figure 4.12.: Class diagram - Controller and its associations

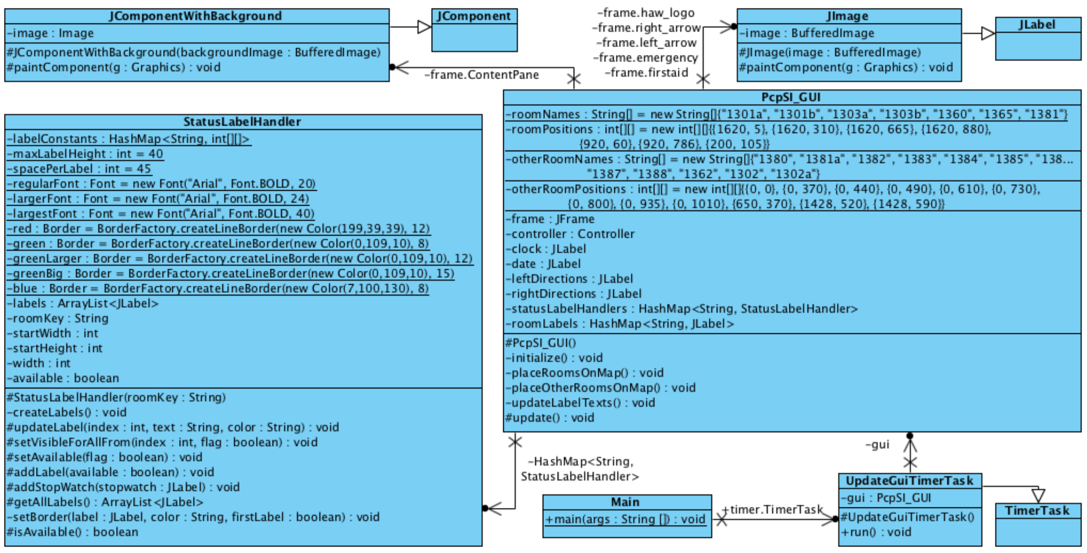

JComponentWithBackground
-image : Image
#JComponentWithBackground(backgroundImage : BufferedImage)
#paintComponent(g : Graphics) : void
JComponent
-frame.ContentPane
-frame.haw_logo
-frame.right_arrow
-frame.left_arrow
-frame.emergency
-frame.firstaid
JImage
-image : BufferedImage
#JImage(image : BufferedImage)
#paintComponent(g : Graphics) : void
JLabel
StatusLabelHandler
-labelConstants : HashMap<String, int[][]>
-maxLabelHeight : int = 40
-spacePerLabel : int = 45
-regularFont : Font = new Font("Arial", Font.BOLD, 20)
-largerFont : Font = new Font("Arial", Font.BOLD, 24)
-largestFont : Font = new Font("Arial", Font.BOLD, 40)
-red : Border = BorderFactory.createLineBorder(new Color(199,39,39), 12)
-green : Border = BorderFactory.createLineBorder(new Color(0,109,10), 8)
-greenLarger : Border = BorderFactory.createLineBorder(new Color(0,109,10), 12)
-greenBig : Border = BorderFactory.createLineBorder(new Color(0,109,10), 15)
-blue : Border = BorderFactory.createLineBorder(new Color(7,100,130), 8)
-labels : ArrayList<JLabel>
-roomKey : String
-startWidth : int
-startHeight : int
-width : int
-available : boolean
#StatusLabelHandler(roomKey : String)
-createLabels() : void
#updateLabel(index : int, text : String, color : String) : void
#setVisibleForAllFrom(index : int, flag : boolean) : void
#setAvailable(flag : boolean) : void
#addLabel(available : boolean) : void
#addStopWatch(stopwatch : JLabel) : void
#getAllLabels() : ArrayList<JLabel>
-setBorder(label : JLabel, color : String, firstLabel : boolean) : void
#isAvailable() : boolean
PcpSI_GUI
-roomNames : String[] = new String[]{"1301a", "1301b", "1303a", "1303b", "1360", "1365", "1381"}
-roomPositions : int[][] = new int[][]{{1620, 5}, {1620, 310}, {1620, 665}, {1620, 880}, {920, 60}, {920, 786}, {200, 105}}
-otherRoomNames : String[] = new String[]{"1380", "1381a", "1382", "1383", "1384", "1385", "138... "1387", "1388", "1362", "1302", "1302a"}
-otherRoomPositions : int[][] = new int[][]{{0, 0}, {0, 370}, {0, 440}, {0, 490}, {0, 610}, {0, 730}, {0, 800}, {0, 935}, {0, 1010}, {650, 370}, {1428, 520}, {1428, 590}}
-frame : JFrame
-controller : Controller
-clock : JLabel
-date : JLabel
-leftDirections : JLabel
-rightDirections : JLabel
-statusLabelHandlers : HashMap<String, StatusLabelHandler>
-roomLabels : HashMap<String, JLabel>
#PcpSI_GUI()
-initialize() : void
-placeRoomsOnMap() : void
-placeOtherRoomsOnMap() : void
-updateLabelTexts() : void
#update() : void
-HashMap<String, StatusLabelHandler>
-gui
Main
+main(args : String []) : void
+timer.TimerTask
UpdateGuiTimerTask
-gui : PcpSI_GUI
#UpdateGuiTimerTask()
+run() : void
TimerTask

Figure 4.13.: Class diagram - GUI and its associations

As can be seen in the class diagram demonstrated in Figure 4.13, *UpdateGuiTimerTask* contains the *PcpSI_GUI* and *PcpSI_GUI* contains the *Controller*. When a timer interrupt occurs, *UpdateGuiTimerTask* calls the *update* method of the *PcpSI_GUI* and *PcpSI_GUI* collects the updated information from the *Controller*. If desired, the class *PcpSI_GUI* can be removed and by associating the *Controller* with the *UpdateGuiTimerTask*, the program would continue to function as it does without a graphical user interface. The *Controller* object is implemented in a way that it logs all the information made available to the GUI, hence, removing the GUI would turn the program into an API. This also means that the program with its current implementation together with its GUI can also be used as an API, which is explained in detail in Section 5.1.1.

4.2.1. Detailed View of Classes

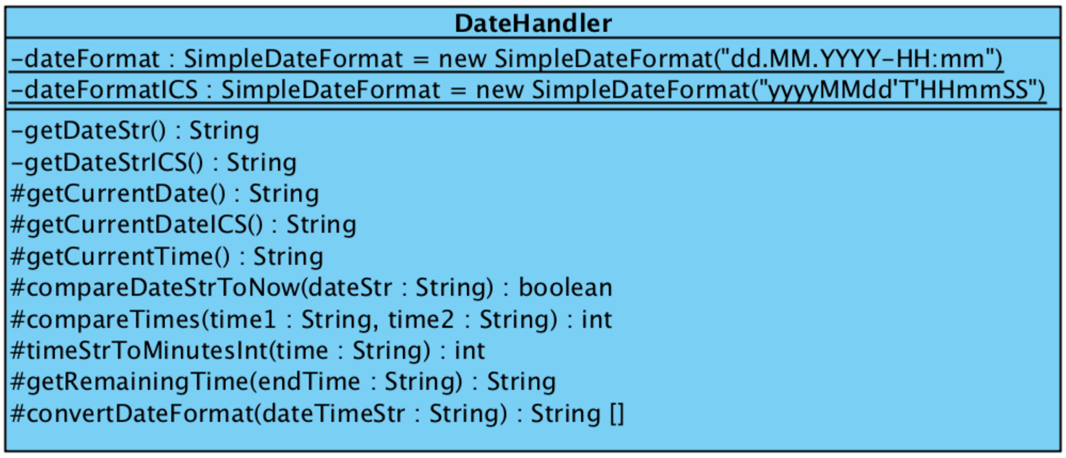
The implementation details of the classes shown in the class diagrams from Figures 4.12 and 4.13 are explained in detail within this Section. The order is set to be from the lower layer classes (handlers) to the upper layer classes (GUI). The reason behind this ordering is to provide a stable understanding of how the core works before explaining the top layer by making references to the core. All the methods and attributes of each class are explained in order to provide a complete overview, which is necessary for demonstrating the realization of the application.

DateHandler

As previously mentioned during the design process (Section 3.4.3), the class *DateHandler* is responsible for handling tasks related to date and time. For this purpose it needs to be familiar with two specific date formats:

- ICS format: yyyyMMdd'T'HHmmSS
- GUI format: dd.MM.YYYY-HH:mm

The main purpose of having two separate date formats is to convert the date format present in the ICS files to a more readable format. The class diagram of the class *DateHandler* showing its attributes and implemented methods is demonstrated in Figure 4.14.



```
DateHandler
-dateFormat : SimpleDateFormat = new SimpleDateFormat("dd.MM.YYYY-HH:mm")
-dateFormatICS : SimpleDateFormat = new SimpleDateFormat("yyyyMMdd'T'HHmmSS")
-getDateStr() : String
-getDateStrICS() : String
#getCurrentDate() : String
#getCurrentDateICS() : String
#getCurrentTime() : String
#compareDateStrToNow(dateStr : String) : boolean
#compareTimes(time1 : String, time2 : String) : int
#timeStrToMinutesInt(time : String) : int
#getRemainingTime(endTime : String) : String
#convertDateFormat(dateTimeStr : String) : String []
```

Figure 4.14.: Class diagram - DateHandler

As can be seen in the class diagram, both attributes of this class are private, static and final. The reason why the access level is set to be private is that these date formats are not required by any of the associations of the class, in other words they are only defined to be used within the class. Since the intention is to keep these formats the same at all times within all instances of the class, they are set to be static and final. The following methods are implemented to give the necessary functionality to the class:

- getDateStr(): Returns the current date and time with the GUI date format. Access level is set to be private because none of this class' associations require the raw date string.
- getDateStrICS(): Similar to the method *getDateStr*, it returns the current date and time; however, the returned date string is formatted with the ICS date format. For the same reason as in the *getDateStr*, this method is set to be private.

- getCurrentDate(): Extracts the current date from the raw date string that it retrieves by calling the method *getDateStr*. Since this method is used by the class' associations, access level is defined to be protected.
- getCurrentDateICS(): Does the exact same thing as the method *getCurrentDate*; however, the returned date has the ICS date format. Access level is also set to be protected for the same reason as in *getCurrentDate*.
- getCurrentTime(): Similar to methods that return a string that contains the date, this method extracts the current time from the raw date string. Access level is set to be protected for the same reasons.
- compareDateStrToNow(dateStr : String): This method is implemented to compare a given date string to the one retrieved from the method *getCurrentDateICS*. It is mainly used to filter out events from an ICS file. The method returns true if the compared dates are a match. Since the method is required to be accessed from other classes, access level is set to be protected.
- compareTimes(time1 : String, time2 : String): As its name suggests, it compares two given time strings. Return value is an integer: 0 for time1 equals time2, 1 for time1 greater than time2 and -1 for time1 less than time2. This method is implemented so that it can be checked if an event, with its start or end time given, has started or ended.
- timeStrToMinutesInt(time : String): Converts a given time string to an integer containing the total minutes. For example if the method is called with the time string "10:30", it returns 10*60 + 30. The reason for implementing this method is to calculate remaining time of an event before it starts or ends.
- getRemainingTime(endTime : String): As its name suggests, this method calculates the remaining time for a given endTime. First it retrieves the minutes representation of the current time and the given time by calling the method *timeStrToMinutesInt*. Afterwards, by subtracting the retrieved integer values, it calculates the remaining time. As the last step, the remaining time in minutes is converted to "hh:mm" format and returned as a string so it can be displayed on the GUI.
- convertDateFormat(dateTimeStr : String): Given raw date string with ICS date format is converted into GUI date format. The method splits the resulting string into date and time and returns an array composed of these strings. The reason for the implementation of this method is that the raw date strings retrieved from the ICS files are not easy to read, so they need to be converted into a more readable format.

FileHandler

As its name suggests and as previously mentioned in Section 3.4.3, the class *FileHandler* is responsible for executing read and write operations on files. The necessity for the class comes from the room availability information being stored in ICS files. An overview of the class is demonstrated with a class diagram in Figure 4.15.

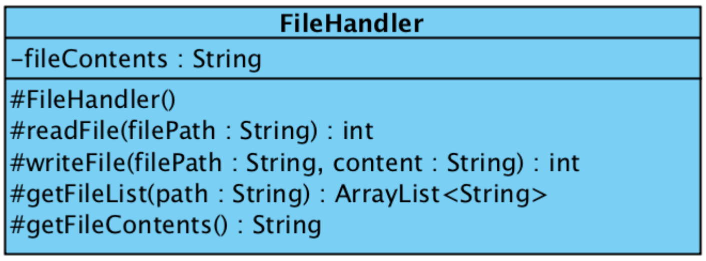

FileHandler
-fileContents : String
#FileHandler()
#readFile(filePath : String) : int
#writeFile(filePath : String, content : String) : int
#getFileList(path : String) : ArrayList<String>
#getFileContents() : String

Figure 4.15.: Class diagram - FileHandler

As can be seen in the class diagram, *FileHandler* has only one attribute: *fileContents : String*. After a read operation on a given file is executed, the contents of the file are assigned to this attribute as a string. Access level of the attribute is set to be private and a getter is implemented so that the code is more readable when a read operation is called on a file. The following methods are implemented to give the desired functionality to the class:

- readFile(filePath : String): Reads the file located at the given path and assigns the contents to the attribute *fileContents*. The method returns an integer to indicate the status of the read operation: 0 for FileNotFoundException, -1 for IOException (in other words file is found at the given path but cannot be read), 1 for indicating that the read process is executed successfully. Implementation of this method is mainly based on the need for reading ICS files.

- writeFile(filePath : String, content : String): Writes the given content to a file at the given path. Similar to the *readFile* method, this method also returns an integer to indicate the status of the write process: -1 for IOException, 1 for write process is successful. The main reason of implementing this method is to give the ability to store all the information displayed on the GUI to log files. Having this information also present in a file makes debugging easier as well as making the system less dependent on the GUI by offering an API functionality.

- getFileList(path : String): The purpose of the method is to return a list of files at a given directory specified by the path. It returns an *ArrayList* of strings with each string containing the absolute path of a file located at the given path. The method is implemented so that every image from the *images* directory can be loaded to the GUI.

ImageHandler

This class is implemented to provide the functionality of reading image files from the *images* directory. The necessity of this class comes from the fact that the background of the GUI is created to be just a plain image of the floor plan. For this reason all the additional images are required to be loaded. The main purpose behind this is to make things more customizable; which at the end directly affects the lifespan of the system. An overview of the class can be seen in Figure 4.16.

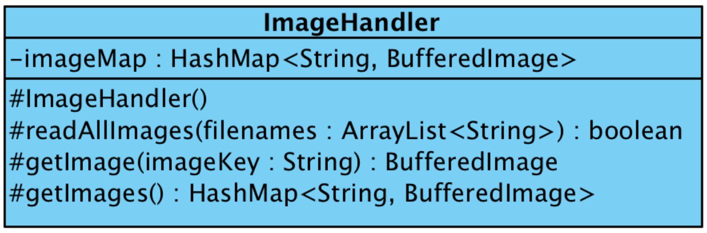

ImageHandler
-imageMap : HashMap<String, BufferedImage>
#ImageHandler()
#readAllImages(filenames : ArrayList<String>) : boolean
#getImage(imageKey : String) : BufferedImage
#getImages() : HashMap<String, BufferedImage>

Figure 4.16.: Class diagram - ImageHandler

As can be seen in the class diagram, the only attribute this class has is a *HashMap* of *String* : *BufferedImage* pairs. The keys in the map are defined to be image file names and the values are the images loaded into Java as *BufferedImage* objects. The map is implemented to be private and two getters are implemented to make the code more readable. As it can also be seen in the class diagram, the following methods are implemented to give the desired functionality to the class:

- readAllImages(filenames : ArrayList<String>): Iterates through the list of image file names given and loads the corresponding images from the *images* directory into *BufferedImage* objects, which are then put into the map with keys as their file name. This method is implemented so that all the images located in the *images* directory are loaded into Java with just one function call.
- getImage(imageKey : String): Returns the *BufferedImage* associated with the given key from the image map. By the help of this method, the GUI can load a specific image when needed instead of getting a reference to the entire map.
- getImages(): Returns a reference object to the image map containing all the loaded images.

RoomInfoCollector

As previously explained in Section 3.4.3, this class is responsible for retrieving the necessary information from an ICS file with the help of a *FileHandler* object. An instance of this class is created for each ICS file so that the event list of each room is contained in a separate object. It also offers the functionality to provide necessary spacing to be added to the information retrieved so that the GUI can use the information as it is without altering. This ability is added to this class to minimize the dependency on the GUI.

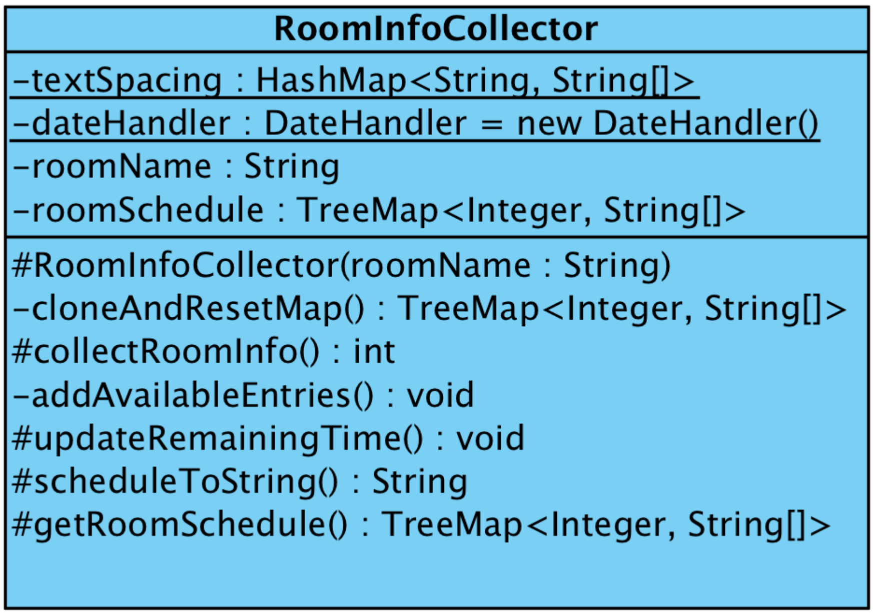

RoomInfoCollector
-textSpacing : HashMap<String, String[]>
-dateHandler : DateHandler = new DateHandler()
-roomName : String
-roomSchedule : TreeMap<Integer, String[]>
#RoomInfoCollector(roomName : String)
-cloneAndResetMap() : TreeMap<Integer, String[]>
#collectRoomInfo() : int
-addAvailableEntries() : void
#updateRemainingTime() : void
#scheduleToString() : String
#getRoomSchedule() : TreeMap<Integer, String[]>

Figure 4.17.: Class diagram - RoomInfoCollector

There are 2 alternatives to achieve the aforementioned goal of the class above: either by creating the schedule information as a complete string ready to be directly set to a label, or by putting all the necessary event information into an array of strings which can be used to formulate the final label text at an upper layer. The first option would indeed minimize the dependency on the GUI, however it would not offer enough room for customization. Customization is important in case certain updates on the GUI view need to be made. As also previously mentioned, a longer life span of the system can only be achieved by allowing customization and for this reason the second option is taken. Therefore, this class provides event information in the form of an array of strings which can be used to create final label texts at an upper layer. An overview of the class can be seen in Figure 4.17.

As demonstrated in the class diagram, there are 3 attributes:

- textSpacing: A private, static and final attribute of type *HashMap<String, String[]>*. Specific text spacings for each room are set within this map. Keys are defined to be room names so that each instance of the class can retrieve the required information by just using their *roomName* attribute.
- dateHandler: Another private, static and final attribute. Since all attributes of the *dateHandler* class are static and final, this attribute of *RoomInfoCollector* class has no chance of getting altered. Therefore, it is set to be static and final. This attribute is used to compute some date time related information regarding the events collected from the ICS files.
- roomName: A private attribute that is accessed only within this class. It is used as an identification for each instance of the class.
- roomSchedule: A private attribute realized in the form of a *TreeMap*. Different than a *HashMap*, entries in the *TreeMap* objects are sorted by their keys [25]. As it is explained in the design process of the GUI layout in Section 3.4.1, events belonging to each room are displayed in a sorted form starting with the minimum remaining time. For this reason the *TreeMap* object offers the desired behavior. The retrieved list of events is sorted within this map by their start times so that the GUI can directly display the information provided without any alteration. As mentioned earlier, these kind of functionalities are provided to minimize the dependency on the GUI.

In order to completely deliver the desired functionality of this class, the following methods are implemented:

- cloneAndResetMap(): A private method intended to be used only within the class. Makes a clone to be returned and resets the *TreeMap* attribute *roomSchedule*. The purpose of implementing this method is to make the code more readable by allowing cloning and resetting of the map by one function call.
- updateRemaniningTime(): This method is implemented to update the remaining times of the scheduled events according to the current time. This method, being one of the key methods in the display process of the scheduled events on the GUI, is explained in detail. For this purpose, the functionality is demonstrated with a sequence diagram in Figure 4.18.

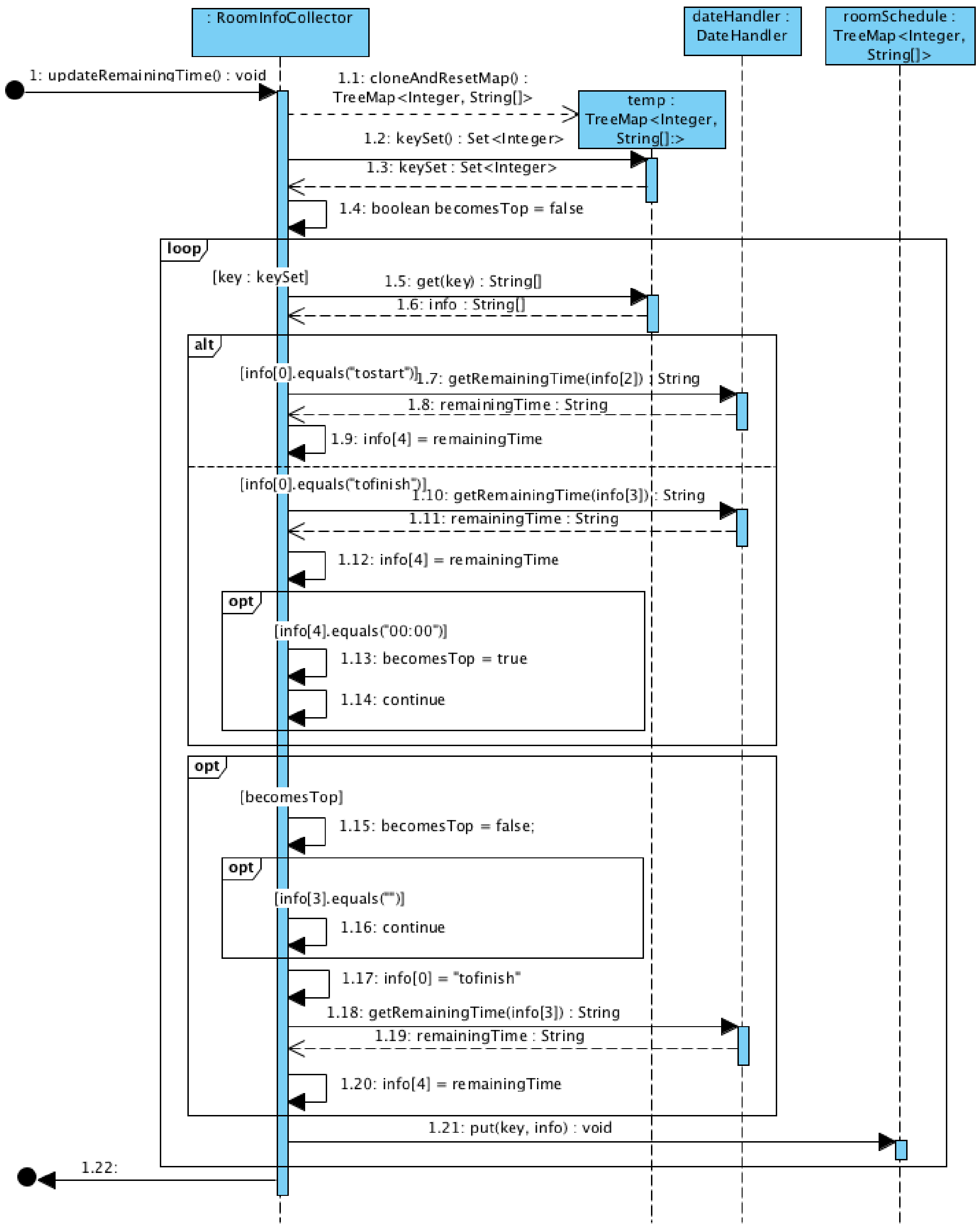

: RoomInfoCollector
dateHandler : DateHandler
roomSchedule : TreeMap<Integer, String[]>
1: updateRemainingTime() : void
1.1: cloneAndResetMap() : TreeMap<Integer, String[]>
temp : TreeMap<Integer, String[]:>
1.2: keySet() : Set<Integer>
1.3: keySet : Set<Integer>
1.4: boolean becomesTop = false
loop
[key : keySet]
1.5: get(key) : String[]
1.6: info : String[]
alt
[info[0].equals("tostart")]
1.7: getRemainingTime(info[2]) : String
1.8: remainingTime : String
1.9: info[4] = remainingTime
[info[0].equals("tofinish")]
1.10: getRemainingTime(info[3]) : String
1.11: remainingTime : String
1.12: info[4] = remainingTime
opt
[info[4].equals("00:00")]
1.13: becomesTop = true
1.14: continue
opt
[becomesTop]
1.15: becomesTop = false;
opt
[info[3].equals("")]
1.16: continue
1.17: info[0] = "tofinish"
1.18: getRemainingTime(info[3]) : String
1.19: remainingTime : String
1.20: info[4] = remainingTime
1.21: put(key, info) : void
1.22:

Figure 4.18.: Sequence diagram - updateRemainingTime()

As can be seen in the demonstrated sequence diagram, the method first calls *cloneAndResetMap()* to retrieve a clone of the attribute *roomSchedule* and reset it afterwards. Then it iterates through the scheduled events within the clone and updates their remaining times. If the event is registered to start at a given time (indicated by the index 0 of its schedule information array of strings - *info*), then the remaining time is updated according to its start time (indicated by the index 2). Additionally, if the event is an ongoing event, then the remaining time is updated according to the event's end time (indicated by index 3). Afterwards, it is checked whether the updated remaining time is equal to "00:00", which indicates that the event has finished. If that is the case, then a flag is set and the event is skipped by calling *continue* on the loop. This flag is used to tell the next event that it should become the top event since the current event has finished. However, the flag is ignored if the next event is the last event in the queue. This is indicated by the end time of an event being set to an empty string. Event lists are appended by an entry indicating from what time onwards the room is available and these entries have no end time. This is the reason why the flag is ignored. If the flag is found to be *true* and the event has a valid end time, then the event information (index 0) is set to indicate that this is an ongoing event by the string "tofinish". Afterwards, the remaining time is updated accordingly and the event is put into the *roomSchedule* map.

- getRoomSchedule(): Returns a reference to the *TreeMap* attribute *roomSchedule*. This method is implemented so that the event list is available within the package.
- addAvailableEntries(): Another private method intended to be used only within the class. Iterates through the events in the *TreeMap* attribute *roomSchedule* and checks the time gaps between events. If a time gap of 15 minutes or more is found, then an event with its summary set to "available" is added covering the gap. The purpose here is to indicate if the room is available between two events together with the availability duration. This is also done if there is no ongoing event to cover the gap between the current time and the next event to start. Additionally, as mentioned earlier, an event indicating from what time on the room is available with an empty end time is added to the end of the list as well. This method is implemented as a part of the class *RoomInfo-Collector* so that the dependency on the GUI can be minimized. Available information is fully indicated before the event list reaches the GUI, allowing the information to be displayed without any alteration.
- collectRoomInfo(): Reads the ICS file associated with the given *roomName*. Uses a *FileHandler* object for this purpose. Considering that this is a key method for the functionality of the Java application, the functionality of this method is demonstrated with a sequence diagram in Figure 4.19.

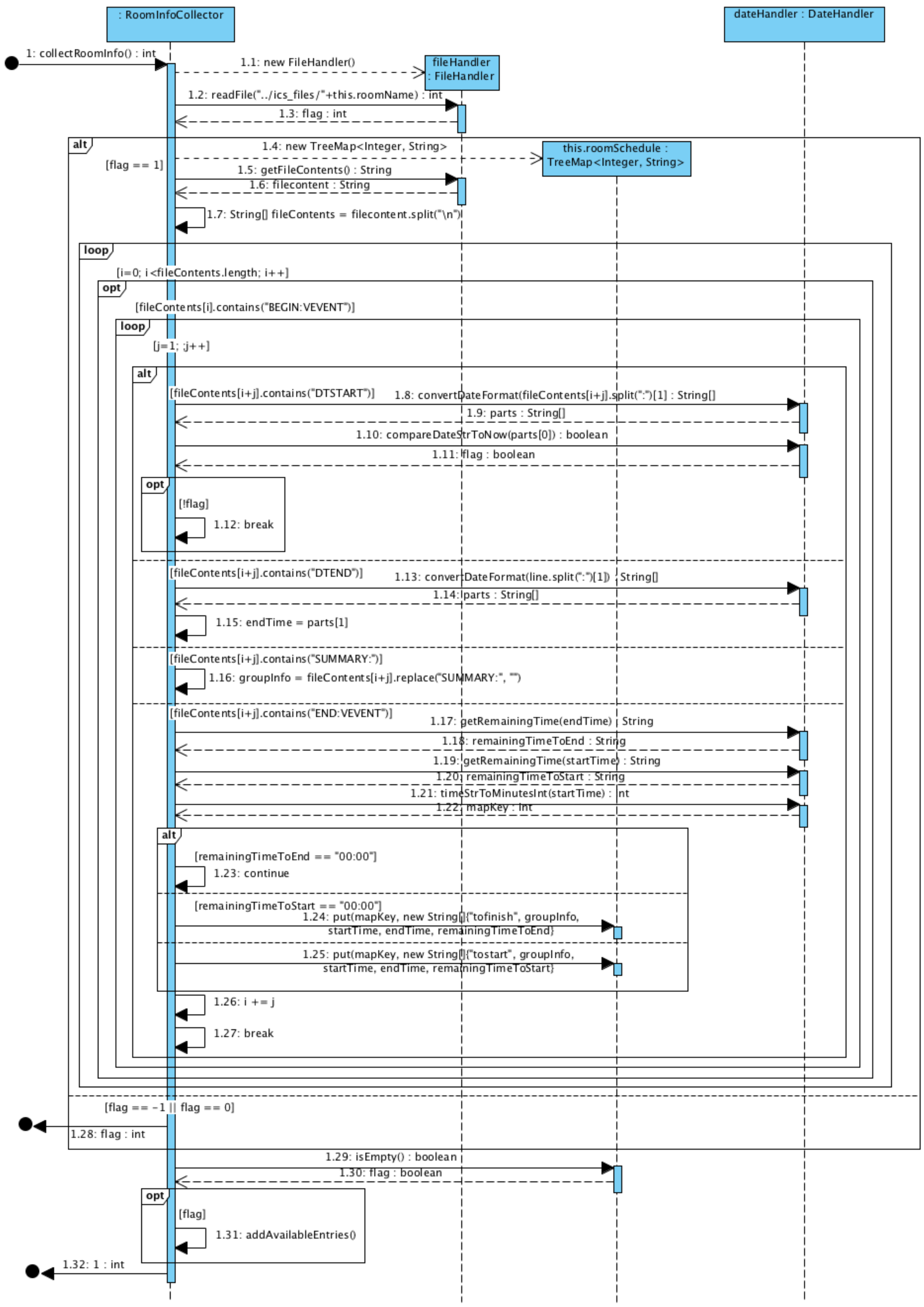

: RoomInfoCollector
dateHandler : DateHandler
1: collectRoomInfo() : int
1.1: new FileHandler()
fileHandler : FileHandler
1.2: readFile("../ics_files/"+this.roomName) : int
1.3: flag : int
alt
[flag == 1]
1.4: new TreeMap<Integer, String>
this.roomSchedule : TreeMap<Integer, String>
1.5: getFileContents() : String
1.6: filecontent : String
1.7: String[] fileContents = filecontent.split("\n")
loop
[i=0; i<fileContents.length; i++]
opt
[fileContents[i].contains("BEGIN:VEVENT")]
loop
[j=1; ;j++]
alt
[fileContents[i+j].contains("DTSTART")]
1.8: convertDateFormat(fileContents[i+j].split(":")[1] : String[]
1.9: parts : String[]
1.10: compareDateStrToNow(parts[0]) : boolean
1.11: flag : boolean
opt
[!flag]
1.12: break
[fileContents[i+j].contains("DTEND")]
1.13: convertDateFormat(line.split(":")[1]) : String[]
1.14: parts : String[]
1.15: endTime = parts[1]
[fileContents[i+j].contains("SUMMARY:")]
1.16: groupInfo = fileContents[i+j].replace("SUMMARY:", "")
[fileContents[i+j].contains("END:VEVENT")]
1.17: getRemainingTime(endTime) : String
1.18: remainingTimeToEnd : String
1.19: getRemainingTime(startTime) : String
1.20: remainingTimeToStart : String
1.21: timeStrToMinutesInt(startTime) : int
1.22: mapKey : int
alt
[remainingTimeToEnd == "00:00"]
1.23: continue
[remainingTimeToStart == "00:00"]
1.24: put(mapKey, new String[]{"tofinish", groupInfo, startTime, endTime, remainingTimeToEnd}
1.25: put(mapKey, new String[]{"tostart", groupInfo, startTime, endTime, remainingTimeToStart}
1.26: i += j
1.27: break
[flag == -1 || flag == 0]
1.28: flag : int
1.29: isEmpty() : boolean
1.30: flag : boolean
opt
[flag]
1.31: addAvailableEntries()
1.32: 1 : int

Figure 4.19.: Sequence diagram - collectRoomInfo()

As demonstrated in the sequence diagram, this method iterates through the ICS file that is read and looks for the keyword *BEGIN:VEVENT*. Once found, it parses the start date and time from the line with the keyword *DTSTART*. Afterwards it checks if the event is occurring on that day. If the dates are found to be a match, the summary and the end time of the event are parsed from the lines with the keywords *SUMMARY* and *DTEND* correspondingly. Once the event block is read fully, all the event related information is put into an array of strings. This array of strings contains the event status in its index 0. For an ongoing event, this field is set to "tofinish" and for an event that has not started, it is set to "tostart". Index 1 of the array contains the event summary and the start - end times are placed in the indexes 2 and 3 respectively. Index 4, which is the last index in the array, contains the remaining time of the event (remaining time either event to start or to finish as indicated by the index 0). Once the mentioned array of strings is set, it is placed into the *TreeMap* attribute *roomSchedule* with the key start time converted into minutes in an integer form. After all the events are parsed, the method *addAvailableEntries* is called to populate the list with available event entries where applicable.

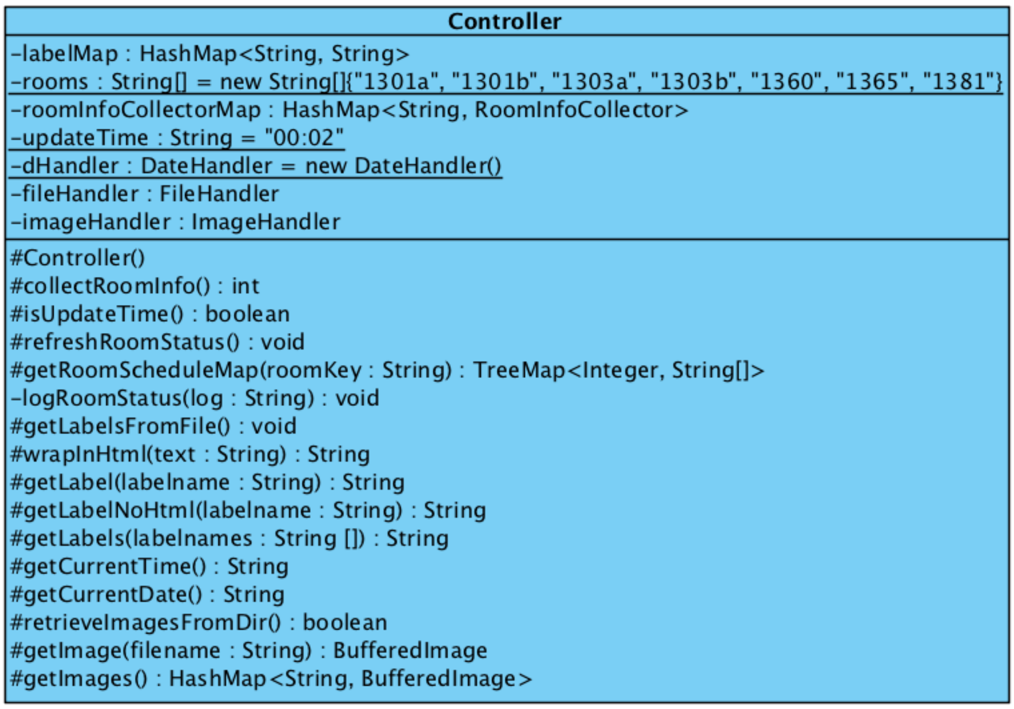
- scheduleToString(): Converts the *TreeMap* attribute *roomSchedule* into a table representation with the format demonstrated in Table 4.3. The demonstrated headers are not included in the string returned by this method. A new row is indicated by "`\n`" where a new column is indicated by "`\t`" within the string. The purpose of providing the demonstrated functionality is to add the ability to log event information shown in the GUI to a file. This way, the system can be used as an API.

Room name	Event summary	Start time	End time	Remaining time
1360	AVAILABLE		12:10	01:15
	E1b-PRP1/02	12:10	15:40	01:15
	AVAILABLE	15:40		04:45

Table 4.3.: Example to an event list table representation returned from the method RoomInfoCollector.scheduleToString()

Controller

The main aim of the class *Controller* is to provide all the necessary event related information of the rooms to the GUI. For this purpose it uses the classes *DateHandler* and *RoomInfoCollector*. The reason for implementing this class is to minimize the number of classes required from a top layer such as GUI. With the help of this class, the GUI needs just one class from the package in order to collect all the necessary information. Minimizing the number of classes required from a top layer makes the top layer more compact, hence minimizing the dependency on the top layer itself. The corresponding class diagram is demonstrated in Figure 4.20.



```
Controller
-labelMap : HashMap<String, String>
-rooms : String[] = new String[]{"1301a", "1301b", "1303a", "1303b", "1360", "1365", "1381"}
-roomInfoCollectorMap : HashMap<String, RoomInfoCollector>
-updateTime : String = "00:02"
-dHandler : DateHandler = new DateHandler()
-fileHandler : FileHandler
-imageHandler : ImageHandler
#Controller()
#collectRoomInfo() : int
#isUpdateTime() : boolean
#refreshRoomStatus() : void
#getRoomScheduleMap(roomKey : String) : TreeMap<Integer, String[]>
-logRoomStatus(log : String) : void
#getLabelsFromFile() : void
#wrapInHtml(text : String) : String
#getLabel(labelname : String) : String
#getLabelNoHtml(labelname : String) : String
#getLabels(labelnames : String []) : String
#getCurrentTime() : String
#getCurrentDate() : String
#retrieveImagesFromDir() : boolean
#getImage(filename : String) : BufferedImage
#getImages() : HashMap<String, BufferedImage>
```

Figure 4.20.: Class diagram - Controller

As can be seen in the class diagram, all attributes are set to be private. The reason is to ensure that the getters are used from the top layer, making the code more readable. There are a total of 7 attributes:

- labelMap: An attribute of type *HashMap<String, String>* with key value pairs of label identifier and the label text. This attribute is used to store the label texts that are read from the config file *labels*.

- rooms: A static final attribute of type array of strings. Contains the room identifiers that are to be used for instantiating *RoomInfoCollector* objects pointing to ICS files with names corresponding to the identifiers from this array.
- roomInfoCollectorMap: This attribute is of type *HashMap<String, RoomInfoCollector>*. Keys of the map are the room names and the values are the corresponding *RoomInfo-Collector* objects. The reason behind having this map implemented is to give the top layer the ability to to access the event list of a room by passing the room identifier to a *Controller* object. With the help of the *HashMap*, the *Controller* would not need to iterate through an array or an *ArrayList* and can access the desired *RoomInfoCollector* object by using the room identifier.
- updateTime: Another static final attribute. This attribute defines the time that the ICS files are re-parsed by the *RoomInfoCollector* objects.
- dHandler: The *DateHandler* object is set to be a static and final attribute to the class *Controller*. The reason behind this way of implementation is, as mentioned earlier, that the *DateHandler* object has no chance of being altered during the program execution since it has no attributes.
- fileHandler: An attribute of type *FileHandler*, which is used to collect labels from the config file *labels*, as well as logging the information passed to the GUI into a file.
- imageHandler: This attribute of type *ImageHandler* is used to retrieve images from the *images* directory and provide them to the GUI.

In order to ensure that the top layer has access to all the necessary information through the *Controller* class, the following methods are implemented as wrappers to certain methods provided by the classes *DateHandler*, *ImageHandler* and *RoomInfoCollector*:

- getCurrentTime(): A wrapper to the method *DateHandler.getCurrentTime()* that calls the *getCurrentTime* method and returns the time string retrieved. This information is required by the GUI for displaying the current time.
- getCurrentDate(): Another wrapper similar to the *getCurrentTime* that calls the *Date-Handler.getCurrentDate()* and returns the retrieved date string. This method is implemented so that the GUI class has access to current date through the *Controller* object.
- getImage(fileName : String): Calls the method *ImageHandler.getImage* with the argument *fileName* and returns the retrieved *BufferedImage*. Allows the GUI to have access to a specific image from the *images* directory.

- getImages(): Similar to the method *getImage*. Instead of returning a single image from the *images* directory, it calls *ImageHandler.getImages()* and returns the *HashMap* retrieved, which contains all the images from the *images* directory.
- getRoomScheduleMap(roomKey : String): Calls the *getRoomSchedule()* of the object *RoomInfoCollector* identified by the *roomKey* provided. Returns the *TreeMap* retrieved from the mentioned function call.

In addition to the mentioned wrapper methods, the following methods are implemented to fully provide the desired functionality to the *Controller* class:

- getLabelsFromFile(): This method is implemented to read the configuration file *labels* and populate the attribute *labelMap* with the collected label texts. Uses the *fileHandler* to retrieve the content of the mentioned configuration file. Iterates through the lines and adds each line defining a label text into the *labelMap* with room identifiers as keys and label texts as values.
- isUpdateTime(): Retrieves the current time by using the *dHandler* attribute and compares it to the time defined in *updateTime* string. If the strings are a match, which indicates that it is time for re-parsing the ICS files, it returns true. Otherwise false is returned. This method is implemented so that the top layer has the ability to tell whether it is time for an update or not by a function call.
- logRoomStatus(log : String): This method is implemented to log the room schedules into a file by using the attribute *fileHandler*. It calls the *FileHandler.writeFile* method with the arguments demonstrated in Listing 4.4.

```
1 "../logs/room_status.log"
2 "room\tsummary\t\t\tstart\tend\tremaining\n"+log
```

Listing 4.4: Arguments for logging the room schedules

As can be seen in Listing 4.4, the first line defines the path for the log file and with the passed string the schedule log is stored in the *logs* directory with the name *room_status.log*. As mentioned earlier this log file has the structure demonstrated in Table 4.3 and the headers are not included in the string created by the method *RoomInfoCollector.scheduleToString*. Therefore, as can be seen in line 2, the headers are added separated by a tab space in front of the table body passed to this method as an argument. In order to give more room to the summary column, the separation is set to be 3 tab spaces instead of 2.

- collectRoomInfo(): Reconstructs the *roomInfoCollectorMap* so that the ICS files are re-parsed. Iterates through the static array of strings *rooms*, and instantiates a *RoomCollectorInfo* object with a room identifier from the array. For each instantiated object, the method *collectRoomInfo* is called, populating the *scheduleMap* of the object. Once this procedure is successfully completed, each object is added to the *roomInfoCollectorMap* with the room identifier as the key. The corresponding functionality is demonstrated with a sequence diagram in Figure 4.21.

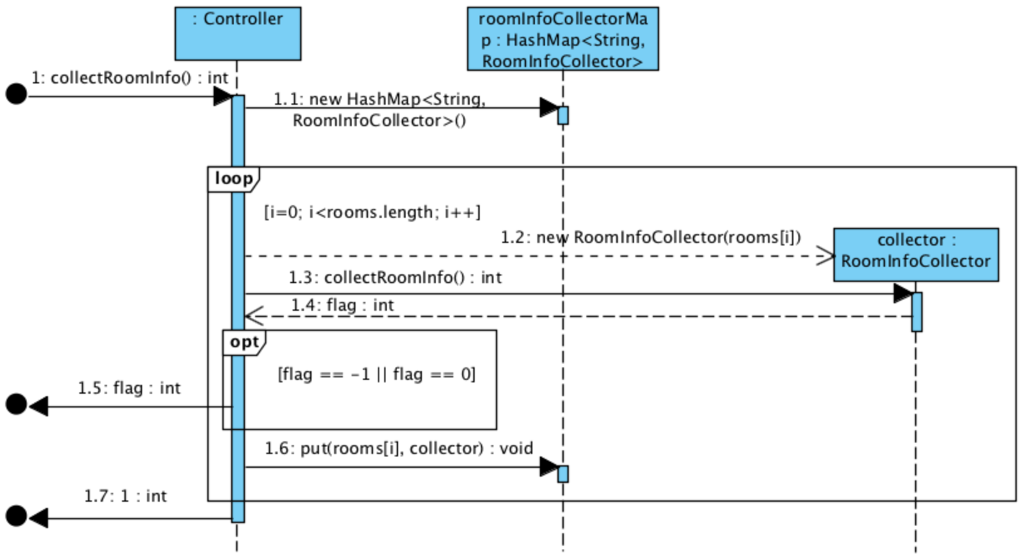

: Controller
roomInfoCollectorMap : HashMap<String, RoomInfoCollector>
1: collectRoomInfo() : int
1.1: new HashMap<String, RoomInfoCollector>()
loop
[i=0; i<rooms.length; i++]
1.2: new RoomInfoCollector(rooms[i])
collector : RoomInfoCollector
1.3: collectRoomInfo() : int
1.4: flag : int
opt
[flag == -1 || flag == 0]
1.5: flag : int
1.6: put(rooms[i], collector) : void
1.7: 1 : int

Figure 4.21.: Sequence diagram - Controller.collectRoomInfo()

- wrapInHtml(text : String): This method is responsible for converting a given text into a html code. For this purpose, new line character sequences within the given text are replaced by *<br>* [5]. The reason behind implementing this method roots from the fact that *JLabel* texts can only be aligned by html code [24]. For this purpose the following html code is used to align the text of a label to the center, which is demonstrated in Listing 4.5. Any given string is wrapped with the demonstrated html code and returned by this method.

```
<html><div style="text-align: center;"> <text-goes-in-here> </
    html>
```

Listing 4.5: Html code for aligning texts in a JLabel

- getLabel(labelname : String): Allows the top layer to access a label defined in the *labels* config file. This method retrieves the label text associated with the label identifier (labelname : String) from the attribute *labelMap*. Afterwards, it calls the *wrapInHtml* method so that the label text is converted into html before it is returned.

- getLabelNoHtml(labelname : String): Offers the same functionality as the method *getLabel*, however it returns the raw label text retrieved from the config file *labels* associated with the given *labelname*.
- getLabels(labelnames : String): Similar to the methods *getLabel* and *getLabelNoHtml*, this method allows the top layer to have access to the label texts defined in the config file *labels*. Different from the aforementioned methods, it combines the label texts for the label identifiers passed as an array of strings and wraps the combined labels with html code. This method is implemented to allow displaying directions for a list of rooms within a single *JLabel* (Figure 4.29).
- refreshRoomStatus(): The purpose of this method is to call the *RoomInfoCollector.updateRemainingTime()* for each object present in the *roomInfoCollectorMap*. As an additional functionality, it also collects the schedule strings from the *RoomInfoCollector* objects and puts them together to form the body of the schedule log table. As the last step, the method *logRoomStatus* is called and the room schedule is logged. This way it is ensured that with a single function call from the top layer, the remaining times of each event within the *RoomInfoCollector* objects are updated together and the changes are logged. The corresponding functionality is demonstrated with a sequence diagram in Figure 4.22.

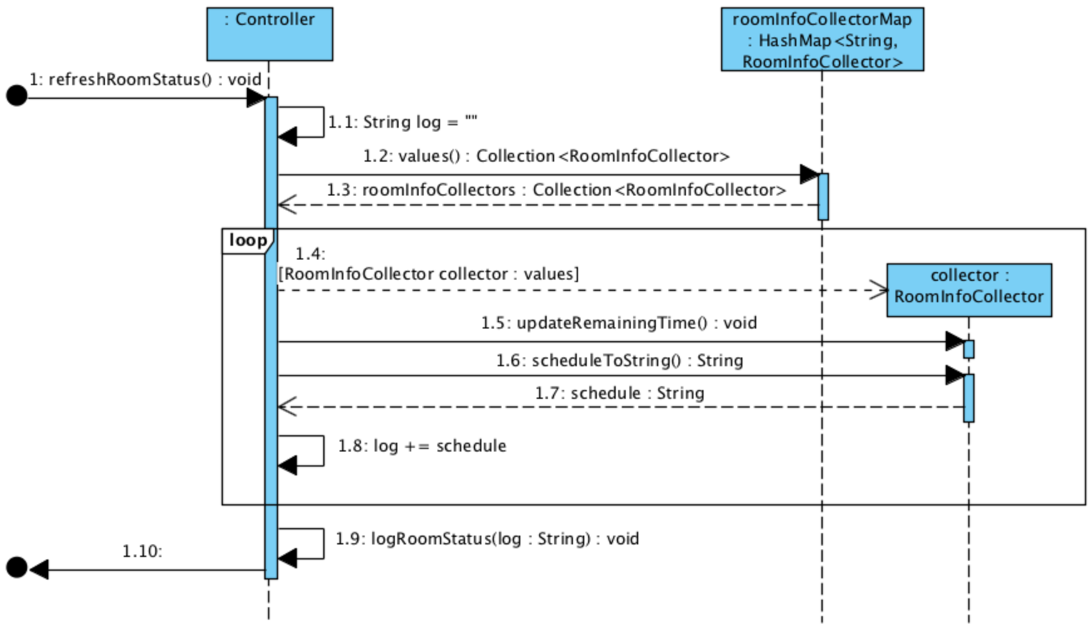

: Controller
roomInfoCollectorMap : HashMap<String, RoomInfoCollector>
1: refreshRoomStatus() : void
1.1: String log = ""
1.2: values() : Collection<RoomInfoCollector>
1.3: roomInfoCollectors : Collection<RoomInfoCollector>
loop
1.4: [RoomInfoCollector collector : values]
collector : RoomInfoCollector
1.5: updateRemainingTime() : void
1.6: scheduleToString() : String
1.7: schedule : String
1.8: log += schedule
1.9: logRoomStatus(log : String) : void
1.10:

Figure 4.22.: Sequence diagram - refreshRoomStatus()

JImage & JComponentWithBackground

These classes are implemented to extend Java classes *JLabel* and *JComponent* respectively for adding background image support. The corresponding class diagrams are demonstrated in Figure 4.23.

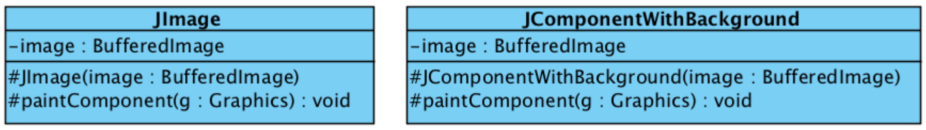

JImage
-image : BufferedImage
#JImage(image : BufferedImage)
#paintComponent(g : Graphics) : void
JComponentWithBackground
-image : BufferedImage
#JComponentWithBackground(image : BufferedImage)
#paintComponent(g : Graphics) : void

Figure 4.23.: Class diagram - JImage & JComponentWithBackground

The aforementioned background image support is implemented by overriding the method *paintComponent* from both of the parent classes *JLabel* and *JComponent*. Its implementation is demonstrated with a sequence diagram in Figure 4.24. Since the overriden methods are exactly the same, to avoid redundancy only the method from the class *JImage* is demonstrated.

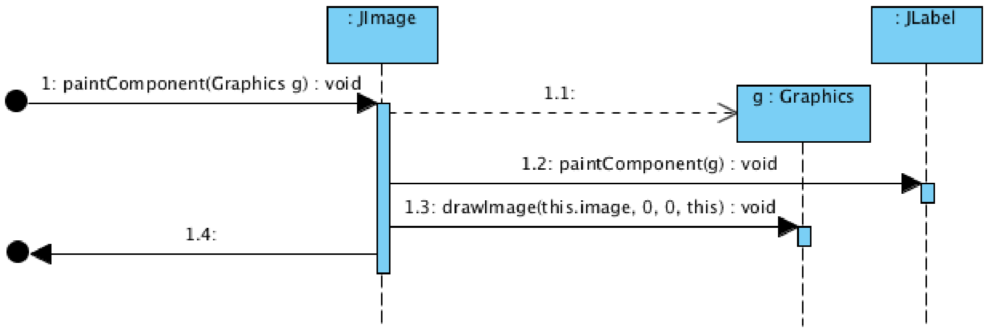
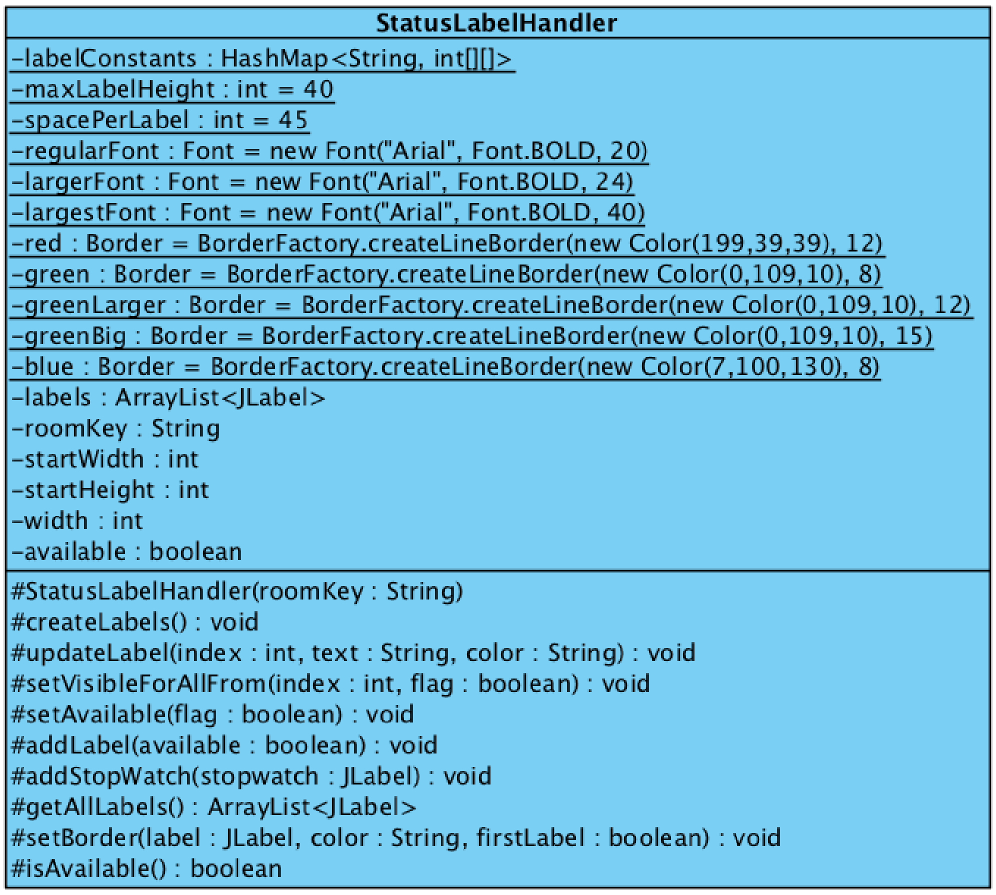

: JImage
: JLabel
1: paintComponent(Graphics g) : void
1.1:
g : Graphics
1.2: paintComponent(g) : void
1.3: drawImage(this.image, 0, 0, this) : void
1.4:

Figure 4.24.: Sequence diagram - paintComponent(Graphics g)

StatusLabelHandler

As previously mentioned in Section 3.4.3, the class *StatusLabelHandler* is responsible for controlling the availability (status) labels within a room area on the GUI. For this reason, each room area requires an instance of this class. The room area that is controlled by an instance of this class is specified by the room identifier passed as an argument to the constructor of this class. As mentioned earlier, the purpose of implementing this class is to minimize the size of the GUI class by packing the handling process of the status labels into a separate class. This way, the dependency on the GUI class is also decreased.

In order to achieve the aforementioned requirements, the class is implemented as demonstrated in Figure 4.25.



```
StatusLabelHandler
-labelConstants : HashMap<String, int[][]>
-maxLabelHeight : int = 40
-spacePerLabel : int = 45
-regularFont : Font = new Font("Arial", Font.BOLD, 20)
-largerFont : Font = new Font("Arial", Font.BOLD, 24)
-largestFont : Font = new Font("Arial", Font.BOLD, 40)
-red : Border = BorderFactory.createLineBorder(new Color(199,39,39), 12)
-green : Border = BorderFactory.createLineBorder(new Color(0,109,10), 8)
-greenLarger : Border = BorderFactory.createLineBorder(new Color(0,109,10), 12)
-greenBig : Border = BorderFactory.createLineBorder(new Color(0,109,10), 15)
-blue : Border = BorderFactory.createLineBorder(new Color(7,100,130), 8)
-labels : ArrayList<JLabel>
-roomKey : String
-startWidth : int
-startHeight : int
-width : int
-available : boolean
#StatusLabelHandler(roomKey : String)
#createLabels() : void
#updateLabel(index : int, text : String, color : String) : void
#setVisibleForAllFrom(index : int, flag : boolean) : void
#setAvailable(flag : boolean) : void
#addLabel(available : boolean) : void
#addStopWatch(stopwatch : JLabel) : void
#getAllLabels() : ArrayList<JLabel>
#setBorder(label : JLabel, color : String, firstLabel : boolean) : void
#isAvailable() : boolean
```

Figure 4.25.: Class diagram - StatusLabelHandler

As can be seen in the class diagram, all the attributes of this class are set to be private. The reason behind setting the access levels this way is that the attributes are intended to be used only within this class. Some of these attributes are also set to be static and final:

- labelConstants: As the attribute name suggests, this *HashMap* contains some constants regarding the size of the labels, which differ from room to room. Room identifier and two dimensional integer array are the key value pairs of this *HashMap*.
- maxLabelHeight, spacePerLabel: These constants define the label height, which is the same for all status labels. The constant height of a label is defined by the attribute *maxLabelHeight* and the space used by a label in total together with its border and the space left for the next border is defined by the attribute *spacePerLabel*.

- regularFont, largerFont, largestFont: As the names suggest, these attributes define the fonts with different sizes used for label texts as demonstrated in Figure 3.10.
- red, greenLarger, greenBig, blue: These attributes define the different borders assigned to the labels. As mentioned earlier, the different sizes of the labels can be seen in Figure 3.10.

The other attributes of the class are as follows:

- labels: This attribute of type *ArrayList<JLabel>* contains all the labels that are displayed within the room area to which an instance of this class is assigned. Labels inside the *ArrayList* are sorted in the same order from the *TreeMap* generated by *RoomInfoCollector* object, which is assigned to the ICS file of the room. In order to make the control mechanism of the labels easier, labels are not removed or resized as mentioned in Section 3.4. In order to achieve the aforementioned requirement, *ArrayList* initialization is implemented in a way that one large label for the top event (index 0) and one big label for the available sign (last index) are added to the *ArrayList* by default. Indexes in between are filled with regular sized labels up to the number that the room area is capable of showing.
- roomKey: A string defining the area on the GUI that the given instance of this class is associated with. Assigned room identifiers are the same as the ones from the *RoomInfoCollector*.
- startWidth, startHeight, width: These attributes define the top left corner coordinates (*startWidht*, *startHeight*) as well as the maximum width allowed for the labels to occupy. These values are retrieved from the static and final attribute of *labelConstants* using the given *roomKey* during the instantiation of the class.
- available: A boolean indicating whether there are events taking place in the room assigned to the instance of this class. This attribute is used by the top layer with the help of its getter and setter.

In order to offer full control over the status labels of a given room area on the GUI, the following methods are implemented:

- addStopWatch(JLabel stopwatch): A method to add a stopwatch image to the top label in the event list shown on the GUI. The label at the index 0 is retrieved from the attribute *labels* of type *ArrayList* first. Afterwards, the label received with the stopwatch image being its background - the argument *stopwatch* of type *JLabel* - is added to the retrieved label.

- addLabel(available : boolean): This method is implemented to instantiate an object of type *JLabel* as a room status indicator and add it to the attribute *labels* of type *ArrayList*. The argument *available* defines whether the label to be added is the largest label, which indicates that the room is available for the whole day. The first added label is a large label for the top event and all the following labels are regular size unless the argument *available* is set to *true*. The corresponding flow is demonstrated with a sequence diagram in Figure 4.26.

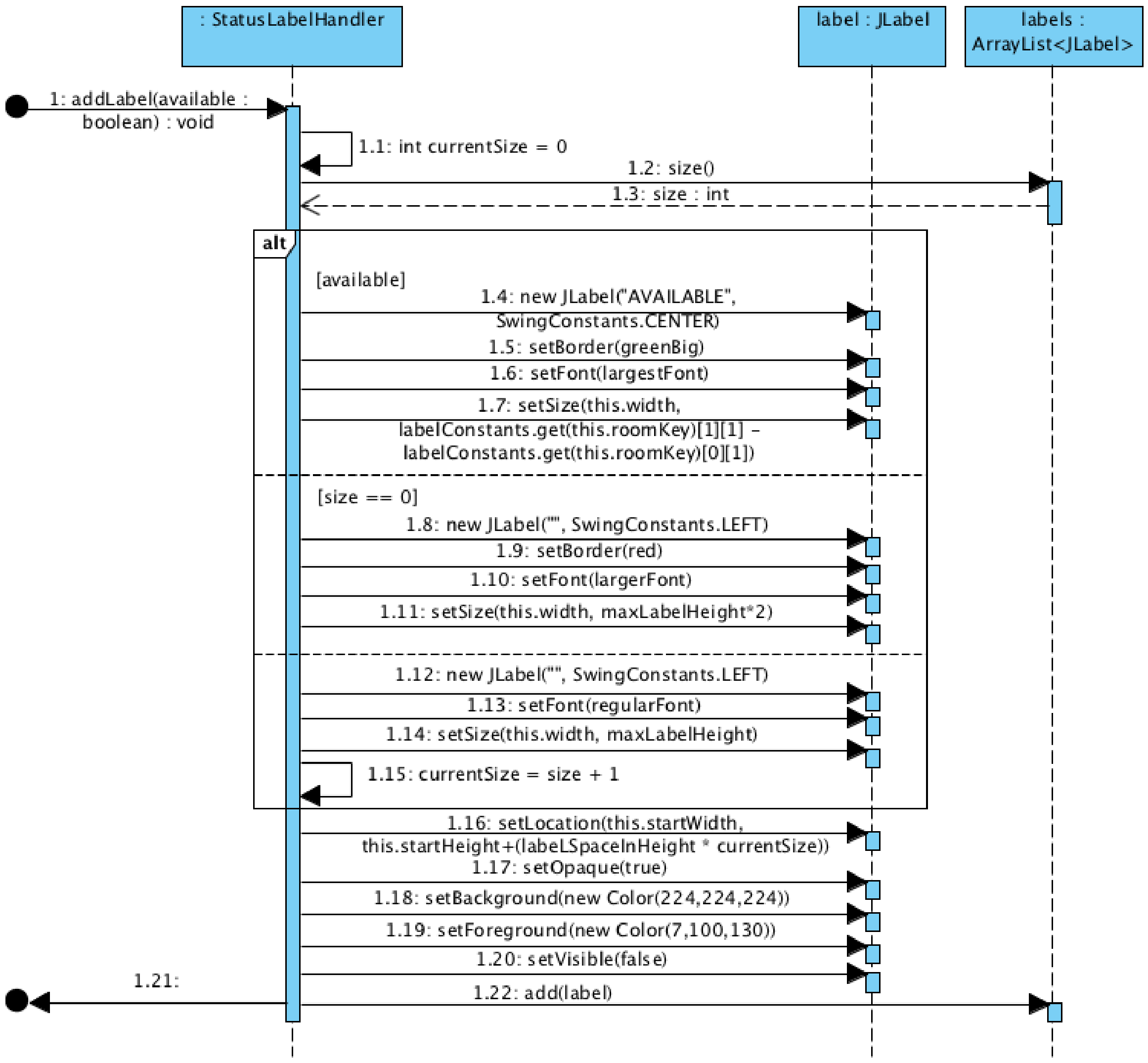

: StatusLabelHandler
label : JLabel
labels : ArrayList<JLabel>
1: addLabel(available : boolean) : void
1.1: int currentSize = 0
1.2: size()
1.3: size : int
alt
[available]
1.4: new JLabel("AVAILABLE", SwingConstants.CENTER)
1.5: setBorder(greenBig)
1.6: setFont(largestFont)
1.7: setSize(this.width, labelConstants.get(this.roomKey)[1][1] - labelConstants.get(this.roomKey)[0][1])
[size == 0]
1.8: new JLabel("", SwingConstants.LEFT)
1.9: setBorder(red)
1.10: setFont(largerFont)
1.11: setSize(this.width, maxLabelHeight*2)
1.12: new JLabel("", SwingConstants.LEFT)
1.13: setFont(regularFont)
1.14: setSize(this.width, maxLabelHeight)
1.15: currentSize = size + 1
1.16: setLocation(this.startWidth, this.startHeight+(labelLSpaceInHeight * currentSize))
1.17: setOpaque(true)
1.18: setBackground(new Color(224,224,224))
1.19: setForeground(new Color(7,100,130))
1.20: setVisible(false)
1.21:
1.22: add(label)

Figure 4.26.: Sequence diagram - addLabel(available : boolean)

- createLabels(): Responsibility of this method is to add the default number of status labels to the attribute *labels* as placeholders for the events. By calling the previously mentioned method *addLabel*, first a larger status label is added to the index 0, followed by a certain number of regular sized status labels and a big green status label with the text "AVAILABLE" as the last element. The reason for this particular implementation is to increase the ease of updating information on the GUI. This implementation allows the status labels to be updated by just assigning of different borders or editing the texts. Additionally, this way it is possible to transfer from the available state to an event list (or vice versa) by just altering the visibility options of the *JLabel* objects.

- setBorder(label : JLabel, color : String, firstLabel : boolean): This method is implemented to be used only within this class and it gives the ability to change the border of a given *JLabel* according to the color string that is passed as an argument. With this method being implemented, the top layer has the ability to indicate a color for a status label by calling the protected method *updateLabel*. The necessity for this method comes from the fact that blue status labels change color to red when they become the top event. Additionally, since the status labels are added to the attribute *labels* of type *ArrayList* as placeholders and altered according to the events, every time a top event finishes and is removed from the list, the border colors need to be set again accordingly. The boolean *firstLabel* is used to assign larger borders to the top events. A *true* being passed indicates that the argument *label* displays the top event and therefore requires a larger border.
- getAllLabels(): Returns a reference to the attribute *labels*. The reason for implementing this method is to give access to the top layer for the status labels so that they can be added to the frame, which is explained in detail later in the description of the class *PcpSI_GUI*.
- isAvailable() - setAvailable(flag : boolean): These methods are implemented as the getter and setter for the attribute *available*. The getter *isAvailable()* is used to tell the top layer whether the instantiated object's event list is empty or not and the setter *setAvailable(flag : boolean)* is used to change the visibility of status labels according to the flag that is passed as an argument.
- setVisibleForAllFrom(index : int, flag : boolean): This is a method that is implemented for achieving the previously mentioned ease of transition of status labels. It makes it possible to change the visibility option of a list of status labels starting from the given index to the last element in the attribute *labels* excluding the big *JLabel* with the text "AVAILABLE". This method is required especially when there are more events registered than the number that can be displayed for a given room. In such situations, all the labels that are not displayed on the GUI can be made invisible with a call to this method.
- updateLabel(index : int, text : String, color : String): This method allows the top layer to update the text together with the border color of a status label. By passing the index

of the status label, the given text and the border color are assigned to the label from the attribute *labels* associated with the index. In order to assign the border color, this method calls the previously mentioned private method *setBorder*. It checks if the given index is less or equal to the maximum number of labels that can be displayed for the room first. If the index is within the limits, then it is checked if the index is 0, which means the status label to be updated belongs to the top event. Since the top event label has a stopwatch at its left hand side, a certain amount of space before the text needs to be added. Once the appropriate spacing is set depending on the given index, the given text is wrapped with *<html><pre> </html></pre>* so that the placed spaces are displayed properly [5]. After the text modification is completed, the text is assigned to the label. This is followed by the call to the previously mentioned method *setBorder* to update the border color of the label according to the color specified by the argument. Since only the visible labels are updated, each label indicated by the index given as the argument is set to visible. The described implementation is demonstrated by a sequence diagram in Figure 4.27.

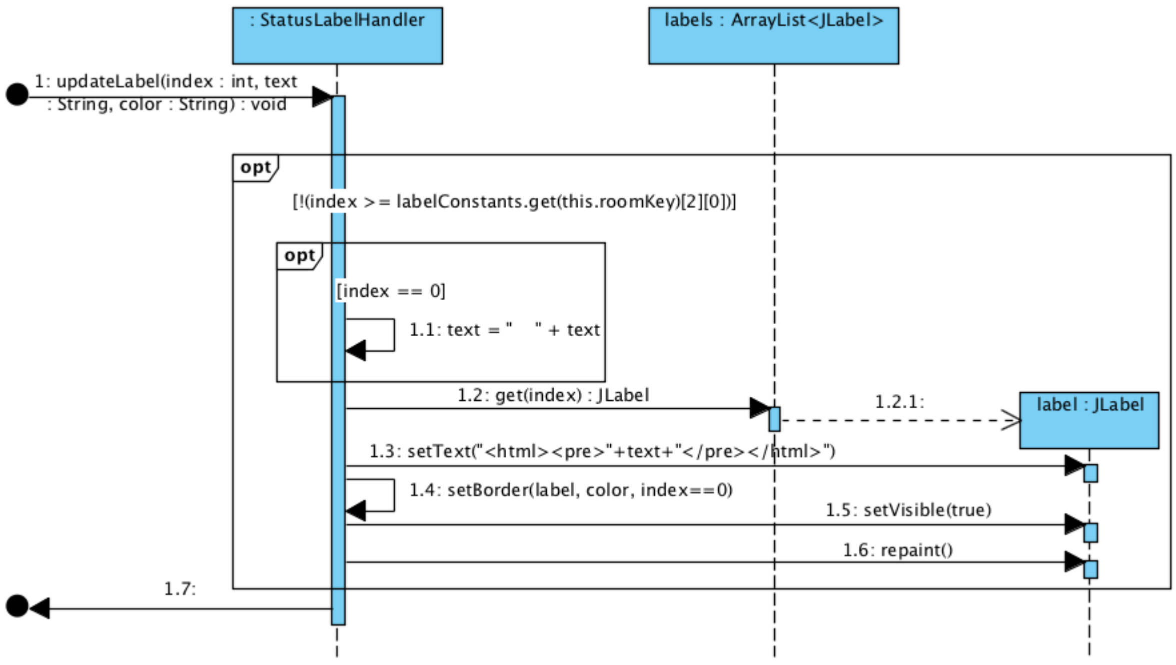

: StatusLabelHandler
labels : ArrayList<JLabel>
1: updateLabel(index : int, text : String, color : String) : void
opt
[!(index >= labelConstants.get(this.roomKey)[2][0])]
opt
[index == 0]
1.1: text = "   " + text
1.2: get(index) : JLabel
1.2.1:
label : JLabel
1.3: setText("<html><pre>"+text+"</pre></html>")
1.4: setBorder(label, color, index==0)
1.5: setVisible(true)
1.6: repaint()
1.7:

Figure 4.27.: Sequence diagram - updateLabel(index : int, text : String, color : String)

PcpSI_GUI

As its name suggests, the class *PcpSI_GUI* is the GUI, in other words the top layer that has been previously mentioned several times. In order to give the ability of retrieving schedule information and applying them to the status label handlers, this class uses an instance of the *Controller* and one instance of the class *StatusLabelHandler* for each room of interest.

One of the challenges of implementing this class is providing the functionality of setting the GUI window to full screen. For this purpose, the Java class *GraphicsEnvironment* is used. This class has a method called *getLocalGraphicsEnvironment*, which returns an object of type Java class *GraphicsEnvironment*. The method *getScreenDevice* of the returned object is supposed to return the display with the index 0 according to the documentation [24]. By calling the method *setFullScreenWindow* with the argument being the frame, the GUI is supposed be set to full screen. However, the described functionality from the documentation is found to be not working. For some reason, the object returned by the method *getScreenDevice* does not point to the right display. For this reason, another method of the class *GraphicsEnvironment* is used, which is called *getScreenDevices*. This method returns the complete list of displays in the form of an array and allow accesses to the desired display by a given index. It is discovered that by referencing the display at the index 0 and calling the method *setFullScreenWindow* on that reference instead, the Java application can be set to full screen successfully.

The corresponding class diagram including all the implemented methods together with the attributes is demonstrated in Figure 4.28.

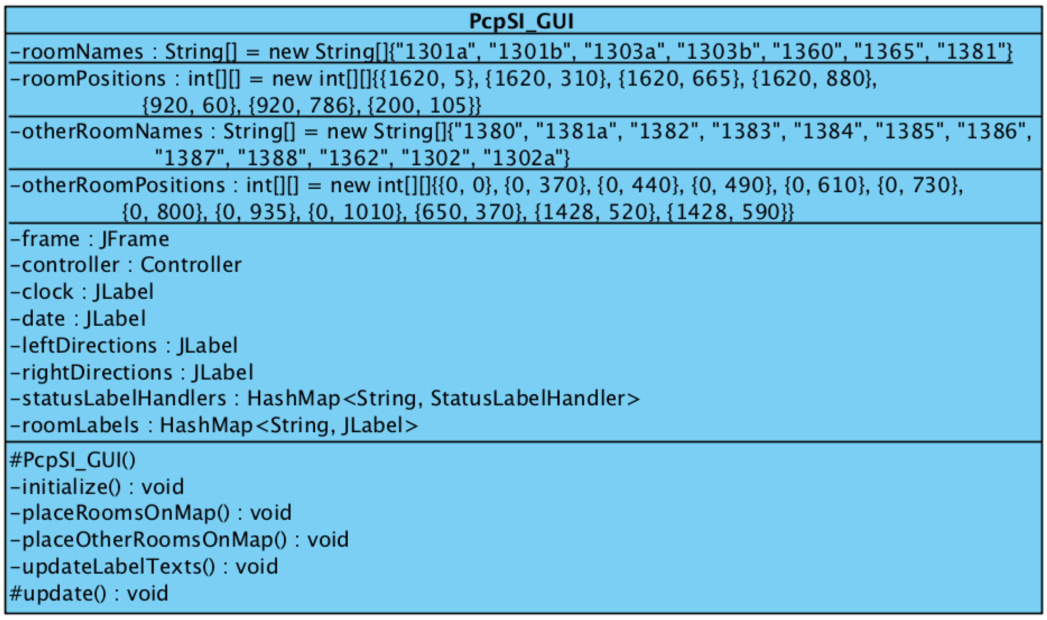

```
PcpSI_GUI
-roomNames : String[] = new String[]{"1301a", "1301b", "1303a", "1303b", "1360", "1365", "1381"}
-roomPositions : int[][] = new int[][]{{1620, 5}, {1620, 310}, {1620, 665}, {1620, 880},
              {920, 60}, {920, 786}, {200, 105}}
-otherRoomNames : String[] = new String[]{"1380", "1381a", "1382", "1383", "1384", "1385", "1386",
               "1387", "1388", "1362", "1302", "1302a"}
-otherRoomPositions : int[][] = new int[][]{{0, 0}, {0, 370}, {0, 440}, {0, 490}, {0, 610}, {0, 730},
             {0, 800}, {0, 935}, {0, 1010}, {650, 370}, {1428, 520}, {1428, 590}}
-frame : JFrame
-controller : Controller
-clock : JLabel
-date : JLabel
-leftDirections : JLabel
-rightDirections : JLabel
-statusLabelHandlers : HashMap<String, StatusLabelHandler>
-roomLabels : HashMap<String, JLabel>
#PcpSI_GUI()
-initialize() : void
-placeRoomsOnMap() : void
-placeOtherRoomsOnMap() : void
-updateLabelTexts() : void
#update() : void
```

Figure 4.28.: Class diagram - PcpSI_GUI

As can be seen in the class diagram demonstrated in Figure 4.28, in order to implement the required functionality, the following attributes are set:

- roomNames, roomPositions, otherRoomNames, otherRoomPositions: These four private, static and final attributes are implemented in order to define room identifiers (*roomNames*, *otherRoomNames*) and to set (x, y) coordinates of the corresponding labels for the identifiers (*roomPositions*, *otherRoomPositions*). Since these identifiers and corresponding positions are intended to be constant and required only once when the GUI is instantiated, they are set to be static and final. The intention of only using these attributes internally within this class is the reason why the access level is set to be private.

- clock, date, leftDirections, rightDirections: These attributes of type *JLabel* are implemented to have references to the displayed time, date and directions for rooms located on the left and right side in respect to the elevators. Since the displayed date and time need to be updated regularly and the displayed directions should adapt to the changes made in the configuration file *labels*, these references are realized in attribute form, instead of being local variables. Displayed time and date, as well as right and left directions are demonstrated in Figure 4.29.

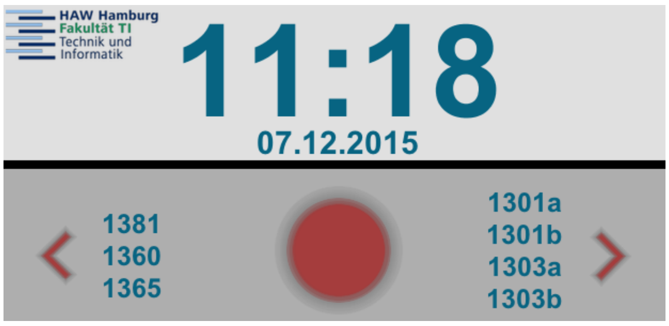


Figure 4.29.: Displayed JLabels clock, date and left, right directions

- controller: As previously mentioned, this class requires to use an instance of the *Controller* class and this attribute is set to assign a reference to the mentioned object of type *Controller*. Since it is intended to use this reference only within this class, access level is set to private.

- statusLabelHandlers: This attribute of type *HashMap<String, StatusLabelHandler>* is set to contain all the references to the objects of type *StatusLabelHandler*, which are instantiated for each room of interest (for each identifier from the attribute *roomNames*). In order to link a schedule information collected from the *controller* with a *statusLabelHandler* object, a *HashMap* is used. This map contains the same keys as the map for the *RoomInfoCollector* objects located in the *controller*. In other words, a *statusLabelHandler* for a given room is assigned the same key as the *RoomInfoCollector* of that room. This way, by using one key to identify a room, this class can access both schedule information and the status labels.

- roomLabels: Another attribute of type *HashMap*, also intended for internal use within this class and set with the access level of private for that reason. A *JLabel* is created for each room identifier and put into this map. This way, with the help of stored references associated with a room identifier, this class can update the room labels according to the configuration file *labels*. As previously mentioned, this is scheduled to take place at the same time as the parsing of the new ICS files, in order to avoid using an additional timer.
- frame: This is the attribute where a reference to the *JFrame* of the GUI is stored [24]. *JFrame* class creates a window in which the GUI is displayed. For the same reasons as the previously mentioned private attributes, this attribute's access level also is set to private.

As demonstrated in the class diagram (Figure 4.28), excluding the constructor there are 5 methods implemented, with 4 of them intended to be used within this class:

- placeRoomsOnMap(): This method is responsible for initializing the labels for the rooms of interest and placing them on the floor plan (map). The position coordinates of the labels as well as the room identifiers are retrieved from the static attributes mentioned earlier *roomPositions* and *roomNames* respectively. Each instantiated label is then added to the attribute *roomLabels* of type *ArrayList* with the corresponding room identifier as the key.
- placeOtherRoomsOnMap(): Similar to the method *placeRoomsOnMap*, this method repeats the exact procedure for all the rooms on the floor except for the rooms of interest. The corresponding room identifiers as well as the label positions are retrieved from the static attributes *otherRoomPositions* and *otherRoomNames* respectively.
- initialize(): As its name suggests, this method is responsible for initializing the GUI. The frame (*frame* attribute) of the GUI is initialized first with its layout set to *null* so that the components can be positioned by defining (x, y) coordinates instead of using a grid layout [24]. Designing a GUI with a grid layout is a complex process, taking a longer implementation time. The only advantage of this layout design is the resizability option it offers for the GUI. However, since the GUI is intended to be used in full screen mode, this advantage is not applicable and therefore in order to avoid complicating the design process a decision is made not to use the grid layout.

As mentioned earlier, in order to add a background image to the GUI, the default *JComponent* class is extended and the *paintComponent* method is overriden. As an alternative, *JFrame* could have been extended with the overriden *paint* method. However the frame is the top level container and therefore its *paint* method actually just calls the corresponding *paint* methods of its components. For this reason, it is a better approach to leave the *JFrame* as it is and introduce an overriden *paint* method (in this case *paintComponent* of *JComponent*) for a component to be added to the frame. In order to do so, the background image (floor plan) is retrieved from the *controller* and passed as an argument to the constructor of the *JComponentWithBackground* class. As the next step the instantiated *JComponentWithBackground* is added to the frame by calling the method *setContentPane* with the *JComponentWithBackground* object passed as an argument. Once the frame with its component is fully established, all the images including emergency exit signs and the first aid sign, left and right arrows, red pointer that indicates where the elevators are, together with the department logo are retrieved from the *controller* object. For each mentioned image a *JImage* object is instantiated and added to the frame with appropriate size and position. The rest of the initialization is completed by calling the previously mentioned methods *placeRoomsOnMap* and *placeOtherRoomsOnMap*.

- updateLabelTexts(): Once called, this method iterates through the labels stored in the attribute *roomLabels*, and by using the room identifier of each label, it retrieves the corresponding texts from the *controller* object. Afterwards, the retrieved texts are set to the labels. The purpose of this method is to introduce the new label texts from the configuration file *labels*.

The only method of this class that has a protected access level and is implemented to be called from outside of this class is the method *update*:

- update(): It is implemented to allow updating the time, date and status labels when needed. Additionally, in order to update the label texts from the configuration file *labels* as well, this method also calls the private method *updateLabeLTexts* when needed. To decide upon the necessity of updating the label texts, the *Controller.isUpdateTime()* is used. After this step the current time and date are retrieved from the *controller* and set to the respective labels *clock* and *date*. Once the date and time on the GUI are updated, it iterates through the *roomNames*. For each room identifier, the corresponding references of the *roomSchedule* (Controller.getRoomScheduleMap(<room identifier>)) and the *StatusLabelHandler* (PcpSI_GUI.StatusLabelHandlerMap.get(<room identifier>)) are retrieved. If the *roomSchedule* map is empty, the method sets the corresponding *StatusLabelHandler* to active. This way, the big green label with the text "AVAILABLE" is displayed for the corresponding room. Otherwise, by iterating through the events within the *roomSchedule*, certain tasks are done. In order to explain these tasks in detail, it is necessary to first demonstrate how the label texts are formulated for the following possible event options:

– Available: If an event is set to be available, there are two possible situations: either the event has started or it is going to start at a later time, which is indicated by the first element from the event information array (strings "tofinish" - "tostart" respectively (Section 4.2.1 collectRoomInfo() method description)). If the event is set to be available and has started already, it means that it is the top event and the label text is set to be *<remaining time> <event summary>* (4th and 1st index from the event information array respectively). If the event has not started yet, that also means there are two possible situations: either the event has an end time specified, which means it indicates an available time period between two events, or it has no end time specified meaning that it is the last event of the day. For the first option, the label text is formulated to be *<event start time> - <event end time> <summary>* (2nd, 3rd and 1st index from the event information array respectively). The other option requires a different formulation since there is no end time specified: *<event start time> <summary>*. For all the events containing the text *available* in their summary, the border color is set to green.

– Lecture or Lab: If the summary of an event does not contain the word "available", similarly, there are two possible situations: either the event has started or it is going to start at a later time. The first option means that the room is occupied and therefore the border color needs to be set to red. Additionally it means that the event is the top event and the label text needs to be set as it is for *available*: *<remaining time> <event summary>*. For the other option, the border color is set to blue and the label text is formulated as follows: *<event start time> - <event end time> <summary>*

In order to set the label texts and the corresponding borders to the respective status labels, the method *StatusLabelHandler.updateLabel* (Section 4.2.1 corresponding method description) is used with the arguments being the formulated label text and a string indicating the color of the border. The corresponding implementation of the method *update* is demonstrated with a sequence diagram in Figure 4.30.

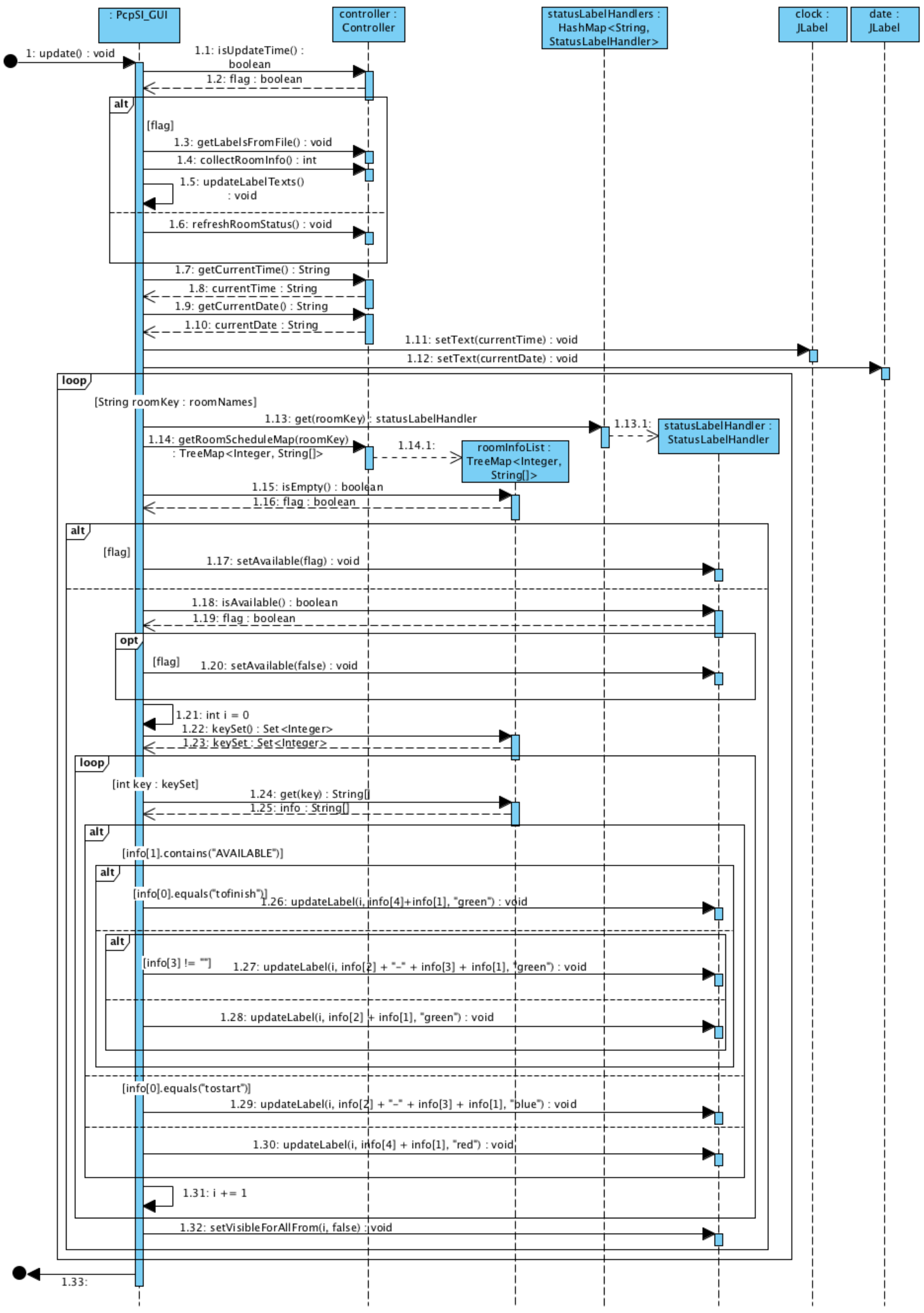

: PcpSI_GUI
controller : Controller
statusLabelHandlers : HashMap<String, StatusLabelHandler>
clock : JLabel
date : JLabel
1: update() : void
1.1: isUpdateTime() : boolean
1.2: flag : boolean
alt
[flag]
1.3: getLabelsFromFile() : void
1.4: collectRoomInfo() : int
1.5: updateLabelTexts() : void
1.6: refreshRoomStatus() : void
1.7: getCurrentTime() : String
1.8: currentTime : String
1.9: getCurrentDate() : String
1.10: currentDate : String
1.11: setText(currentTime) : void
1.12: setText(currentDate) : void
loop
[String roomKey : roomNames]
1.13: get(roomKey) : statusLabelHandler
1.13.1:
statusLabelHandler : StatusLabelHandler
1.14: getRoomScheduleMap(roomKey) : TreeMap<Integer, String[]>
1.14.1:
roomInfoList : TreeMap<Integer, String[]>
1.15: isEmpty() : boolean
1.16: flag : boolean
alt
[flag]
1.17: setAvailable(flag) : void
1.18: isAvailable() : boolean
1.19: flag : boolean
opt
[flag]
1.20: setAvailable(false) : void
1.21: int i = 0
1.22: keySet() : Set<Integer>
1.23: keySet : Set<Integer>
loop
[int key : keySet]
1.24: get(key) : String[]
1.25: info : String[]
alt
[info[1].contains("AVAILABLE")]
alt
[info[0].equals("tofinish")]
1.26: updateLabel(i, info[4]+info[1], "green") : void
alt
[info[3] != ""]
1.27: updateLabel(i, info[2] + "-" + info[3] + info[1], "green") : void
1.28: updateLabel(i, info[2] + info[1], "green") : void
[info[0].equals("tostart")]
1.29: updateLabel(i, info[2] + "-" + info[3] + info[1], "blue") : void
1.30: updateLabel(i, info[4] + info[1], "red") : void
1.31: i += 1
1.32: setVisibleForAllFrom(i, false) : void
1.33:

Figure 4.30.: Sequence diagram - update()

UpdateGuiTimerTask

As previously mentioned in Section 3.3.1, the purpose of this class is to provide 1 minute periodic execution of the method *PcpSI_GUI.update()*. In order to fulfill this purpose, it extends the Java class *TimerTask* by overriding the method *run*. A private attribute *gui* is used to store a reference to the instantiated object of type *PcpSI_GUI*. The instantiation of this object and the assignment of the reference to the mentioned attribute take place inside the constructor of the class. The method *run* just calls *gui.update()*. The corresponding implementation is demonstrated with a class diagram in Figure 4.31.

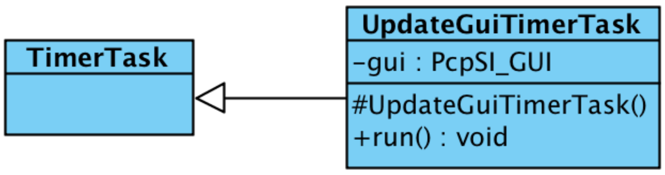


Figure 4.31.: Class diagram - UpdateGuiTimerTask

Main

This class is implemented to contain the *main* method of the Java application. It is responsible for instantiating the *UpdateGuiTimerTask* and scheduling it to be executed every minute. For this purpose, first a Java *Timer* object is instantiated and its method *scheduleAtFixedRate* is used [24]. This method expects 3 arguments:

- Reference to a *TimerTask* object: a reference to the instantiated object of type *UpdateGuiTimerTask* is passed to ensure the implemented *UpdateGuiTimerTask.run* method is executed periodically.
- Delay in milliseconds: used to delay the first execution of the *TimerTask* by the given amount of milliseconds. The idea here is to sync the execution of the *TimerTask* with the system clock so that the minute and the remaining times of events displayed on the GUI match the system clock, which is automatically obtained from the wireless network by the Raspberry Pi. For this purpose, this delay is set to be the milliseconds remaining to a whole minute at the execution time. This is achieved by calling the *System.currentTimeMillis()* method first, which returns the current time in milliseconds. Dividing this value by 1000 and casting to integer converts the current time to seconds. The obtained amount of seconds equals to the total number of seconds in the current time (*hour*:*minute*:*second*); in other words: *hour**24*60 + *minute**60 + *second*. Since the first two terms in the demonstrated addition are both multiplied by 60, by taking the modulus 60 of the total seconds, the *second* in the current time is obtained in integer

form. After this step, the delay is set by subtracting the obtained value from 60 and multiplying it by 1000 (conversion back to milliseconds), to ensure the *TimerTask* is executed exactly when the system clock *minute* changes.

- Period in milliseconds: As previously mentioned, the desired period of execution is one minute, hence this argument is set to *1*60*1000*.

Described flow of the *main* method is demonstrated by a sequence diagram in Figure 4.32.

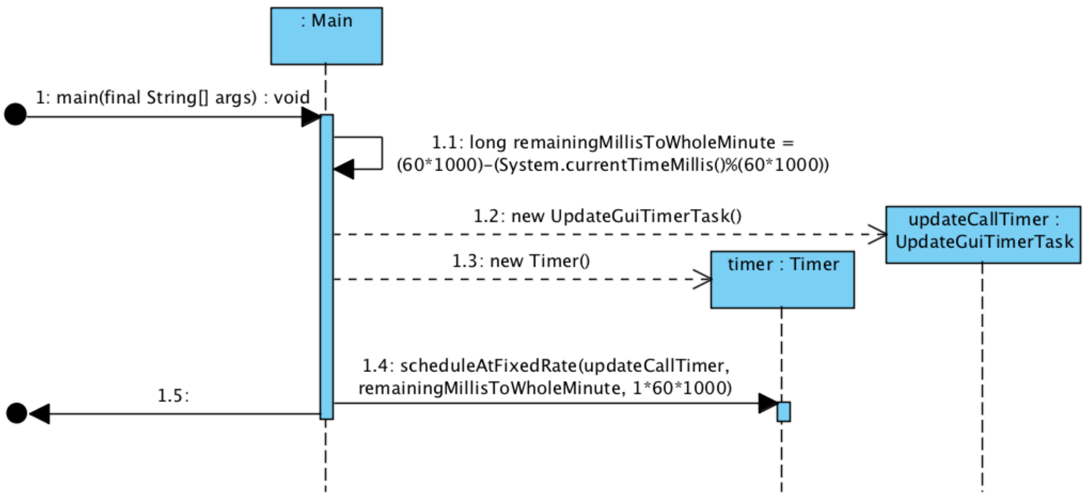

: Main
1: main(final String[] args) : void
1.1: long remainingMillisToWholeMinute = (60*1000)-(System.currentTimeMillis()%(60*1000))
1.2: new UpdateGuiTimerTask()
updateCallTimer : UpdateGuiTimerTask
1.3: new Timer()
timer : Timer
1.4: scheduleAtFixedRate(updateCallTimer, remainingMillisToWholeMinute, 1*60*1000)
1.5:

Figure 4.32.: Sequence diagram - main

4.2.2. Test Cases

In order to test the functionality of the Java GUI fully, testing the following scenarios is sufficient:

- Displaying events from the ICS files at start up
- Refreshing remaining times of the delayed events
- Updating the event lists on a new day
- Room status transitions
  - Available to occupied
  - Occupied to available
  - After the last event on the list finishes transition to available

For the sake of simplicity and to demonstrate readable texts in the images, screen shots of the GUI focus on the area with room 1360 and the time when the screen shot is taken.

Start up

This test case focuses on whether the events are displayed correctly at the start up of the Java GUI. For this purpose, the date 17.11.2015 is selected and the event schedule of room 1360 is taken into consideration. The events scheduled for 17.11.2015 from the ICS file of room 1360 are demonstrated in Listing 4.6.

```
BEGIN:VEVENT
SUMMARY:BMT3-MSP1/01
DTSTART;TZID=Europe/Berlin:20151117T081000
DTEND;TZID=Europe/Berlin:20151117T112500
END:VEVENT
--------------
BEGIN:VEVENT
SUMMARY:E1b-PRP1/02
DTSTART;TZID=Europe/Berlin:20151117T121000
DTEND;TZID=Europe/Berlin:20151117T154000
END:VEVENT
--------------
BEGIN:VEVENT
SUMMARY:IE3-EL2 tutorial
DTSTART;TZID=Europe/Berlin:20151117T160000
DTEND;TZID=Europe/Berlin:20151117T173000
END:VEVENT
```

Listing 4.6: ICS file contents of the room 1360 for 17.11.2015

As can be seen in Listing 4.6 there are 3 events scheduled: *BMT3-MSP1/01* from 08:10 to 11:25, *E1b-PRP1/02* from 12:10 to 15:40, and *IE3-EL2 tutorial* from 16:00 to 17:30. Starting the Java GUI with the system date set to 17.11.2015, it is seen that the program is able to display the event list correctly at start up, including the right remaining times together with the populated available events where applicable (Figure 4.33).

As demonstrated in the screen shot, not all the events from the ICS file are displayed. The reason here is the limit set for this particular room on the number of events it can display at a given time. As can be seen, a 5th row cannot fit into the area drawn for the room and therefore it is set to be invisible, however it is indeed in the list of events, which is demonstrated in Figure Figure 4.36.

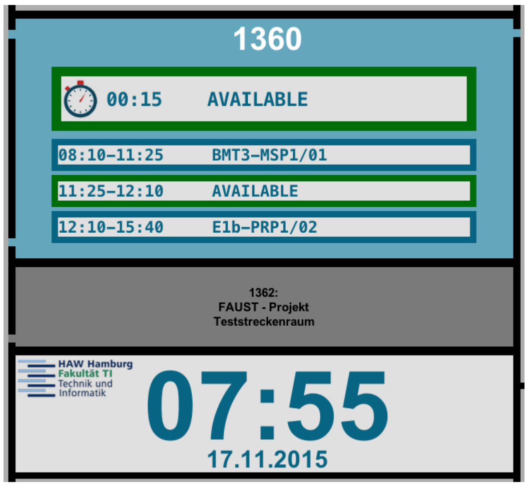


Figure 4.33.: Event list of room 1360 for 17.11.2015

Remaining time updates

In order to test the remaining time updates, the same date, hence the same events from the Listing *startupeventlist* are used. After starting the Java GUI and waiting for about a minute, the expected change in the remaining time of the top event is shown on the GUI, which is demonstrated in Figure 4.34.

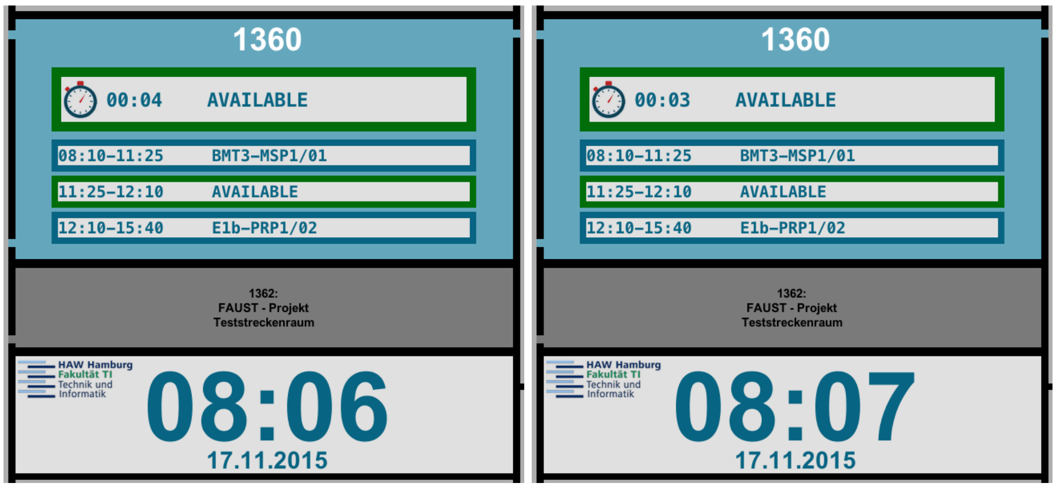
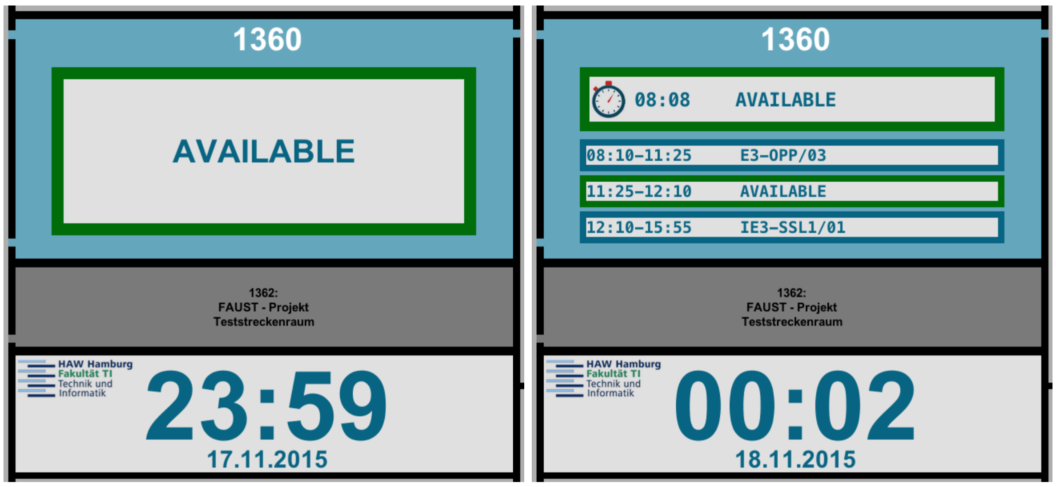


Figure 4.34.: Remaining time change of the top event

As can be seen, the remaining time is correctly updated with respect to the current time displayed.

Day transitions and new event lists

The purpose of this test case is to check if the event lists are updated at the end of the day according to the new day. In order to complete this test, first the system time is set to 23:59 and after waiting for 3 minutes (updates take place at 00:02), the new event lists are checked. The corresponding screen shots are demonstrated in Figure 4.35.



1360

AVAILABLE

1362:
FAUST - Projekt
Teststreckenraum

HAW Hamburg
Fakultät TI
Technik und
Informatik

23:59
17.11.2015

1360

08:08 AVAILABLE

08:10-11:25 E3-OPP/03
11:25-12:10 AVAILABLE
12:10-15:55 IE3-SSL1/01

1362:
FAUST - Projekt
Teststreckenraum

HAW Hamburg
Fakultät TI
Technik und
Informatik

00:02
18.11.2015

Figure 4.35.: Event list transition at the end of a day

As can be seen in Figure 4.35, after midnight, exactly at 00:02, the event list gets updated according to the new day.

Top event transitions

As mentioned earlier there a are couple of cases to be covered within the top event transitions. The transition of the top event from available to occupied is considered first. It is expected that the top event is replaced by the event below and all other events move up by one. As described earlier, this is not a move per se, it is just a reassignment of label texts to the stationary labels. Additionally, the border color of the top event should change from green to red. The corresponding screen shot is demonstrated in Figure 4.36.

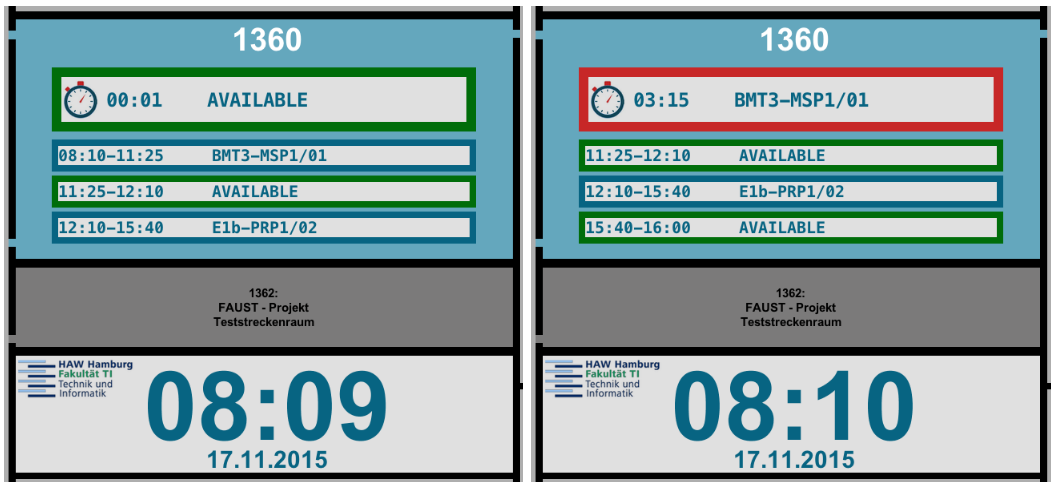


Figure 4.36.: Top event transition - available to occupied

As can be seen, transitions from available to occupied for the top events function as expected. A similar behavior is expected when the top event changes from occupied to available. Events should move up by one as described before. However, for this case the border color should change from red to green. Screen shot of the corresponding change is demonstrated in Figure 4.37 and this behavior also seems to function as expected.

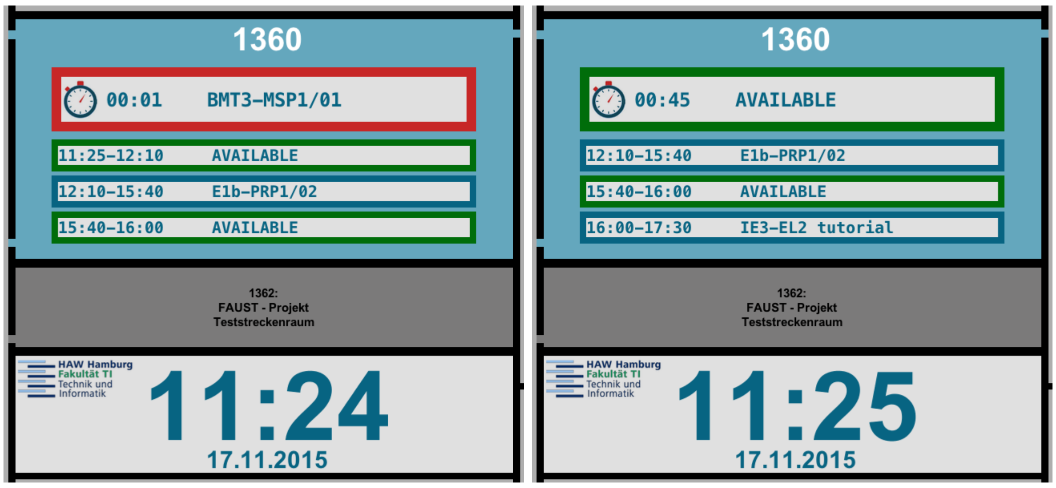


Figure 4.37.: Top event transition - occupied to available

Last transition that is tested occurs when the last event in the event list finishes. At this point, the big available sign is expected to be displayed. This is done by changing the visibility property of the labels. All the status labels are set to invisible and the large available sign to visible. The functionality of the corresponding transition is demonstrated in Figure 4.38.

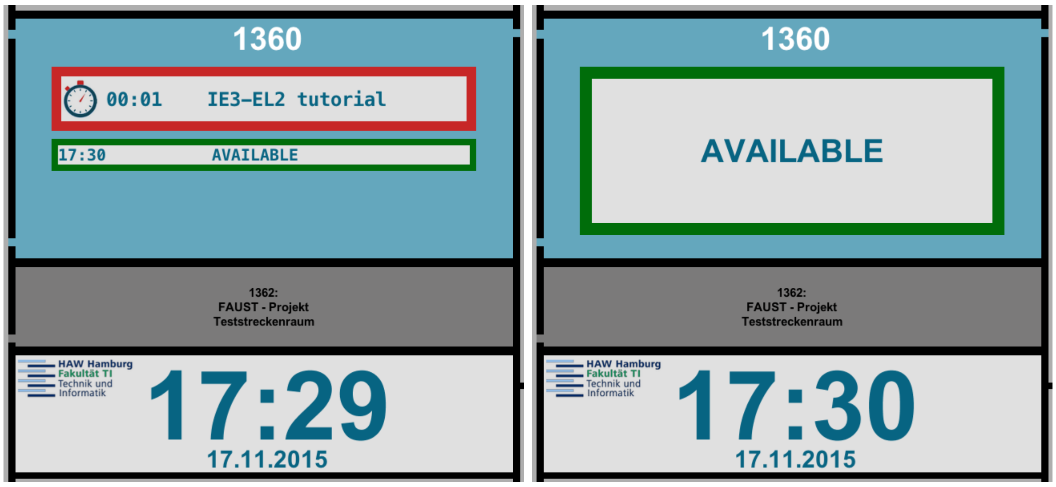


Figure 4.38.: Last event finishes - transition to available

Execution time

In order to test the final system's computation capability as well as the computation weight of the application in more advanced systems, the Java GUI is executed on 4 different systems with the following processors:

- ARM1176JZF-S single core 700Mhz (Raspberry Pi model B revision 1)
- ARM Cortex-A7 quad core 900Mhz (Raspberry Pi model B revision 2)
- Pentium 4 single core 2.86Ghz
- I7 dual core 1.76Ghz

On all the systems, the method *PcpSI_GUI.update()* is tested. As previously mentioned in the method's description in Section 4.2.1, it can either ask for an update on the schedule lists, meaning the ICS files are accessed, or it can just retrieve the refreshed schedule lists from the earlier parsed ICS files. Both scenarios are executed 100 times on all the devices and the mean value of the execution time is calculated. The corresponding results are demonstrated in Table 4.4.

	no file access [ms]	file access [ms]
ARM1176JZF-S single core 700Mhz	37.3	2342
ARM Cortex-A7 quad core 900Mhz	10.4	662.3
Pentium 4 single core 2.86Ghz	2.4	276
i7 dual core 1.76Ghz	0.82	13.7

Table 4.4.: Execution time of *PcpSI_GUI.update()* on different systems

The demonstrated results indicate that the method *PcpSI_GUI.update()*, which is executed periodically every minute, is going to be keeping the CPU busy for about 10ms 1439 times a day (24 * 60 - 1) and about 662.3ms once a day. This shows that the system is going to be idle most of the time meaning very little power consumption. Comparing values from the table to each other also proves that the Raspberry Pi model B revision 2 is capable of running the developed GUI application *PcpSI* with a good enough performance.

4.3. Bash Script PcpSI

In order to ensure that the system has the ability to start the Java GUI application natively, the compilation and execution process of the Java source code is automated by a bash script called *PcpSI*. This script is created and placed in the project directory as well as under the directory */usr/bin/* so that it is accessible as a native executable within the system. This way, regardless of the directory where a shell is started, by executing the command *PcpSI*, the Java GUI can be started. This is useful especially if the application is desired to be started via ssh.

An optional argument to the bash script is defined to be *-c*. If the script is called with this argument, then it recompiles the Java code and executes, otherwise the earlier compiled version is executed. The idea here is to allow a faster execution of the application if no recompilation is required.

The first thing done by the script is navigating to the directory *classes* where the Java source code is located. Then it checks the existence of the previously mentioned argument -*c*. If the argument is present, it compiles the Java code by calling the Java compiler *javac* with the argument *Main.java*. Once the compilation is completed, the created Java executable is executed by the command *DISPLAY=:0 java Main*. In order to make sure the GUI always starts on the native display of the system, the display environment variable is set to 0. The purpose here is to make sure the application can also be started via the ssh.

4.4. Final GUI View

The final result of the GUI is demonstrated in Figure 4.39.

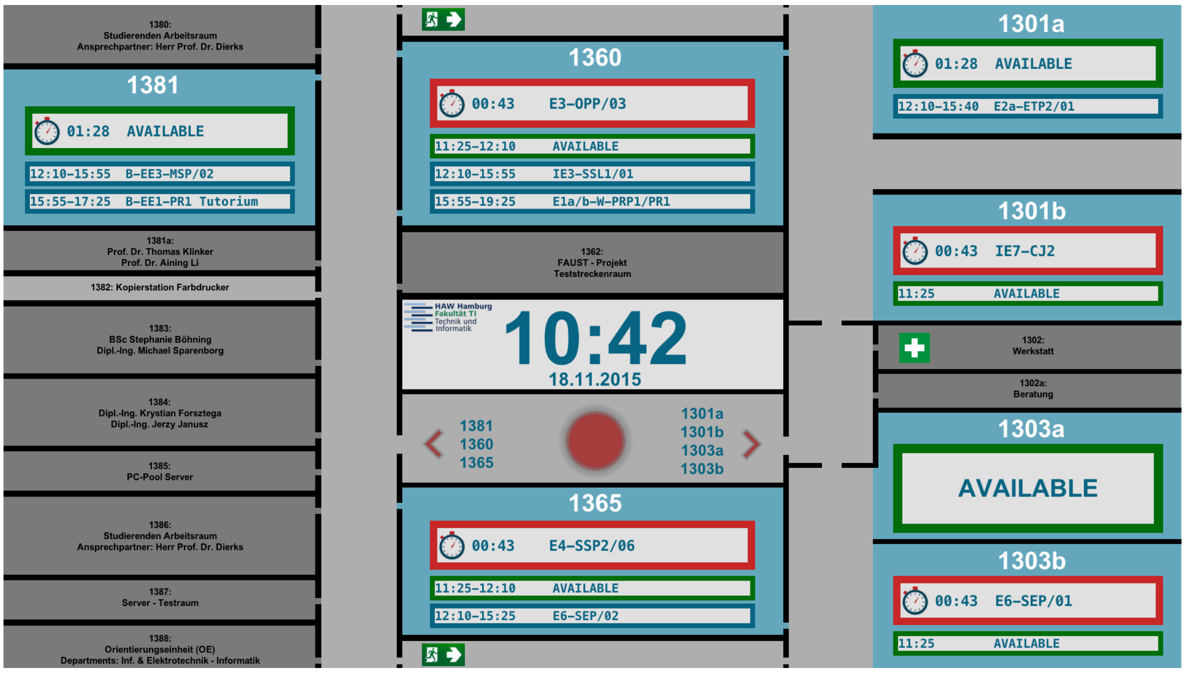


Figure 4.39.: Final view of the GUI

5. Conclusion

As demonstrated throughout this thesis, a self-maintained system with a resource aware concept is developed successfully on a low cost system using a Raspberry Pi. The system offers very low power consumption together with more than enough computation power. Additionally, with the analysis that is made, including the design and implementation process, the Raspberry Pi model B revision 2 is proven to be the best fit for the required task.

In order to make the most of the Raspberry Pi, all the useful features offered by its operating system are used:

- Preinstalled Python and Java compilers
- Cron command scheduling
- Ssh server
- Command-line based wireless network utility tool wpa_supplicant

Considering the initial requirements mentioned in Section 1.1, the automated process of establishing Internet access is realized with the help of a Python script using the preinstalled tool wpa_supplicant, which is periodically executed by cron command scheduling. Acquiring internet access this way also allows the system to download ICS files for the rooms of interest from a remote location, which is again executed with the help of a Python script registered as a cron job to be executed periodically. With the help of the preinstalled tool wpa_supplicant together with the cron command scheduling, the implemented Python scripts finalize the back-end of the system.

Additional to the system's back-end, a front-end is developed to fulfill the requirements of displaying the schedule information of the rooms of interest on a display in a user friendly manner. In order to do so a Java GUI application is implemented with the capability of parsing the ICS files downloaded by the back-end. Being able to retrieve the necessary information from the ICS files, Java GUI offers the full functionality of displaying the retrieved schedules in a user friendly fashion as required.

In order to minimize the necessity of maintenance even further, a wrapper in the form of a Python script is implemented to restart the Java application in case it crashes. With the help of this wrapper being scheduled to start at the start up of the Raspbian OS, the system is allowed to work without any necessity for any form of a user input starting from the point it is powered on and connected to a display in the range of the "HAW.1X" wireless network.

The aim of having a resource aware concept with minimal power consumption and cost, together with self-maintenance capabilities is successfully achieved. However, having all the requirements fulfilled does not mean there is no room for improvement or further development.

5.1. Possible Improvements & Further Development

As previously demonstrated in Table 4.4, the parsing of the ICS files take a fair amount of time considering how little the system requires to assess and display the collected information. With the current implementation, regardless of a given ICS file offering new information or not, it is parsed every day at 00:02. However, this can be changed with a small addition to the application when desired. Each event block within the ICS file contains a keyword *Stand* followed by a date, which indicates the version of the ICS file. If there are changes made to a given file, the registered date (Stand) is updated accordingly. With this information in hand, a further development could be made that checks the revision date of the ICS file before parsing it. By adding an additional attribute to the *RoomInfoCollector* class where the last parsed revision date is stored, the class can tell whether the ICS file issued to be parsed has any new information compared to the one parsed earlier. If the revision date is different, then it can parse the new ICS file and overwrite the attribute containing the revision date with the date from the new file.

The aforementioned functionality is not implemented due to time constraints. However, since the system is idle most of the time and considering it offers enough computation power, this improvement is not a necessity. Worst case scenario, the method *PcpSI_GUI.update()* takes about 660-670ms while parsing the files and goes back into the idle state waiting for the next timer interrupt. The timer interrupt being 1 minute periodic allows the system to stay in the idle state long enough and significantly decreases the importance of this implementation. A possible reason for this improvement can be, however, to decrease the required computation power even further for running the application on a less capable system if required at a later point.

5.1.1. API Option

A more significant improvement for the system would be to allow remote system access to the information displayed on its GUI. At the moment, the system generates a log file located in *logs* directory with the name *room_status.log*, which is updated every minute with the schedule information displayed on the GUI (Listing 5.1).

```
room    summary              start   end     remaining
1303a   -
1303b   E6-SEP/01            08:10   11:25   00:42
        AVAILABLE            11:25           00:42

1360    E3-OPP/03            08:10   11:25   00:42
        AVAILABLE            11:25   12:10   00:42
        IE3-SSL1/01          12:10   15:55   01:27
        E1a/b-W-PRP1/PR1     15:55   19:25   05:12
        AVAILABLE            19:25           08:42

1301a   AVAILABLE                    12:10   01:27
        E2a-ETP2/01          12:10   15:40   01:27
        AVAILABLE            15:40           04:57

1301b   IE7-CJ2 08:10        11:25   00:42
        AVAILABLE            11:25           00:42

1365    E4-SSP2/06           08:10   11:25   00:42
        AVAILABLE            11:25   12:10   00:42
        E6-SEP/02            12:10   15:25   01:27
        AVAILABLE            15:25           04:42

1381    AVAILABLE                    12:10   01:27
        B-EE3-MSP/02         12:10   15:55   01:27
        B-EE1-PR1 Tutorium   15:55   17:25   05:12
        AVAILABLE            17:25           06:42
```

Listing 5.1: Content example to the log file *room_status.log*

Since there is ssh access allowed to the system, this log file is accessible throughout the local area network. With this in mind, the schedule information generated by this system can be displayed anywhere within the range of the HAW.1X network. In other words, the system can be used as an API providing the schedule information of the PC Pool rooms in text format, which can be parsed to be displayed in a different form elsewhere. An example would be a display at the entrance of the building showing the floor plan including room schedules. Sub systems like the PC Pool Status Indicator (PcpSI) can be developed for each floor displaying the schedule information on a display in front of the elevators and set to forward the displayed information to a parent system located at the entrance of the building. The parent system could have a touch screen interface displaying each floor on a different page and could allow navigation between pages via user input.

5.1.2. Webpage

Additionally, in a similar fashion to the mentioned log file, the system can be further developed to generate an html file to be displayed on a web page. Theoretically, the web page can also be run on the system by a web server. However, the accesses to the server should be kept at a minimum level due to the limited computation power offered by the system. This way, the schedule information can be made available to other systems within the HAW.1X network.

References

[1] ADAFRUIT: *Power Usage*. https://learn.adafruit.com/embedded-linux-board-comparison/power-usage Accessed: 02/12/2015,

[2] ADAFRUIT: *Raspberry Pi 2, Model B*. https://www.adafruit.com/pdfs/raspberrypi2modelb.pdf Accessed: 28/11/2015,

[3] BENCHOFF, Brian: *Hands On With The Intel Edison*. http://hackaday.com/2014/09/10/hands-on-with-the-intel-edison/ Accessed: 02/12/2015,

[4] DEBIAN WIKI: *Fonts*. https://wiki.debian.org/Fonts Accessed: 05/12/2015,

[5] DUCKETT, Jon: *HTML and CSS: Design and Build Websites*. John Wiley & Sons, 2011. – ISBN 9781118206911

[6] ELINUX.ORG: *CI20 Hardware*. http://elinux.org/CI20_Hardware Accessed: 02/12/2015,

[7] ELINUX.ORG: *RPi Easy SD Card Setup*. http://elinux.org/RPi_Easy_SD_Card_Setup Accessed: 01/12/2015,

[8] ELINUX.ORG: *RPi raspi-config*. http://elinux.org/RPi_raspi-config Accessed: 03/12/2015,

[9] ELINUX.ORG: *RPi USB Wi-Fi Adapter*. http://elinux.org/RPi_USB_Wi-Fi_Adapters Accessed: 18/01/2016,

[10] ELINUX.ORG: *RPiconfig*. http://elinux.org/RPiconfig Accessed: 21/01/2016,

[11] GENTOO WIKI: *Sysbench*. https://wiki.gentoo.org/wiki/Sysbench Accessed: 23/01/2016,

[12] GILLMOR, Daniel K. ; SISSEL, Jordan: *Ubuntu Manuals - xdotool*. http://manpages.ubuntu.com/manpages/raring/man1/xdotool.1.html Accessed: 05/12/2015,

[13] HAMBLEY, Allan R.: *Electrical Engineering Principles and Applications*. 4. Pearson Education, Inc, 2007. – ISBN 9780132066921

[14] HAMINGTON, Suzie: *The Illustrated Dictionary of Computer Science*. Lotus Press, 2004. – ISBN 9788189093242

[15] HARRINGTON, William: *Learning Raspbian*. Packt Publishing Ltd, 2015. – ISBN 9781784390181

[16] HUNT, David: *Raspberry Pi 2 Benchmarked*. http://www.davidhunt.ie/raspberry-pi-2-benchmarked/ Accessed: 10/01/2016,

[17] KIRKLAND, James ; CARMICHAEL, David ; TINKER, Christopher L. ; TINKER, Gregory L.: *Linux Troubleshooting for System Administrators and Power Users*. Prentice Hall Professional, 2006. – ISBN 9780132797399

[18] LB-LINK: *LB-LINK BL-LW05-5R1 Product Description*. http://p.globalsources.com/IMAGES/PDT/SPEC/146/K1072982146.pdf Accessed: 30/11/2015,

[19] MALINEN, Jouni: *Linux WPA/WPA2/IEEE 802.1X Supplicant*. http://w1.fi/wpa_supplicant/ Accessed: 15/12/2015,

[20] MARTELLI, Alex: *Python in a Nutshell*. 2. O'Reilly Media, Inc., 2006. – ISBN 9781449379100

[21] MONK, Simon: *Raspberry Pi Cookbook*. O'Reilly Media, Inc, 2013. – ISBN 9781449365295

[22] OBJECT MANAGEMENT GROUP: *Unified Modeling Language (UML) Resource Page*. http://www.uml.org Accessed: 26/01/2016,

[23] OPENSSL SOFTWARE FOUNDATION: *OpenSSL SHA*. https://www.openssl.org/docs/manmaster/crypto/sha.html Accessed: 01/12/2015,

[24] ORACLE: *Java 8 Oracle Documentation*. https://docs.oracle.com/javase/8/docs/api/index.html Accessed: 09/12/2015,

[25] ORACLE: *Java Platform, Standard Edition 8 Names and Versions*. http://www.oracle.com/technetwork/java/javase/jdk8-naming-2157130.html Accessed: 10/12/2015,

[26] PETRELEY, Nicholas ; BACON, Jono: *Linux Desktop Hacks*. O'Reilly Media, Inc., 2005. – ISBN 9783897214200

[27] PROCACCIANTI, Giuseppe ; VETRO, Antonio ; ARDITO, Luca ; MORISIO, Maurizio: Profiling Power Consumption on Desktop Computer Systems. In: *ICT-GLOW'11 Proceedings of the First international conference on Information and communication on technology for the fight against global warming* (2011), S. 110–123

[28] RASPBERRY PI FOUNDATION: *FAQ - Power*. https://www.raspberrypi.org/help/faqs/#power Accessed: 09/01/2016,

[29] RASPBERRY PI FOUNDATION: *SSH (SECURE SHELL)*. https://www.raspberrypi.org/documentation/remote-access/ssh/ Accessed: 10/12/2015,

[30] SAILER, Thomas: *Linux USB Utilities - lsusb*. http://linuxcommand.org/man_pages/lsusb8.html Accessed: 05/12/2015,

[31] UPTON, Liz: *Five Million Sold*. https://www.raspberrypi.org/blog/five-million-sold/ Accessed: 22/01/2016,

[32] VACCA, John R.: *Guide to Wireless Network Security*. Springer Science & Business Media, 2006. – ISBN 9780387298450

[33] XORG FOUNDATION: *Advanced Topics FAQ*. http://www.x.org/wiki/AdvancedTopicsFAQ/ Accessed: 05/12/2015,

A. RT5370 Wireless Adapter *lsusb -v* Output

This stdout of the linux command *lsusb -v* is only showing the part containing information regarding the wireless adapter RT5370. The information about the wireless adapter relevant for the system is shown in line 26 with the keyword *MaxPower*. As can be seen, the maximum current draw set for RT5370 is 450mA.

```
 1 Bus 001 Device 004: ID 148f:5370 Ralink Technology, Corp. RT5370
      Wireless Adapter
 2 Device Descriptor:
 3   bLength                18
 4   bDescriptorType         1
 5   bcdUSB               2.00
 6   bDeviceClass            0 (Defined at Interface level)
 7   bDeviceSubClass         0
 8   bDeviceProtocol         0
 9   bMaxPacketSize0        64
10   idVendor           0x148f Ralink Technology, Corp.
11   idProduct          0x5370 RT5370 Wireless Adapter
12   bcdDevice            1.01
13   iManufacturer           1
14   iProduct                2
15   iSerial                 3
16   bNumConfigurations      1
17   Configuration Descriptor:
18     bLength                 9
19     bDescriptorType         2
20     wTotalLength           67
21     bNumInterfaces          1
22     bConfigurationValue     1
23     iConfiguration          0
24     bmAttributes         0x80
25       (Bus Powered)
26     MaxPower              450mA
27     Interface Descriptor:
    bLength                 9
    bDescriptorType         4
    bInterfaceNumber        0
    bAlternateSetting       0
    bNumEndpoints           7
    bInterfaceClass       255 Vendor Specific Class
    bInterfaceSubClass    255 Vendor Specific Subclass
    bInterfaceProtocol    255 Vendor Specific Protocol
    iInterface              5
    Endpoint Descriptor:
      bLength                 7
      bDescriptorType         5
      bEndpointAddress     0x81  EP 1 IN
      bmAttributes            2
        Transfer Type            Bulk
        Synch Type               None
        Usage Type               Data
      wMaxPacketSize     0x0200  1x 512 bytes
      bInterval               0
    Endpoint Descriptor:
      bLength                 7
      bDescriptorType         5
      bEndpointAddress     0x01  EP 1 OUT
      bmAttributes            2
        Transfer Type            Bulk
        Synch Type               None
        Usage Type               Data
      wMaxPacketSize     0x0200  1x 512 bytes
      bInterval               0
    Endpoint Descriptor:
      bLength                 7
      bDescriptorType         5
      bEndpointAddress     0x02  EP 2 OUT
      bmAttributes            2
        Transfer Type            Bulk
        Synch Type               None
        Usage Type               Data
      wMaxPacketSize     0x0200  1x 512 bytes
      bInterval               0
    Endpoint Descriptor:
      bLength                 7
      bDescriptorType         5
      bEndpointAddress     0x03  EP 3 OUT
      bmAttributes            2
```

```
          Transfer Type            Bulk
          Synch Type               None
          Usage Type               Data
        wMaxPacketSize     0x0200  1x 512 bytes
        bInterval               0
      Endpoint Descriptor:
        bLength                 7
        bDescriptorType         5
        bEndpointAddress     0x04  EP 4 OUT
        bmAttributes            2
          Transfer Type            Bulk
          Synch Type               None
          Usage Type               Data
        wMaxPacketSize     0x0200  1x 512 bytes
        bInterval               0
      Endpoint Descriptor:
        bLength                 7
        bDescriptorType         5
        bEndpointAddress     0x05  EP 5 OUT
        bmAttributes            2
          Transfer Type            Bulk
          Synch Type               None
          Usage Type               Data
        wMaxPacketSize     0x0200  1x 512 bytes
        bInterval               0
      Endpoint Descriptor:
        bLength                 7
        bDescriptorType         5
        bEndpointAddress     0x06  EP 6 OUT
        bmAttributes            2
          Transfer Type            Bulk
          Synch Type               None
          Usage Type               Data
        wMaxPacketSize     0x0200  1x 512 bytes
        bInterval               0
```

B. The Default Crontab of Raspbian OS

The default version of the crontab from Raspbian OS:

```
# Edit this file to introduce tasks to be run by cron.
#
# Each task to run has to be defined through a single line
# indicating with different fields when the task will be run
# and what command to run for the task
#
# To define the time you can provide concrete values for
# minute (m), hour (h), day of month (dom), month (mon),
# and day of week (dow) or use '*' in these fields (for 'any').#
# Notice that tasks will be started based on the cron's system
# daemon's notion of time and timezones.
#
# Output of the crontab jobs (including errors) is sent through
# email to the user the crontab file belongs to (unless redirected
   ).
#
# For example, you can run a backup of all your user accounts
# at 5 a.m every week with:
# 0 5 * * 1 tar -zcf /var/backups/home.tgz /home/
#
# For more information see the manual pages of crontab(5) and cron
   (8)
#
# m h  dom mon dow   command
```

C. Configuration File *labels*

The default version of the *labels* configuration file:

```
1301a=1301a
1301b=1301b
1303a=1303a
1303b=1303b
1360=1360
1365=1365
1381=1381
1380=1380:\nStudierenden Arbeitsraum\nAnsprechpartner: Herr Prof.
   Dr. Dierks
1381a=1381a:\nProf. Dr. Thomas Klinker\nProf. Dr. Aining Li
1382=1382: Kopierstation Farbdrucker
1383=1383:\nBSc Stephanie Boehning\nDipl.-Ing. Michael Sparenborg
1384=1384:\nDipl.-Ing. Krystian Forsztega\nDipl.-Ing. Jerzy Janusz
1385=1385:\nPC-Pool Server
1386=1386:\nStudierenden Arbeitsraum\nAnsprechpartner: Herr Prof.
   Dr. Dierks
1387=1387:\nServer - Testraum
1388=1388:\nOrientierungseinheit (OE)\nDepartments: Inf. &
   Elektrotechnik - Informatik
1362=1362:\nFAUST - Projekt\nTeststreckenraum
1302=1302:\nWerkstatt
1302a=1302a:\nBeratung
```

The corresponding effect of the default *labels* configuration file on the GUI background can be found on the next page (Figure C.1).

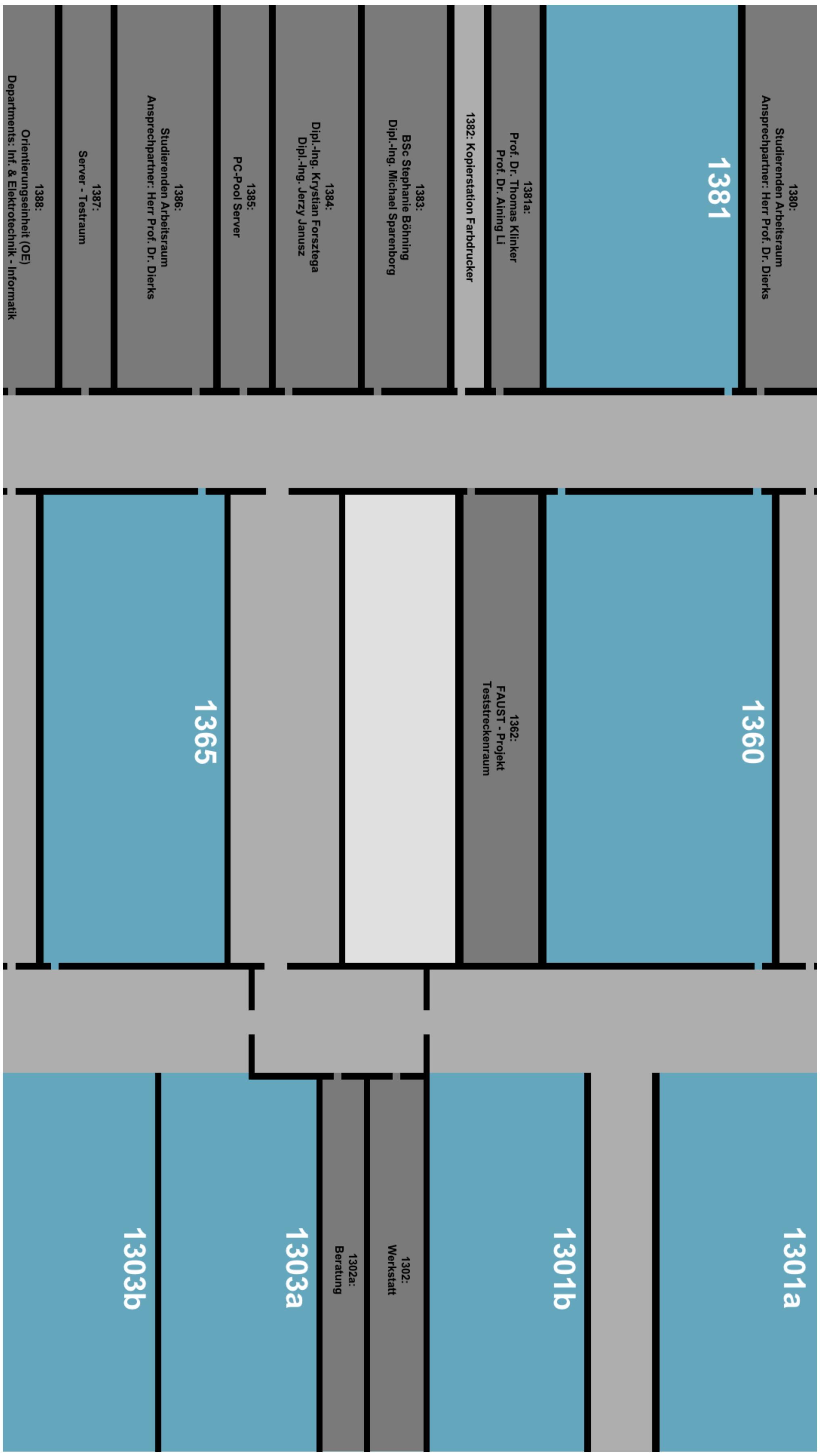


Figure C.1.: Resulting labels defined by the default *labels* configuration file

D. DVD Contents

This Bachelor Thesis contains an appendix of source codes, hardware documentation and a clone of the final Rasbian operating system on a DVD. This Appendix is deposited with Prof. Dr. rer. nat. Henning Dierks.

Declaration

I declare within the meaning of section 25(4) of the Ex-amination and Study Regulations of the International De-gree Course Information Engineering that: this Bachelor report has been completed by myself inde-pendently without outside help and only the defined sources and study aids were used. Sections that reflect the thoughts or works of others are made known through the definition of sources.

Hamburg, February 15, 2016	
City, Date	sign