AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE
WYDZIAŁ FIZYKI I INFORMATYKI STOSOWANEJ

KATEDRA INFORMATYKI STOSOWANEJ I FIZYKI KOMPUTEROWEJ

Projekt dyplomowy

System sterowania i automatyzacji pracy komory klimatycznej
Climate chamber control and automation system

Autor: Dawid Jan Wojdylak
Kierunek studiów: Informatyka Stosowana
Opiekun pracy: dr hab. inż. Andrzej Wetula

Kraków, 2023

Spis treści

1 Wprowadzenie

W obecnych czasach bardzo szybkiego rozwoju techniki ogromna liczba różnego rodzaju urządzeń trafia do użytku ludzi. Aby urządzenia te cechowała trwałość, niezawodność oraz bezpieczeństwo używania, często na etapie przedprodukcyjnym wykonuje się na nich skomplikowane testy. Dotyczy to szczególnie przedmiotów wykorzystywanych na zewnątrz, a więc narażonych na zróżnicowane warunki atmosferyczne. Jednym z rodzajów takich testów są testy z użyciem tzw. komór klimatycznych (szaf klimatycznych).

Komora klimatyczna jest urządzeniem, które na pierwszy rzut oka może przypominać lodówkę albo piekarnik. Skojarzenie to jest trafne, ponieważ w wolnej interpretacji jest to ich połączenie. Urządzenie składa się z izolowanej wnęki zamykanej izolowanymi drzwiczkami najczęściej wyposażonymi w szybę. W pozostałej części obudowy znajdują się odpowiednie moduły grzewcze i chłodzące. Wszystko po to, aby mieć możliwość utrzymania lub zmiany temperatury oraz wilgotności (czasem również innych parametrów) w dowolnym czasie. W zależności od skali testów, zmienia się skala rozmiaru używanych komór - od tych porównywalnych do kuchenki mikrofalowej, do wielkich komór, do których można wejść. Jeden z przykładów takiej komory widoczny jest na rysunku 1.

Rysunek 1: Zdjęcie przedstawiające komorę klimatyczną. Źródło: [1]

Komory klimatyczne są używane głównie w przemyśle motoryzacyjnym, farmaceutycznym, budowlanym oraz elektrotechnicznym. Testuje się w nich właściwości i stabilność materiałów w różnych warunkach klimatycznych, odporność na warunki atmosferyczne, a nawet symuluje się ich postarzanie. W dalszej części pracy komora/szafa klimatyczna będzie nazywana komorą.

1.1 Cele pracy

Zadaniem autora tego projektu było stworzenie systemu, który umożliwia sterowanie komorą klimatyczną za pomocą aplikacji komputerowej. Składa się on z komory (lub wielu komór), serwera zarządzającego komorą oraz aplikacji z graficznym interfejsem użytkownika. Jednym z założeń była generyczność programu, czyli możliwość zastosowania go do wielu modeli komór. Aplikacja miała też przeprowadzać proces autoryzacji użytkownika. Ostatnim założeniem była możliwość generowania skryptów dla komory, które są zbiorem instrukcji w określonym czasie.

2 Wykorzystane technologie i narzędzia

2.1 Aplikacja użytkownika

Python

Python to otwartoźródłowy (ang. *Open-source*) interpretowany język programowania wysokiego poziomu. Został stworzony przez Guido van Rossum na początku lat 90. i jest rozwijany do dziś [2]. Jest jednym z najczęściej używanych języków programowania. Swoją popularność zawdzięcza m.in. łatwości i szybkości pisania czytelnego i zwięzłego kodu, a także wysoką uniwersalnością. Język ten stosuje tzw. dynamiczne typowanie. Polega ono na przypisywaniu typów zmiennym w trakcie działania programu, w przeciwieństwie do typowania statycznego. Takie rozwiązanie ma swoje wady, np. gorsza wydajność programu, czy utrudnione debugowanie. Język Python wspiera różne paradygmaty programowania, np. programowanie obiektowe, funkcyjne oraz proceduralne. Programista ma do dyspozycji niezwykle bogatą gamę bibliotek standardowych oraz zewnętrznych.

SSH

SSH (ang. *secure shell*) to protokół komunikacyjny używający sieci TCP/IP[1] [3]. Cechuje go bezpieczeństwo transferu danych, ponieważ jest to połączenie szyfrowane. Realizuje typowe połączenie klient-serwer. Serwer SSH uruchamia demony (ang. *daemon*)[2], z którymi może połączyć się klient. Program klienta jest wspierany przez większość systemów operacyjnych, w tym Windows, macOS i Linux.

Paramiko

Paramiko jest przeznaczoną dla Pythona biblioteką obsługującą połączenia SSH. Posiada wysokopoziomowy interfejs, dzięki czemu można łatwo nawiązać połączenie i z niego korzystać. Przykład użycia na ldistingu 1.

```
client = SSHClient()
client.load_system_host_keys()
client.connect('ssh.example.com')
stdin, stdout, stderr = client.exec_command('ls␣-l')
```

Listing 1: Przykład użycia połączenia SSH jako klient z użyciem biblioteki Paramiko. Źródło: [4]

XML

XML (ang. *Extensible Markup Language*) to rozszerzalny język znaczników [5]. Służy do reprezentowania danych w formie tekstowej. Używa plików z rozszerzeniem `.xml`. Struktura danych jest strukturą drzewa[3]. Jego zaletą jest dowolność w nazewnictwie wszystkich węzłów. Przykład użycia dokumentu XML na listingu 2.

[1] (ang. *Transmition Control Protocol/Internet Protocol)* to zbiór protokołów przeznaczonych do transmisji danych przez sieci komputerowe.

[2] Program wielozadaniowego systemu operacyjnego, który wykonuje procesy w tle.

[3] Nieskierowany, acykliczny i spójny graf.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <item>
        <subitem></subitem>
    </item>
</root>
```

Listing 2: Prosty przykład dokumentu XML

xml.etree.ElementTree

xml.etree.ElementTree to wydajne i proste API[4] dla języka Python [6]. Klasa ElementTree umożliwia parsowanie[5] dokumentów XML oraz sprawne poruszanie się po nich.

PyQt

PyQt jest oficjalnym API Qt dla języka Python [7]. Qt to wieloplatformowe oprogramowanie komputerowe zorientowane obiektowo. Zawiera biblioteki do tworzenia graficznego interfejsu użytkownika (ang. *GUI, graphical user interface*). Oryginalnie przeznaczony jest do języków C++, Java oraz dla dedykowanego języka QML.

Qt oprócz narzędzi do tworzenia GUI zawiera własne implementacje wielu innych narzędzi, np. obsługa plików, wielowątkowości, sieci, testów jednostkowych, grafiki 3D, własne typy zmiennych i wiele innych.

Innowacyjną cechą Qt jest obecność sygnałów i slotów [8]. Używa się ich pomiędzy dwiema instancjami klas. Qt ma wiele zaimplementowanych sygnałów (np. zdarzenie kliknięcia lewym przyciskiem myszy w obrębie okna głównego aplikacji). Można również emitować własne sygnały. Sygnały odbierane są przez funkcje zwane slotami. Mogą przyjmować dowolne argumenty. Jest to mechanizm szczególnie przydatny przy projektowaniu interfejsów graficznych, ponieważ wyjątkowo często zachodzi tam potrzeba komunikowania różnych zdarzeń (np. kliknięcie przycisku). Mechanizm sygnałów i slotów jest łatwiejszą i bardziej zwięzłą alternatywą dla techniki wywołania zwrotnego (ang. *callback*). Umożliwia zastosowanie wzorca projektowego[6] (ang. *design pattern*) obserwator[7].

QtChart jest biblioteką służącą do tworzenia wykresów w Qt. Istnieje kilka innych rozwiązań, które pomagają w tworzeniu wykresów. Popularny Matplotlib jest dobry do tworzenia wykresów matematycznych, czy statystycznych. Jednak jego personalizacja, taka jak obsługa zdarzeń myszy na wykresie, nie jest tak łatwa, jak w dedykowanej bibliotece Qt. Dostępny jest jeszcze pyqtgraph, który świetnie radzi sobie z obsługą zdarzeń, a jego wykresy są bardzo responsywne. Dokumentacja tej biblioteki zawiera sporo przykładów, ale nie jest wystarczająco dobrze napisana - autor nie znalazł sposobu, aby przeciążyć i spersonalizować obsługę myszy. Dlatego też zdecydowano się na użycie biblioteki QtChart, która ma bardzo dobrą dokumentację i świetnie wpasowuje się w środowisko Qt.

[4](ang. *application programming interface*) to interfejs programistyczny aplikacji, czyli przepis na porozumiewanie się dwóch lub więcej języków programowania.

[5]Przetwarzanie i analizowanie łańcuchów znakowych w celu wyodrębnienia z nich informacji.

[6]Opis rozwiązań popularnych problemów w informatyce.

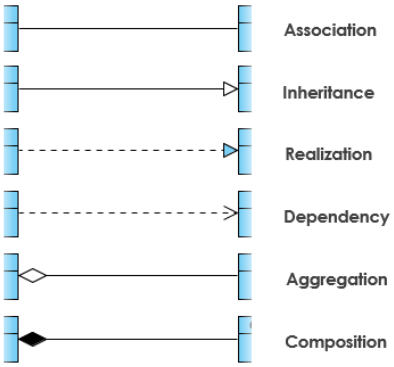
[7]Wzorzec projektowy używany do powiadamiania obiektów znajdujących się w grupie zainteresowanych o zmianach stanu głównego obiektu.

unittest

unittest jest zorientowanym obiektowo frameworkiem[8] przeznaczonym do wykonywania testów jednostkowych[9] [9]. Inspirowany jest narzędziem znanym z języka programowania Java - JUnit.

Język UML

Język UML (ang. *Unified Modeling Language*) to zbiór narzędzi do projektowania i wizualizacji systemów. Doskonale sprawdza się w modelach obiektowych. Oferuje wiele rodzajów diagramów, zarówno behawioralnych, jak i strukturalnych. Język ten jest jest oficjalnie rozwijany przez konsorcjum OMG (*Object Management Group*), które jest wspierane przez ok. 300 firm informatycznych (Źródło [10]). W tej pracy wykorzystano UML do stworzenia diagramów klas.



Rysunek 2: Rodzaje relacji pomiędzy klasami w języku UML. Źródło: [11]

Na rysunku 2. przedstawiono następujące związki pomiędzy klasami zdefiniowane w języku UML:

- skojarzenie (ang. *association*) - obiekty dwóch klas są w pewien sposób związane ze sobą, np. wykres jest związany z oknem głównym aplikacji, które go wyświetla. Skojarzenie przedstawione jest za pomocą ciągłej linii.
- Dziedziczenie (ang. *inheritance*) lub uogólnienie (ang. *generalization*) to implementacja szczególnego przypadku klasy, np. klasa `UI_Login` reprezentująca okienko logowania użytkownika dziedziczy po `QDialog` z przestrzeni `QtWidgets`. Dziedziczenie reprezentuje linia ciągła zakończona niewypełnionym trójkątem.
- Realizacja (ang. *realization*) oznaczana jest linią przerywaną zakończoną zamalowanym trójkątem. Oznacza, że klasa realizuje interfejs, czyli abstrakcyjną klasę deklarującą pola, ale bez ich implementacji.
- Zależność (ang. *dependency*) jest oznaczana linią przerywaną zakończoną strzałką i reprezentuje sytuację, w której klasa korzysta lub wykonuje operacje na obiektach innej klasy, ale bez potrzeby przechowywania ich.
- Agregacja (ang. *aggregation*) oznaczana jest linią prostą zakończoną rombem o pustym wnętrzu. Stosowana jest w sytuacjach, gdy obiekty klasy są przechowywane w drugiej klasie. Mogą jednak być jednocześnie przechowywane w innych klasach i nie są niszczone wraz z usuwaniem obiektu, który je przechowuje.
- Kompozycja (ang. *composition*) to silniejszy wariant agregacji. Oznacza się ją linią ciągłą zakończoną zamalowanym rombem. Obiekty jednej klasy są przechowywane w drugiej klasie i bez niej nie istnieją.

[8] Zestaw bibliotek przeznaczony do tworzenia oprogramowania. Przez dostarczenie wcześniej zaimplementowanych narzędzi i komponentów często przyspiesza proces pisania kodu.

[9] Jeden z rodzajów testowania oprogramowania. Jego najważniejszą cechą jest testowanie pojedynczych jednostek programu. Testy odbywają się porównywanie wyników z wartościami oczekiwanymi.

2.2 Serwer

Ubuntu Serwer

Ubuntu Server w wersji LTS 22.04 (ang. *Long Term Support*) jest wariantem zwykłego Ubuntu, lecz pozbawionym interfejsu graficznego GNOME. Używa się go za pomocą wiersza poleceń (ang. *Command-Line Interface, CLI*) powłoki systemowej (ang. *shell*)[10] bash.

Ubuntu to jedna najpopularniejszych dystrybucji systemów operacyjnych z rodziny Linux. Jest oparta na systemie Debian. Korzysta ze środowiska graficznego Unity, które jest nakładką na GNOME. Źródło: [12].

Cron

Cron jest to jeden z demonów systemów typu Unix[11]. Korzysta się z niego za pomocą pliku crontab. Pozwala na uruchamianie komend i skryptów cyklicznie lub o wyznaczonych godzinach. Polecenie `crontab -e` otwiera plik konfiguracyjny dla bieżącego użytkownika. Następnie do pliku wpisuje się linijki tekstu o następującym formacie:

`[minuta]␣[godzina]␣[dzień miesiąca]␣[miesiąc]␣[dzień tygodnia]␣[polecenie]`

Stosuje się następujące znaki:

- `*` - każda wartość,
- `,` - separator wielu wartości,
- `-` - zakres wartości,
- `/` - krok.

Przykłady:

- `0 0-23 * * * who >> ~/who_log.txt` logowanie zalogowanych użytkowników co godzinę,
- `15-45/15 22 * * 1-5,7 ~/my_script.sh` uruchamianie skryptu `my_script.sh` codziennie z wyłączeniem soboty o 22:15, 22:30 i 22:45.

Źródło: [13]

Ethernet

Ethernet to zbiór zasad określających proces przepływu danych. Opisuje format i proces transmisji ramek, a także specyfikację budowy przewodów.

Do komunikacji serwera Ubuntu z komorą przez protokół Ethernet użyto biblioteki do języka Python o nazwie **socket**. Biblioteka ta umożliwie m. in. skomunikowanie dwóch jednostek istniejących w sieci komputerowej (połączenie klient-serwer).

[10] Program komputerowy będący graficznym lub tekstowym pośrednikiem pomiędzy komputerem a człowiekiem.

[11] Popularny w latach 70. i 80. system operacyjny, którego implementacje używane są do dziś, np. Linux, macOS.

Socket umożliwia wykonanie niskopoziomowych instrukcji napisanych w języku C za pomocą wysokopoziomowego języka Python. Jest biblioteką zorientowaną obiektowo. Aby z niej skorzystać, tworzy się obiekt klasy socket:

```
s = socket(AF_INET, SOCK_STREAM).
```

Konstruktor wymaga podania rodziny adresów (w tym przypadku `AF_INET` do IPv4) oraz typu połączenia (domyślnie `SOCK_STREAM`). Aby połączyć się do serwera jako klient, korzysta się z metody `connect(HOST, PORT)`. Jako argumenty przyjmuje adres IP serwera oraz port. Do wysyłania bajtów służy metoda `send(byte)`. Do odbierania metoda `recv(bufsize)`, gdzie `bufsize`, to argument reprezentujący maksymalny rozmiar bufora, który chcemy odebrać. Odpowiedź serwera zwracana jest w formie bajtów. Źródło: [14].

2.3 Narzędzia

Visual Studio Code

Visual Studio Code to edytor tekstu przeznaczony do programowania stworzony przez firmę Microsoft. Jest darmowy, otwartoźródłowy i wieloplatformowy. Posiada bardzo dużo cech, które pomagają w pisaniu kodu, np. wsparcie debugowania, kolorowanie i podpowiadanie składni, obsługa systemu kontroli wersji Git, czy wbudowany terminal. Dodatkowo istnieje możliwość instalowania dodatkowych wtyczek, które jeszcze bardziej rozszerzają jego możliwości.

QtCreator

QtCreator jest wieloplatformowym zintegrowanym środowiskiem programistycznym (ang. *integrated development environment*, IDE) będącym częścią rodziny Qt. W tym projekcie dyplomowym nie korzystano jednak z jego podstawowych funkcjonalności pisania i edycji kodu, a używano jedynie tzw. UI Designera. Jest to graficzne narzędzie umożliwiające zaprojektowanie interfejsu graficznego tworzonej aplikacji. Projekt zapisuje się do pliku z rozszerzeniem `.ui`. Po zapisaniu takiego pliku należy wygenerować plik pythonowy `.py` korzystając z polecenia `pyuic5` (Python User Interface Compiler, część biblioteki Qt).

Git

Git jest darmowym i otwartoźródłowym rozproszonym systemem kontroli wersji. Służy głównie do śledzenia zmian w kodzie i ułatwia pracę nad projektem tworzonym przez wiele osób. Twórcą Gita Linus Torvalds, który jest też twórcą jądra Linuksa.

Visual Paradigm

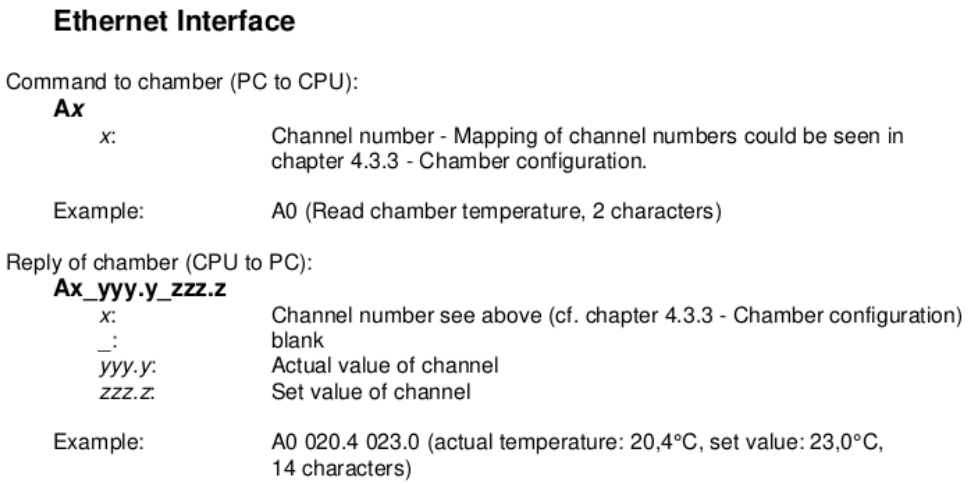
Visual Paradigm jest wieloplatformowaym narzędziem przeznaczonym do języka UML. Jego twórcą jest Visual Paradigm International Ltd. Główną funkcjonalnością programu jest możliwość zaawansowanego tworzenia i edycji diagramów UML. Dostępna jest wersja programu do uruchomienia w przeglądarce internetowej: Visual Paradigm Online.

3 Implementacja systemu

3.1 Plik konfiguracyjny aplikacji użytkownika

Jednym z założeń tworzenia aplikacji była jej generyczność. Używane komory klimatyczne często różnią się modelami, a nawet producentami. Dlatego jednym z pierwszych problemów było opracowanie pliku konfiguracyjnego dla aplikacji użytkownika. Taki plik ma być interpretowalny przez aplikację oraz ma dostarczyć jej informacji o sposobie komunikacji między komorą a aplikacją uwzględniając komunikację w obie strony. Krócej: ma być to zbiór przepisów na porozumiewanie się z komorą.

Zdecydowano, że dobrym wyborem będzie użycie języka XML ze względu na jego elastyczność oraz łatwość parsowania w języku Python.



Ethernet Interface

Command to chamber (PC to CPU):

Ax

x:	Channel number - Mapping of channel numbers could be seen in chapter 4.3.3 - Chamber configuration.
Example:	A0 (Read chamber temperature, 2 characters)

Reply of chamber (CPU to PC):

Ax_yyy.y_zzz.z

x:	Channel number see above (cf. chapter 4.3.3 - Chamber configuration)
_:	blank
yyy.y:	Actual value of channel
zzz.z:	Set value of channel
Example:	A0 020.4 023.0 (actual temperature: 20,4°C, set value: 23,0°C, 14 characters)

Rysunek 3: Fragment dokumentacji komory klimatycznej marki CTS przedstawiający strukturę komendy odczytującej wartości analogowych kanałów komory

Przepis na plik konfiguracyjny

Plik konfiguracyjny opracowywano zgodnie z dokumentacją komory. Jej fragment przedstawiono na rysunku 3. Dotyczy on odczytu kanałów analogowych komory, takich jak kanał temperatury, wilgotności, czy zapełnienia zbiornika z wodą.

Na listingu 3. przedstawiono ogólny przepis na plik konfiguracyjny. linijka 1. to deklaracja o użyciu XML-a, jego wersja i kodowanie znaków[12].

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <commands>
3      <config>
4          <chamberName>Komora Sala 2.16</chamberName>
5          <ip>10.224.157.82</ip>
6      </config>
7      <command name="[Nazwa komendy]">
8          <description>[Opis komendy]</description>
9          <request>
10             <argument>[Pierwszy argument zapytania]</argument>
```

[12]Reprezentacja znaków interpretowalna przez komputer.

```
        </request>
        <response>
            <argument>[Pierwszy argument odpowiedzi]</argument>
            <argument descr="Opis odpowiedzi">[Kolejny argument
    odpowiedzi (z opisem)]</argument>
        </response>
    </command>
</commands>
```

Listing 3: Ogólna struktura pliku konfiguracyjnego .xml

Węzłem stanowiącym korzeń dokumentu (ang. *root element*) jest węzeł `<commands>`. Pierwszym podwęzłem jest węzeł `<config>`. Zawiera węzły `<chamberName>` z nazwą komory oraz `ip` z adresem IP do serwera. Kolejnymi podwęzłami (ang. *child*) są elementy `<command>` i każdy z nich reprezentuje jedną komendę, za pomocą której program może komunikować się z komorą. Elementy typu `<command>` muszą mieć atrybut `name=""`, co wynika ze sposobu parsowania pliku przez program - `name` staje się nazwą obiektu wyrażającego komendę. Jest to jedyny obowiązkowy atrybut w pliku konfiguracyjnym. Jeżeli `<command>` zawiera argumenty z wartościami wprowadzanymi przez użytkownika (np. ustawienie temperatury), to powinien mieć atrybut `type=` z wartością `"user_value"`

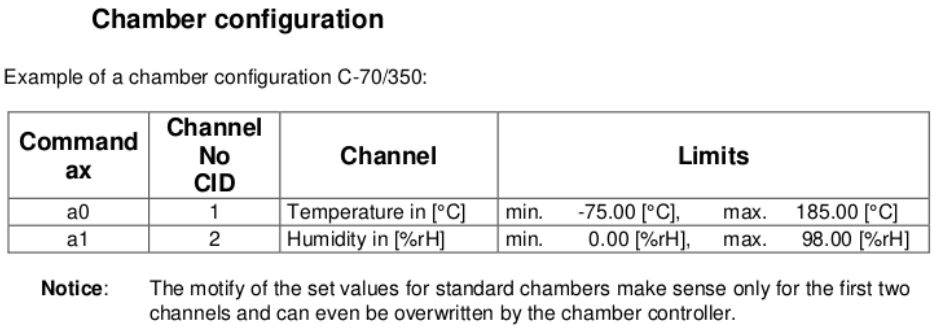
Węzły `<command>` zawierają podwęzły `<request>` oraz `<response>`. Zawartość pierwszego z nich służy do kompletowania wiadomości wysyłanej do komory. Zawartość drugiego pomaga rozkodować odpowiedź komory. Oba rodzaje węzłów składają się z podwęzłów `<argument>`. Zawartość subwęzłów `<argument>` to znak lub łańcuch znaków, który zostanie wysłany lub odebrany z komory.

Możliwe jest też użycie następujących atrybutów węzłów `<argument>`:

- `descr` - nazwa opisująca argument,
- `min` - minimalna możliwa wartość argumentu,
- `max` - maksymalna możliwa wartość argumentu,
- `unit` - jednostka argumentu,
- `type` - modyfikowalność argumentu (jeśli jest modyfikowalny przez użytkownika, powinien mieć wartość „user_value”) albo znak spacji (wartość „blank”).

Przykład komendy typu „set”

Listing 4. przedstawia przykładową budowę węzła komendy (`<command>`), która ma za zadanie zadać wartość komorze. Ten węzeł został napisany zgodnie z dokumentacją komory, której odpowiednie fragmenty znajdują się na rysunku 3. oraz rysunku 4.



Chamber configuration

Example of a chamber configuration C-70/350:

Command ax	Channel No CID	Channel	Limits			
a0	1	Temperature in [°C]	min.	-75.00 [°C],	max.	185.00 [°C]
a1	2	Humidity in [%rH]	min.	0.00 [%rH],	max.	98.00 [%rH]

Notice: The motify of the set values for standard chambers make sense only for the first two channels and can even be overwritten by the chamber controller.

Rysunek 4: Fragment dokumentacji komory klimatycznej marki CTS przedstawiający tabelę dostępnych kanałów analogowych z możliwością ustawiania wartości

Komenda ma ustawioną nazwę „set_temperature”[13], aby czytelnie przedstawić użytkownikowi pełnioną przez nią funkcję. Wypełniony jest też atrybut `type=` za pomocą wartości `"user_value"` w celu zasygnalizowania aplikacji, że ta komenda wymaga wprowadzenia wartości przez użytkownika. Pierwszy i drugi `<argument>` (linijka 4. i 5.) zawierają znaki, których wymaga dokumentacja, aby zmienić wartość temperatury. Wartością kolejnego argumentu z linijki 6. jest znak spacji. Ma on atrybut `type=` z wartością `"blank"`, aby zapewnić program, że w tym miejscu jest znak spacji. Ostatni `<argument>` (linijka 7.) jest najbogatszy w atrybuty, aby poinformować program o fakcie, że to wartość tego argumentu należy zmienić uwzględniając minimum i maximum wartości. Zawiera też informację o jednostce wprowadzanej wartości.

```
1  ⋮
2  <command name="set_temperature" type="user_value">
3      <request>
4          <argument>a</argument>
5          <argument>0</argument>
6          <argument type="blank"> </argument>
7          <argument descr="set_value" min="-75.00" max="185.00" unit="
       celc_deg" type="user_value">
8              yyy.y
9          </argument>
10     </request>
11     <response>
12         <argument>a</argument>
13     </response>
14 </command>
15 ⋮
```

Listing 4: Przykład węzła komendy typu „set” pliku konfiguracyjnego XML. Fragment pliku CtsInterfaceProtocol.xml

Przykład komendy typu „read”

Listing 5. jest reprezentacją komendy, która jedynie odczytuje wartości z komory. W tym przypadku bardziej rozwinięty jest węzeł `<response>` (linijka 7.) i zawiera on argumenty z atrybutami opisującymi nazwy oraz jednostki wartości, o których

[13]Obecność znaku „_” jest tutaj ignorowana przy wyświetlaniu nazwy użytkownikowi.

informuje nas komora. W tym miejscu istotna jest liczba znaków każdego argumentu w węźle `<response>`, ponieważ program porównuje ją z łańcuchem znakowym uzyskanym z odpowiedzi komory.

```
 ⋮
<command name="read_temperature">
    <request>
        <argument>A</argument>
        <argument>0</argument>
    </request>
    <response>
        <argument>A</argument>
        <argument>0</argument>
        <argument type="blank"> </argument>
        <argument descr="actual␣temperature" unit="celc_deg">yyy.y</
    argument>
        <argument type="blank"> </argument>
        <argument descr="set␣temperature" unit="celc_deg">zzz.z</
    argument>
    </response>
</command>
 ⋮
```

Listing 5: Przykład węzła komendy typu „read” pliku konfiguracyjnego XML. Fragment pliku CtsInterfaceProtocol.xml

Kod fragmentu kodu na listingu 5. został napisany w oparciu o dokumentację komory, której odpowiednie fragmenty zostały przedstawione na rysunku 3. oraz rysunku 5.

Chamber configuration

Example of a chamber configuration C-70/350:

Value Ax	Channel No CID	Channel	Limits
A0	1	Temperature in [°C]	min. -75.00 [°C], max. 185.00 [°C]
A1	2	Humidity in [%rH]	min. 0.00 [%rH], max. 98.00 [%rH]
A2	3	Water storage in [l]	min. 0.00 [l], max. 15.00 [l]
A3	4	TempSupplyAir in [°C]	min. -75.00 [°C], max. 185.00 [°C]
A4	5	TempExhAir in [°C]	min. -75.00 [°C], max. 185.00 [°C]
A5	6	HumidSupplAir in [%rH]	min. 5.00 [%rH], max. 98.00 [%rH]
A6	7	HumidExhAir in [%rH]	min. 5.00 [%rH], max. 98.00 [%rH]

Rysunek 5: Fragment dokumentacji komory klimatycznej marki CTS przedstawiający tabelę dostępnych kanałów analogowych

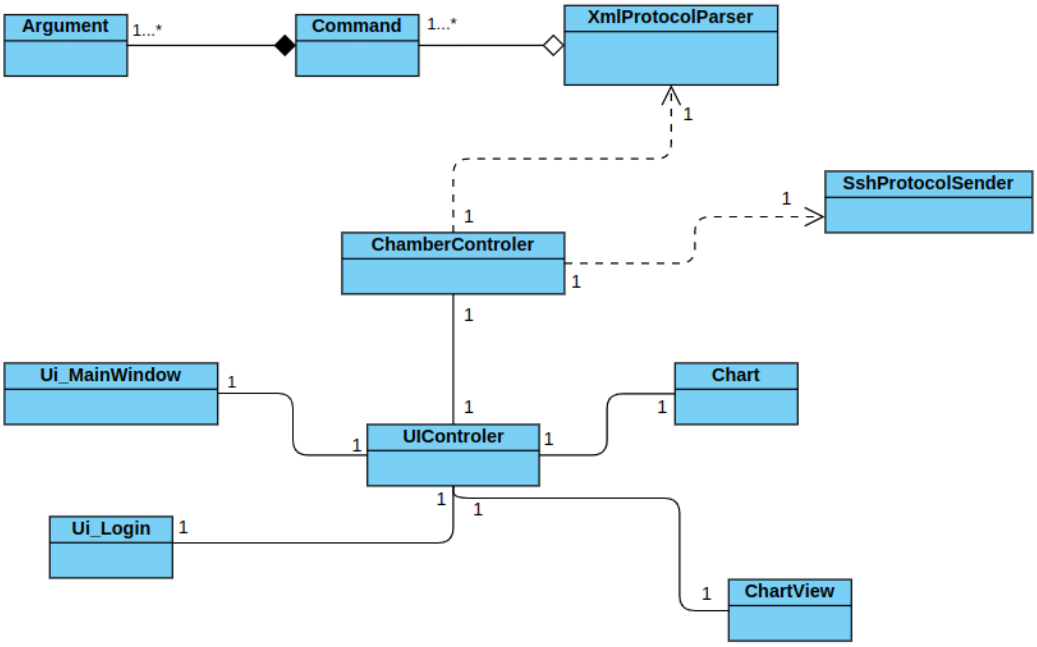
Krótkie podsumowanie

Dzięki takiemu podejściu wprowadzania komend komunikacyjnych do aplikacji, użytkownik może sam skomponować własny plik konfiguracyjny XML przestrzegając zasad jego tworzenia. Przez to aplikacja może być wykorzystywana z różnymi komorami, które obsługują komunikację przez Ethernet. Być może w przyszłości

zajdzie potrzeba stworzenia narzędzia z interfejsem graficznym, które ułatwiłoby tworzenie takiego pliku.

3.2 Implementacja aplikacji użytkownika

Schemat aplikacji użytkownika



Rysunek 6: Diagram UML przedstawiający relacje klas w aplikacji użytkownika

W pisaniu aplikacji użytkownika wykorzystano podejście obiektowe. Na rysunku 6. przedstawiono diagram UML klas. „Najmniejszą" częścią programu są obiekty klasy `Argument`. Obiekty tej klasy przechowywane są w klasie `Command`, do której komendy z pliku konfiguracyjnego XML parsuje klasa `XmlProtocolParser`. API backendu[14] stanowi klasa `ChamberControler`. Część frontendowa[15] to klasa `MainWindow` oraz klasa `UIControler`, która obsługuje większość zdarzeń w GUI. Klasy `Chart` oraz `ChartView` odpowiedzialne są za wykresy.

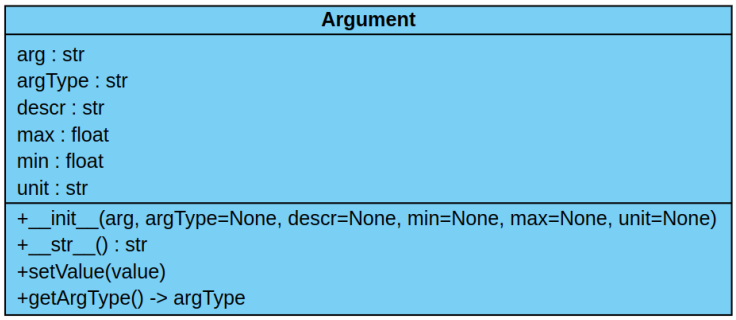
Klasa Argument

Klasa `Argument` realizuje węzły `<argument>` (listing 3.). Koncept klasy `Argument` przedstawiono na diagramie UML na rysunku 7. Konstruktor tej klasy wymaga tylko jednego argumentu, który reprezentuje zawartość argumentu z pliku konfiguracyjnego XML, czyli fragment wysyłanego do komory łańcucha znaków. Ten sam konstruktor poprzez argumenty domyślne umożliwia również wypełnienie dodatkowych pól klasy: `argType`, `descr`, `min`, `max`, `unit`. Jeżeli któraś z tych wartości nie zostanie dostarczona, zostanie przypisana do nich wartość `None`. Jest to odpowiednik

[14] Obszar oprogramowania niedostępny dla użytkownika. Zazwyczaj znajduje się w nim większość procesów i operacji programu.

[15] Obszar oprogramowania wchodzący w bezpośrednią interakcję z użytkownikiem.

wartości `null`[16]. Słowo kluczowe `self` w języku Python odnosi się do pól niestatycznych klasy.



Rysunek 7: Diagram UML opisujący klasę Argument

Przeciążono metodę `__str__()`. Obecność znaków „_” świadczy o fakcie, że mamy do czynienia z tzw. Magiczną Metodą (zwane też Metodą Dunder, ang. *Double Underscore*. Źródło:[15]). Są to metody, których wywoływanie odbywa się niejawnie (np. konstruktor). Metoda `__str__()` wywoła się, gdy np. obiekt zostanie podany jako argument funkcji `print()` lub bezpośrednio `str()`.

Ostatnie dwie metody to tzw. gettery i settery. W języku Python ich obecność nie jest wymagana, jako że pola klas nie są prywatne. Autor użył ich jednak w celu zachowania tzw. enkapsulacji[17] danych zgodnie z paradygmatem programowania obiektowego.

Klasa Command

Rysunek 8. prezentuje zarys klasy `Command`. Jest ona ekwiwalentem węzła `<command>` pliku konfiguracyjnego XML. Klasa posiada pola przechowujące nazwę komendy, wartości podwęzłów `<request>` i `<responsе>` (czyli obiekty klasy `Argument`) zaznaczono to na diagramie UML za pomocą linii z czarnym kwadratem/rombem reprezentującej w języku UML kompozycję. Pozostałe pola zawierają opis komendy, modyfikowalność komendy przez użytkownika oraz pole informujące program o ewentualnym wystąpieniu błędnego żądania.

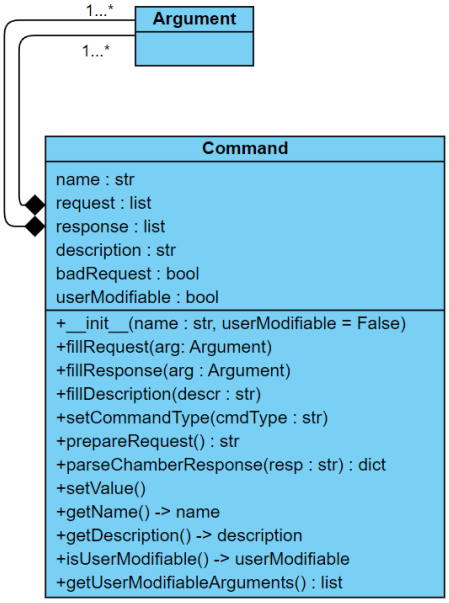
[16]Wartość występująca w językach programowania oznaczająca pustą wartość. Często jej interpretacja liczbowa wynosi 0.

[17]Poprawniej: hermetyzacja - ukrywanie pól klasy, aby były dostępne jedynie dla jej metod wewnętrznych.[16]

Metody `fillRequest()` i `fillResponse()` wypełniają odpowiednie listy obiektami klasy `Argument`. Funkcje `fillDescription()` oraz `setCommandType()` uzupełniają opis komendy oraz informację o tym, czy jest typem komendy modifykowalnym przez użytkownika.

Metoda `prepareRequest()` jest odpowiedzialna za posklejanie znaków obiektów klasy `Argument` przechowywanych w liście `self.request` w łańcuch znaków gotowy do wysłania do komory.

Metoda `parseChamberResponse()`, której implementacja została przedstawiona na listingu 6., rozkodowuje łańcuch znaków otrzymany w odpowiedzi od komory. Dokonuje tego na podstawie listy `self.response` przechowującej obiekty klasy `Argument`. Porównuje znaki, zlicza je i na tej podstawie rozdziela je i przypisuje im odpowiedni opis. Zwraca odpowiednio przygotowany słownik[18].



Rysunek 8: Diagram UML opisujący klasę Command

```
1     def parseChamberResponse(self, resp : str):
2         i = 0
3         result = {}
4         try:
5             for arg in self.response:
6                 temp_str = ""
7                 if i >= len(resp): break
8                 for _ in arg.arg:
9                     temp_str += resp[i]
10                    i += 1
11                if arg.descr != None:
12                    result[arg.descr] = temp_str
13            return result
14        except Exception as e:
15            raise
```

Listing 6: Implementacja metody `parseChamberResponse()` klasy `Command`

Metoda `setValue()` jest setterem, a `getName()`, `getDescription()`, `isUserModifiable()` oraz `getUserModifiableArguments()` to gettery. W ostatniej z metod zastosowano tzw. listy składane (ang. *list comprehension*). Jest to sposób tworzenia nowej listy na podstawie innej listy lub innego obiektu iterowalnego (Źródło: [18]). Przykład takiej listy i jednocześnie implementacja metody `getUserModifiableArguments()` umieszczono na listingu 7. Zwracana lista została utworzona poprzez przeiterowanie istniejącej listy `request` i jednoczesne sprawdzenie za pomocą instrukcji warunkowej `if`, czy argument przeznaczony jest do modyfikowania przez użytkownika. W tym miejscu skorzystano z gettera klasy `Argument`.

[18]Słownik, zwany też mapą lub tablicą asocjacyjną, to uogólnienie listy, którego indeksami mogą być inne wartości niż liczby naturalne. Źródło [17]

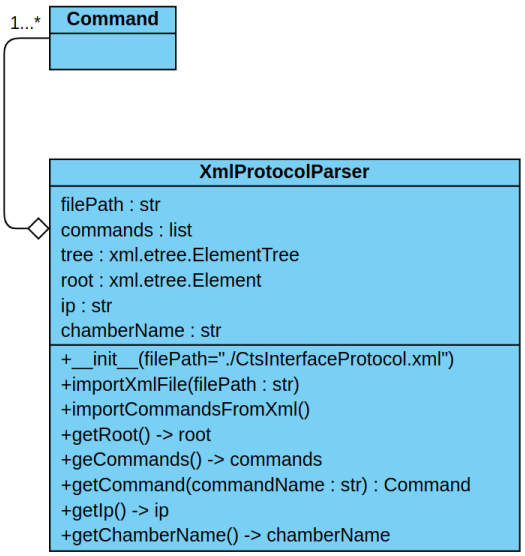
Składanie list jest charakterystyczną cechą języka Python i umożliwia wykonanie mniej lub bardziej złożonych operacji w jednej linijce kodu.

```
def getUserModifiableArguments(self):
    return [r for r in self.request if r.getArgType() == "user_value"]
```

Listing 7: Przykład listy składanej z metody `getUserModifiableArguments()` klasy `Command`

Klasa XmlProtocolParser

Na rysunku 9. widnieje schemat klasy `XmlProtocolParser`. Odpowiada ona za parsowanie pliku konfiguracyjnego XML oraz zmagazynowanie odpowiednio wypełnionych obiektów klas `Argument` i `Command`. Klasa zawiera pola przechowujące informacje o ścieżce do pliku konfiguracyjnego, listę komend (agregacja na diagramie UML), uchwyt do drzewa pliku XML oraz uchwyt do jego korzenia, a także łańcuch znaków adresu IP i nazwę komory. Konstruktor domyślny zapewnia domyślną ścieżkę do pliku konfiguracyjnego XML. Konstruktor automatycznie wywołuje metodę `importXmlFile()`, która wczytuje plik konfiguracyjny XML, parsuje go i zapisuje w poprzednio wymienionych polach klasy.



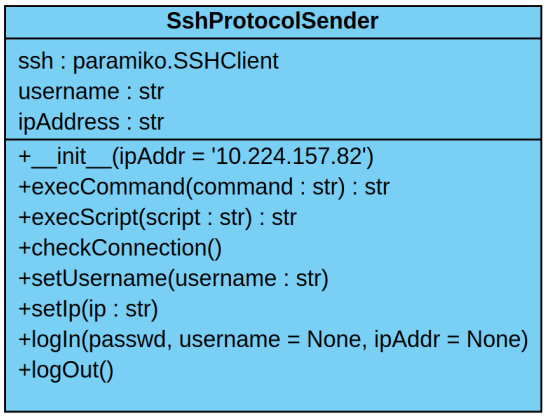
Rysunek 9: Schemat klasy `XmlProtocolParser`

Aby wypełnić listę komend, należy wywołać metodę `importCommandsFromXml()`. Getter `getCommands()` zwraca listę wszystkich komend, `getRoot()` zwraca korzeń XML, a `getCommand(commandName: str)` zwraca pojedynczą komendę dopasowując ją po nazwie. Metody `getIp()` oraz `getChamberName()` zwracają odpowiednio łańcuchy znaków z adresem ip oraz nazwą komory. Obie te wartości powinny znaleźć się w pliku konfiguracyjnym XML, z którego zostaną odczytane przez klasę `XmlProtocolParser`.

Klasa SshProtocolSender

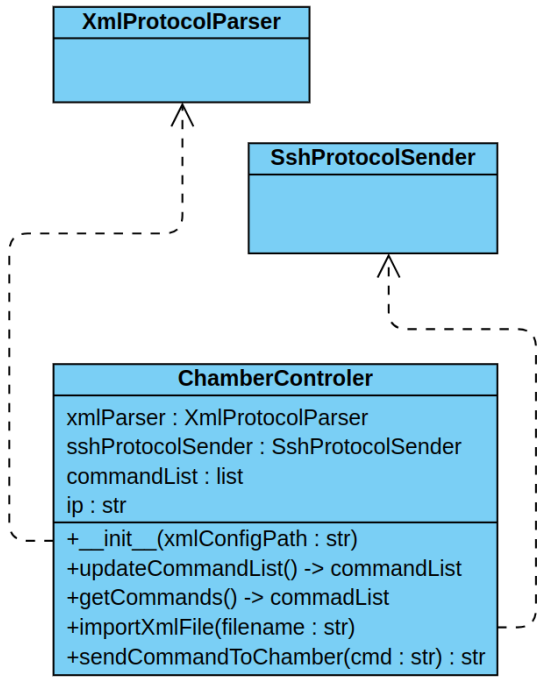
Przedstawiona na rysunku 10. klasa `SshProtocolSender` zawiera pola i funkcje odpowiedzialne za komunikację aplikacji użytkownika z serwerem Ubuntu. Klasa korzysta z frameworku Paramiko i implementuje klienta ssh. Zawiera pola przechowujące niezbędną do zalogowania nazwę użytkownika i adres ip serwera. Konstruktor zawiera domyślny adres ip, który można też uzupełnić za pomocą settera `setIp(ip)`. Funkcja `execCommand()` przyjmuje poprzednio przygotowaną komendę w formie łańcucha znaków i wysyła ją do serwera. Serwer następnie za pomocą protokołu Ethernet przekazuje tę komendę do komory. Jej odpowiedź zwracana jest przez tę samą funkcję.

Funkcja `execScript()` działa na identycznej zasadzie, co jej poprzednik, z tą różnicą, że przekazuje cały skrypt, który wygenerowany został na podstawie stworzonego przez użytkownika wykresu. Funkcja `checkConnection()` sprawdza, czy połączenie ssh jest aktywne. Zwraca odpowiednią wartość typu `bool`. Funkcja `setUsername()` wypełnia pole `username` nazwą użytkownika, potrzebną do zalogowania. Aby się zalogować, potrzebna jest metoda `login()`, której argumentem jest hasło. Dodatkowo można w niej podać nazwę użytkownika i adres IP serwera. Do wylogowania się służy metoda `logOut()`.



Rysunek 10: Zarys klasy `SshProtocolSender`

Klasa ChamberControler

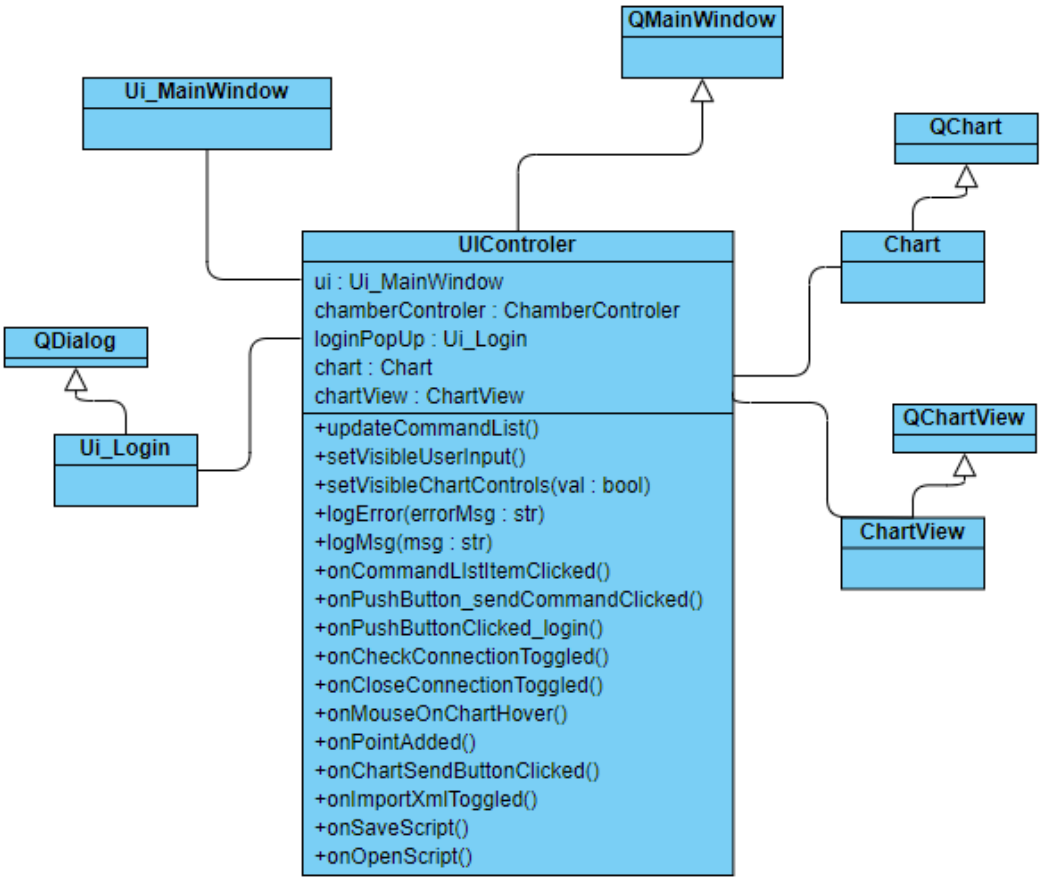


Rysunek 11: Koncept klasy `ChamberControler`

Klasa `ChamberControler`, przedstawiona na rysunku 11., stanowi interfejs backendu aplikacji (API). Jej działania wykorzystują funkcjonalność klas `XmlProtocolParser` oraz `SshProtocolSender`. Aby to zaznaczyć, użyto połączenia UML typu zależność (ang. *dependency*). Klasa posiada pola, przechowujące obiekty tych klas, które inicjalizowane są w konstruktorze. Konstruktor wymaga argumentu ze ścieżką do pliku konfiguracyjnego XML. Metoda `updateCommandList()` pobiera komendy z klasy `XmlProtocolParser`. Getter `getCommands()` zwraca listę komend. Funkcja `importXmlFile()` jako argument przyjmuje ścieżkę do pliku konfiguracyjnego XML i za pomocą klasy `XmlProtocolParser` parsuje i pobiera listę komend. Ostatnia metoda wysyła komendę do komory przez klasę `sshProtocolSender` i zwraca jej odpowiedź.

Klasy UI_MainWindow oraz UI_Login

Schemat klas `UI_MainWindow oraz UI_Login` został przedstawiony na rysunku 12. Są to klasy, których kod został wygenerowany z pliku `*.ui` przy użyciu pyuic5. Klasa `UI_MainWindow` tworzy szkielet okna głównego aplikacji. Poszczególne okna i kon-



Rysunek 12: Diagram UML prezentujący schemat klasy `UIControler`

trolki są obsługiwane lub dynamicznie dodawane w klasie `UIControler`. Powodem takiego podejścia jest uniknięcie edycji klasy `UI_MainWindow`, której kod zostałby nadpisany po każdym generowaniu pliku `MainWindow.py` z `MainWindow.ui`. Dzięki temu autor mógł niezależnie modyfikować zawartość interfejsu graficznego, a przy tym zachowywać postępy pozostałych prac.

Klasa UIControler

Klasa `UIControler`, przedstawiona na rysunku 12., reguluje wszystko, co dzieje się w graficznym interfejsie użytkownika. Dziedziczy po klasie `QMainWindow` z przestrzeni `QtWidgets`. Takie podejście jest standardowe w programowaniu w technologii Qt i pozwala korzystać ze wszystkich gotowych metod `QMainWindow`, przeciążać je i implementować własne na potrzeby tworzenia GUI.

Konstruktor klasy `UIControler` na samym początku explicite wywołuje konstruktor klasy `QMainWindow`, po której dziedziczy:

```
QtWidgets.QMainWindow.__init__(self).
```

Dzieje się to po to, żeby przekazać klasie nadrzędnej uchwyt do samego siebie (`self`). W języku Python istnieje jeszcze jeden sposób, aby tego dokonać. Służy do tego słowo kluczowe `super`. W takim przypadku pierwsza linijka konstruktora klasy

`UIControler` wyglądałaby następująco:

```
super(UIControler, self).__init__().
```

W konstruktorze inicjalizowane są pola do API backendu (`chamberControler`) oraz do klas obsługujących wykres (`chart` i `chartView`). Linijka

```
self.ui.verticalLayout.addWidget(self.chartView)
```

konstruktora klasy `UIControler` jest przykładem, jak można (dynamicznie) dodawać widgety do layoutów (czyli obiektów odpowiedzialnych za ułożenie widgetów w GUI). Linijka

```
self.ui.pushButton_send.clicked.connect(self.onPushButton_sendClicked)
```

jest przykładem połączenia sygnału `clicked` obiektu reprezentującego przycisk w GUI ze slotem będącym metodą klasy `UIControler`. W linijce

```
QShortcut(QKeySequence('Ctrl+Q'), self).activated
        .connect(QApplication.instance().quit)
```

widnieje przykład wykorzystania sekwencji klawiszy jako skrótu klawiaturowego w celu zamknięcia okna aplikacji.

Poniżej przedstawiono opis metod klasy`UIControler`.

`updateCommandList()` pobiera wczytane komendy i ich nazwami wypełnia listę w GUI.

`setVisibleUserInput()` pokazuje lub chowa kontrolkę przyjmującą wartości użytkownika w zależności, czy wybrana komenda wymaga wprowadzenia wartości.

`setVisibleChartControls()` pokazuje lub chowa kontrolki sterujące wykresem w zależności, czy zakładka z wykresem jest wybrana.

`logError(errorMsg : str)` służy do wyświetlania błędów aplikacji, np. brak połączenia z komorą. Jako argument łańcuch znaków i wyświetla go w oknie przeznaczonym do wyświetlania użytkownikowi logów. Wyświetlony tekst ma kolor czerwony. Komunikat błędu pojawia się również w dolnym pasku statusu.

`logMsg(msg : str)` wyświetla komunikaty w oknie logów i pasku statusu. W przeciwieństwie do poprzedniej funkcji, kolor wyświetlanego tekstu jest czarny.

`onCommandListItemClicked()` jest slotem i reaguje na kliknięcie w komendę zawartą w liście komend. Informuje program o wybranej komendzie i odpowiednio przygotowuje GUI: ukrywa lub pokazuje kontrolkę wprowadzania wartości oraz wyświetla opis komendy w pasku statusu. Pasek znajduje się na samym dole okna aplikacji.

`onPushButton_sendCommandClicked()` to slot, który jest połączony ze zdarzeniem kliknięcia w przycisk „Send Command”. Aktywowana zbiera informacje o ewentualnym wprowadzeniu danych przez użytkownika, a następnie komunikuje się z backendem, który wysyła wybraną komendę i zwraca odpowiedź komory. Odpowiedź ta w tej samej funkcji wyświetlana jest w oknie dla niej przeznaczonym.

`onPushButtonClicked_login()` jest slotem, który odpowiada za proces logowania użytkownika. Wyświetla okno (obiekt klasy `UI_Login`) umożliwiające wpisanie loginu oraz hasła. Następnie dane te przekazuje do backendu, aby nawiązać autoryzowane połączenie z serwerem.

`onCheckConnectionToggled()` to slot połączony z sygnałem uruchomienia przełącznika o nazwie „Check connection" znajdującego się w zakładce „File" w górnym pasku okna. Wyświetla komunikat z informacją, czy połączenie z serwerem i z komorą jest aktywne.

`onCloseConnectionToggled()` to slot, który zamyka połączenie z serwerem.

`onMouseOnChartHover()` jest wołana przez każde zdarzenie najechania kursorem na wykres. Efektem jej działania jest odebranie aktualnej pozycji myszy na wykresie i wyświetlenie go na żywo w pasku bocznym programu. Celem tej operacji jest poinformowanie użytkownika o dokładnej pozycji kursora podczas tworzenia punktów na wykresie.

`onPointAdded()` jest wywoływana podczas dodawania punktu na wykres. Wyświetla różnicę iksowych współrzędnych nowego i przedostatniego punktu celem poinformowania użytkownika o czasie trwania zdarzenia, które naniósł na wykres.

`onChartSendButtonClicked()` jest slotem dla przycisku „Send" będącego częścią interfejsu wykresu. Pobiera z wykresu wygenerowany skrypt, który jest zbiorem komend dla komory zgodnych z utworzonym przez użytkownika wykresem.

`onImportXmlToggled()` to slot, który aktywowany jest przez wybranie przełącznika „Import .xml config file". Umożliwia on wczytanie własnego pliku konfiguracyjnego XML.

`onSaveScript()` umożliwia zapis skryptu do pliku .csv (ang. *comma separated value*).

`onOpenScript()` umożliwia odczyt skryptu z pliku .csv.

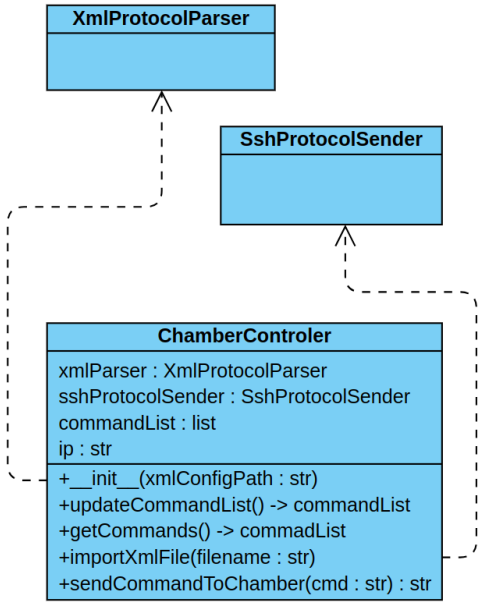
```
1 def setUpWindow():
2     app = QtWidgets.QApplication(sys.argv)
3     uiC = UIControler()
4     uiC.show()
5     uiC.updateCommandList()
6     sys.exit(app.exec_())
```

Listing 8: Funkcja uruchamiająca całą aplikację znajdująca się w pliku `UIControler.py`

W pliku `UIControler.py` oprócz klasy `UIControler` znajduje się również funkcja `setUpWindow()` przedstawiona na listingu 8. Wywoływana jest bezpośrednio w pliku `main.py`. Jej zadaniem jest uruchomienie całej aplikacji. Zdarzenie to ma miejsce konkretnie w linijce 6. Znak podkreślenia na końcu metody `exec_()` jest obecny dlatego, że początkowo Qt było pisane dla Pythona 2.x. Obecnie jest także dostępna wersja tej funkcji bez znaku podkreślenia.

Klasa ChartView

Rysunek 13. demonstruje schemat klasy `ChartView`. Klasa ta powstała głównie w celu przeciążenia metod klasy `QChartView` odpowiedzialnych za zdarzenia myszy oraz klawiatury. Z tego powodu `ChartView` dziedziczy po `QChartView`. Jej konstruktor zawiera uchwyty do głównego okna (aby móc przesłać współrzędne kursora) oraz do klasy `Chart` (żeby utworzyć punkt po kliknięciu prawego przycisku myszy).



Rysunek 13: Diagram UML przedstawiający klasę `ChartView`

Każda metoda tej klasy przeciąża metody obsługujące zdarzenia. Zdarzenie, które autor chciał przeciążyć potrzebuje również jego obsługi pod kątem innych rodzajów tego zdarzenia. Z tego względu każda z metod wywołuje jej poprzednika z klasy przeciążonej i zwraca jego wartość, np. w przypadku `mouseMoveEvent(event: QtGui.QMouseEvent)`:

```
return super().mouseMoveEvent(event).
```

Metoda `mouseMoveEvent()` wczytuje bieżącą pozycję kursora w oknie aplikacji, mapuje go na współrzędne wykresu i zapisuje do zmiennej będącej polem klasy. Zaraz po tym informuje klasę `UIControler` o zmianie pozycji kursora, a ta wyświetla tę informację użytkownikowi.

Funkcja `mousePressEvent()` wykrywa wciśnięcie prawego przycisku myszy na wykresie powodując rysowanie punktu.

Funkcja `wheelEvent()` odpowiedzialna jest za przesuwanie oraz przybliżanie wykresu. Obie te operacje wykonują się jedynie względem poziomej osi czasu, gdyż wartości na pionowej osi nie powinny być zmieniane. Rodzaj wykonywanej operacji zależy od tego, czy użytkownik przytrzymuje klawisz `ctrl`. Sterowanie klawiszem `ctrl` zaimplementowane jest w funkcjach `keyPressEvent()` oraz `keyReleaseEvent()`.

Klasa Chart

Klasa `Chart` jest implementacją wykresu, który służy do tworzenia skryptów dla komory. Jej schemat przedstawiono na rysunku 14. Bazuje na klasie `QChart`, po której dziedziczy.

Konstruktor klasy `Chart` podobnie jak w poprzednich przypadkach rozpoczyna swoją pracę od wywołania konstruktora klasy nadrzędnej. Przechowuje też uchwyt `self.parent` do okna głównego aplikacji, aby móc mu przekazywać informację o odstępie czasowym dwóch ostatnio utworzonych punktów.

Tworzone przez użytkownika punkty przechowywane są w listach `tempPoints` oraz `humidPoints`. Pole `tempPoints` jest kontenerem na punkty ustawiające temperaturę komory, natomiast pole `humidPoints` przechowuje punkty zadające wilgotność powietrza. Parametr `scatterEnable` posiada informację, czy użytkownik chce, aby na wykresie każdy punkt był przedstawiony za pomocą dodatkowej kropki. Zmienna `isHumid` determinuje rysowanie wilgotności, gdy ma wartość `True`, lub rysowanie temperatury, gdy przyjmie wartość `False`.

Rola konstruktora klasy `Chart` nie skończyła się jeszcze, ponieważ zajmuje się również ustawieniem parametrów wykresu. Instancjami reprezentującymi punkty

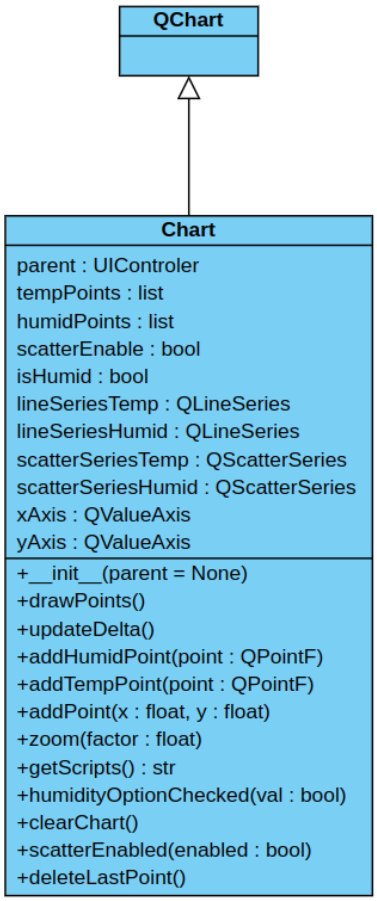
temperatury i wilgotności bezpośrednio na widocznym w aplikacji wykresie są obiekty klas `QLineSeries` (rysowanie prostych) oraz `QScatterSeries` (rysowanie punktów).

Kolejnymi elementami wykresu, które należało skonfigurować, są osie. Po stworzeniu obiektu osi należy go dodać do wykresu oraz do „serii" zawierających punkty rysowane na wykresie.

Idea działania metody `drawPoints()`, której zadaniem jest rysowanie punktów na wykresie została przedstawiona na listingu 9. Najpierw odbywa się czyszczenie okna wykresu z nieaktualnych w danej iteracji punktów za pomocą metody `clear()` (linijki 2. i 4.).

```
1  def drawPoints(self):
2      self.lineSeriesTemp.clear()
3      ⋮
4      self.scatterSeriesTemp.clear()
5      ⋮
6
7      for i in self.tempPoints:
8          self.lineSeriesTemp << i
9          if self.scatterEnable:
10             self.scatterSeriesTemp << i
11     ⋮
12     self.update()
```

Listing 9: Część implementacji metody `drawPoint()`, która wyświetla punkty na wykresie



Rysunek 14: Diagram UML klasy `Chart`

Następne odpowiednie „serie" są aktualizowane nowymi punktami przechowywanymi w listach klasy (linijki od 7. do 11.). „Serie" wypełnia się za pomocą operatora przesunięcia bitowego w lewo `<<`. W ostatniej linijce funkcji wołana jest metoda `update()`, której zadaniem jest odświeżenie okna wykresu. W efekcie użytkownikowi wyświetlają się aktualne punkty.

Kolejną metodą klasy `Chart` jest metoda `updateDelta()`, którą prezentuje listing 10. Oblicza ona odległość czasową dwóch ostatnich punktów na wykresie. Wartość ta roboczo została nazwana `delta`. W 2. linijce funkcji inicjalizowana jest zmienna `delta` wartością -1. Klasa `UIControler` po otrzymaniu wartości delty sprawdza, czy jest ona nieujemna i wyświetla ją tylko wtedy. Po sprawdzeniu, czy w liście znajduje się więcej niż 1 punkt (instrukcja warunkowa w linijce 3.), wartość delty oblicza się w linijce 5. Autor skorzystał tutaj z odniesienia do ostatnich elementów listy używając ujemnych liczb w kwadratowych nawiasach. W tej samej instrukcji warunkowej sprawdzany jest parametr `isHumid`, dzięki któremu `delta` liczona jest dla wilgotności, jeśli użytkownik tworzy wykres wilgotności, lub temperatury, jeśli użytkownik rysuje wykres temperatury. Na końcu metoda informuje klasę `UIControler` o nowej wartości delty, aby ta wyświetliła ją użytkownikowi.

```
def updateDelta(self):
    delta = -1
    if self.isHumid and len(self.humidPoints) > 1:
        delta = abs(self.humidPoints[-1].x()
                    - self.humidPoints[-2].x())
        ⋮
    self.parent.onPointAdded(delta)
```

Listing 10: Część implementacji metody `updateDelta()`, zadaniem której jest obliczenie różnicy czasu dwóch ostatnich punktów

Listing 11. Przedstawia implementację metody `addHumidPoint()`. Jej zadaniem jest sprawdzenie i ewentualne poprawienie punktu wilgotności dodawanego przez użytkownika, a następnie zapisanie go do odpowiedniej listy. Parametr `HOR_TOL` w linijce 2. służy ułatwieniu rysowania poziomych linii. Jest to ważna cecha rysowania wykresów generujących skrypty dla komory klimatycznej, ponieważ użytkownikom często zależy na utrzymywaniu danego parametru stałego przez konkretny okres czasu. Bez tej funkcjonalności bardzo trudno byłoby narysować za pomocą myszy dwa punkty, przez które przechodzi prosta równoległa do poziomej osi czasu. Funkcjonalność ta zrealizowana jest w linijce 4. i 5. Instrukcja warunkowa w linijce 6. sprawdza, czy rysowany przez użytkownika wykres spełnia matematyczne warunki funkcji, czyli czy jest zachowane następstwo czasowe. W linijce 8. program dodatkowo zapewnia, że użytkownik nie zada ujemnej wartości wilgotności. Po sprawdzeniu wszystkich warunków punkt jest dodawany do przeznaczonego dla niego kontenera (linijka 10.), a wykres zostaje zaktualizowany (linijka 11.).

```
def addHumidPoint(self, point : QtCore.QPointF):
    HOR_TOL = 2
    if len(self.humidPoints) > 0:
        if abs(self.humidPoints[-1].y() - point.y()) <= HOR_TOL:
            point.setY(self.humidPoints[-1].y())
        if point.x() <= self.humidPoints[-1].x():
            point.setX(self.humidPoints[-1].x() + 0.1)
    if point.y() < 0.:
        point.setY(0.)
    self.humidPoints.append(point)
    self.drawPoints()
```

Listing 11: Realizacja metody `addHumidPoint()`, której zadaniem jest przetworzenie i dodanie punktu wilgotności do listy

W tym akapicie omówione zostaną pozostałe metody klasy `Chart` (rysunek 14.). Metoda `addTempPoint()` działa identycznie, jak omówiona wyżej `addHumidPoint()`, zapisując punkty temperatury. Funkcja `addPoint()` dodaje punkt korzystając z dwóch poprzednich metod. Funkcja `zoom()` jest przeciążeniem jej poprzednika z klasy `QChart`. Przybliża lub oddala wykres nie zmieniając przy tym pionowej osi. Wołana jest podczas obsługi zdarzenia przewijania myszą z poziomu klasy `ChartView`. Funkcja `getScripts()` generuje skrypt na podstawie wykresu narysowanego przez użytkownika. Sposób generowania skryptu polega na obliczaniu czasu trwania określonej akcji (pozioma oś czasu) oraz obliczaniu współczynnika nachylenia prostej a (wzór (1)). Wzór na współczynnik a uzyskano wykonując bardzo proste przekształcenia po skorzystaniu ze wzoru na tangens w trójkącie prostokątnym. Jeśli zdefiniuje się punkty $A(x_A, y_A)$ oraz $B(x_B, y_B)$ takie, że $x_A < x_B$, to wzór na współczynnik

nachylenia przyjmuje postać widoczną na wzorze (2).

$$y = ax + b \tag{1}$$

$$a = \frac{y_B - y_A}{x_B - x_A} \tag{2}$$

Metoda `humidityOptionChecked()` jest slotem i zmienia opcję rysowania na rysowanie wilgotności lub rysowanie temperatury. Warto zauważyć, że ten slot przyjmuje wartość `bool` - Qt umożliwia przekazywanie argumentów za pomocą sygnałów i slotów. Funkcja `clearChart()` czyści wykres i nadaje stan domyślny atrybutom wykresu. Slot `scatterEnabled()` odbiera sygnał kontrolki GUI i informuje klasę, czy użytkownik życzy sobie rysowanie punktów w formie kropek na wykresie. Ostatnią funkcją klasy `Chart` jest slot `deleteLastPoint()`. Usuwa on ostatni punkt temperatury lub wilgotności w zależności od wybranej opcji. Korzysta z metody `pop()` pythonowej klasy `list`.

Testy jednostkowe

Przetestowano klasę `XmlProtocolParser`, która parsuje plik konfiguracyjny XML. Zdecydowano się na testy jednostkowe, ponieważ w prosty sposób umożliwiają one przetestowanie działania wszystkich komend osobno. Dzięki temu autor mógł odnaleźć błędy zarówno w algorytmie parsowania pliku konfiguracyjnego XML, jak i w samym pliku XML.

Do testów wykorzystano framework o nazwie unittest. Przy wyborze odpowiedniego frameworku kierowano się jego obiektowością oraz dostępnością oficjalnej dokumentacji ze strony python.org [20].

W celu przeprowadzenia testów skonstruowano klasę `TestXmlParser`, która dziedziczy po `unittest.TestCase`. Skorzystano z metody `setUpClass()`, która wywołana jest przed wykonaniem wszystkich testów. Jej zawartość przedstawiono na listingu 12. Zadaniem tej metody jest stworzenie obiektu klasy `XmlProtocolParser` oraz zaimportowanie komend, aby później je przetestować.

```
⋮
@classmethod
def setUpClass(self):
    self.xmlParser = XmlProtocolParser()
    self.commands = self.xmlParser.importCommandsFromXml()
⋮
```

Listing 12: Metoda setUpClass() wykonywana przed testami jednostkowymi w frameworku unittest

Dostępne są również metody:

- `tearDownClass()` - wywoływana po wykonaniu testów,
- `setUp()` - wywoływana przed każdym testem,
- `tearDown()` - wywoływana po każdym teście.

Testy przeprowadza się za pomocą asercji, czyli wyrażenia zwracającego wartość `True` (prawda), gdy test się powiedzie, lub wartość `False` (fałsz) w przypadku przeciwnym. W tym fragmencie pracy skupiono się na testowaniu łańcuchów znakowych, które wygenerowane przez aplikację trafiają do komory przez Ethernet. Dlatego skorzystano z metody `assertEqual()` klasy `unittest.TestCase`. Funkcja ta przyjmuje dwa argumenty i zwraca prawdę, jeśli ich wartości są równe, lub fałsz, jeśli nie są.

Testy napisano w oparciu o dokumentację komory, która jest załącznikiem projektu.

```
1 def test_set_humidity(self):
2     ⋮
3     self.xmlParser.getCommand("set_humidity").setValue("50.00")
4     self.assertEqual(
5       self.xmlParser.getCommand("set_humidity").prepareRequest(),
6       "a1␣50.00"
7     )
8     ⋮
```

Listing 13: Metoda testująca zadawanie gradientu temperatur wykonana przy pomocy frameworku unittest

Każdy test w unittest jest implementowany przez napisanie odpowiedniej metody. Listing 13. przedstawia fragment implementacji jednej z takich metod dla testowania ustawiania wilgotności. W pierwszym kroku (linijka 3.) ustawiana jest żądana wartość, a w następnym kroku (linijka 4.) weryfikowane jest, czy aplikacja wygenerowała właściwy (zgodny z dokumentacją komory) łańcuch znaków. W przypadku chęci ustawienia wilgotności na wartość 50%, program powinien wysłać łańcuch znaków w postaci: `"a1␣50.00"`[19].

Aby pokryć wszystkie możliwe przypadki, testy napisano dla każdej komendy uwzględniając wartości ze środka możliwego zakresu, krawędzi zakresu i spoza zakresu.

Testy unittest uruchamia się za pomocą wiersza poleceń. W celu wywołania modułu unittest należy uruchomić interpreter python z flagą „`-m`” podać nazwę modułu oraz nazwę pliku z testami. W przypadku tego projektu polecenie wyglądało następująco:

`python -m unittest -v testXmlProtocolParser.`[20]

Flaga „`-v`” (od ang. *verbose*, co znaczy gadatliwy) służy do włączenia opcji wyświetlania i opisania wyników testów. Taki wynik został pokazany na listingu 14. Po korekcjach wykrytych w ten sposób błędów, naprawiono je i uzyskano pozytywne wyniki wszystkich testów. Dzięki temu autor zmniejszył zbiór potencjalnych bugów (czyli błędów oprogramowania), które najpewniej ujawniłyby się w trakcie użytkowania programu.

```
1 test_continue_chamber (testXmlProtocolParser.TestXmlParser) ... ok
2 test_pause_chamber (testXmlProtocolParser.TestXmlParser) ... ok
3 test_set_temperature (testXmlProtocolParser.TestXmlParser) ... ok
4 ⋮
5 test_switch_off_chamber (testXmlProtocolParser.TestXmlParser) ... ok
6 test_switch_on_chamber (testXmlProtocolParser.TestXmlParser) ... ok
```

[19] Umieszczenie cudzysłowu w celu przedstawienia łańcucha znaków, nie cytowania.
[20] Obecność kropki wynika z końca zdania, nie jest to część komendy.

```
7
8  ----------------------------------------------------------------------
9  Ran 22 tests in 0.001s
10
11 OK
```

Listing 14: Skrócony wynik uruchomienia testów klasy `XmlProtocolParser`

Nieczęsto jednak zdarza się, że programista za pierwszym razem napisze kod, który jest pozbawiony błędów. Przykład odpowiedzi programu unittest na negatywny test umieszczono na listingu 15.

```
1  test_read_temperature_ramp_parameters (testXmlProtocolParser.
       TestXmlParser) ... ok
2  ⋮
3  test_read_time (testXmlProtocolParser.TestXmlParser) ... FAIL
4  ⋮
5  test_switch_off_chamber (testXmlProtocolParser.TestXmlParser) ... ok
6  test_switch_on_chamber (testXmlProtocolParser.TestXmlParser) ... ok
7
8  ======================================================================
9  FAIL: test_read_time (testXmlProtocolParser.TestXmlParser)
10 ----------------------------------------------------------------------
11 Traceback (most recent call last):
12   File "PC/testXmlProtocolParser.py", line 16, in test_read_time
13     self.assertEqual(self.xmlParser.getCommand("read_time").
       prepareRequest(), "T")
14 AssertionError: 't' != 'T'
15 - t
16 + T
17
18
19 ----------------------------------------------------------------------
20 Ran 22 tests in 0.002s
21
22 FAILED (failures=1)
```

Listing 15: Skrócony wynik testów zakończonych niepowodzeniem

W przypadku braku powodzenia któregoś z testów, unittest poinformuje testera o tym, który z nich się nie powiódł (linijka 3. oraz 9.). Wyrzuci odpowiedni wyjątek (ang. *exception*[21]), dzięki któremu tester może rozpoznać rodzaj błędu (linijka 14.). Wyświetli również informację o spodziewanej i otrzymanej wartości.

3.3 Implementacja aplikacji serwerowej

Połączenie z siecią bezprzewodową

Pierwszym problemem po zainstalowaniu i skonfigurowaniu Ubuntu Server 22.04 LTS było podłączenie urządzenia do sieci bezprzewodowej WiFi oraz nadanie mu stałego adresu IP. Do skonfigurowania WiFi użyto `nmcli`, który jest narzędziem z CLI do zarządzania siecią [22]. Okazało się, że sieć bezprzewodowa w tym przypadku nie zawsze jest stabilna. Gdy doszło do zerwania połączenia, serwer nie nawiązał go ponownie. Autor nie był w stanie ustalić przyczyn. W związku z tym napisał prosty skrypt w języku bash, który przedstawiono na listingu 16.

[21] Wyjątki to mechanizm wykorzystywany w programowaniu do sygnalizowania błędów. Źródło: [21]

```
1  #!/bin/bash
2  ⋮
3  date >> $(pwd)/wifi_log.txt
4
5  if [ ping -q -c 1 8.8.8.8 &>/dev/null ]; then
6      printf "The␣Internect␣connection␣is␣not␣active.\nReseting...\n" >>
       $(pwd)/wifi_log.txt
7
8      nmcli networking off
9      nmcli networking on
10 else
11     printf "Internet␣connection␣is␣active.\n" >> $(pwd)/wifi_log.txt
12 fi
```

Listing 16: Skrypt zajmujący się sprawdzaniem i restartowaniem połączenia WiFi

1. linijka skryptu jest konieczna w przypadku skryptów w języku bash, aby poinformować powłokę, jakiego interpretera użyć do jego uruchomienia. W linijce 3. za pomocą operatora przekierowania strumienia wyjścia `>>` do pliku wifi_log.txt zapisywana jest informacja o dacie uruchomienia skryptu. `$(pwd)` to zmienna przechowująca ścieżkę bezwzględną do katalogu roboczego. Instrukcja warunkowa w linijce 5. wykonuje polecenie `ping` na serwerze Google i sprawdza, czy połączenie jest aktywne. Flaga `-q` wymusza brak wypisania wyniku komendy (ang. *quiet*). Wypisze się jedynie podsumowanie. Flaga `-c` oraz następująca po niej liczba 1 powoduje pojedyncze wykonanie polecenia `ping`. Adres 8.8.8.8 to IP serwera Google. `&> /dev/null` przekierowuje strumień wyjścia i strumień błędu do pliku null, aby się go pozbyć.

```
1  Thu Sep  1 03:05:01 PM UTC 2022
2  Internet connection is active.
3  ************************************************************************
4  Thu Sep  1 03:10:01 PM UTC 2022
5  The Internect connection is not active.
6  Reseting...
```

Listing 17: Fragment pliku wifi_log.txt

Jeśli ping w linijce 5. zwróci wartość `false` to sieć resetuje się. W przeciwnym wypadku zapisuje jedynie informację o prawidłowym połączeniu. Fragment pliku wifi_log.txt przedstawiono na listingu 17.

Skrypt z listingu 1. uruchamiany jest cyklicznie za pomocą demona cron. Aby dostać się do pliku crontab należy wykonać polecenie

`sudo crontab -e`.

Obecność `sudo` oznacza wykonanie komendy jako administrator. W pliku cron umieszczono linijkę

`*/5 * * * * /wifi_task/wifitask.sh`,

która uruchamia skrypt `wifitask.sh` cyklicznie co 5 minut.

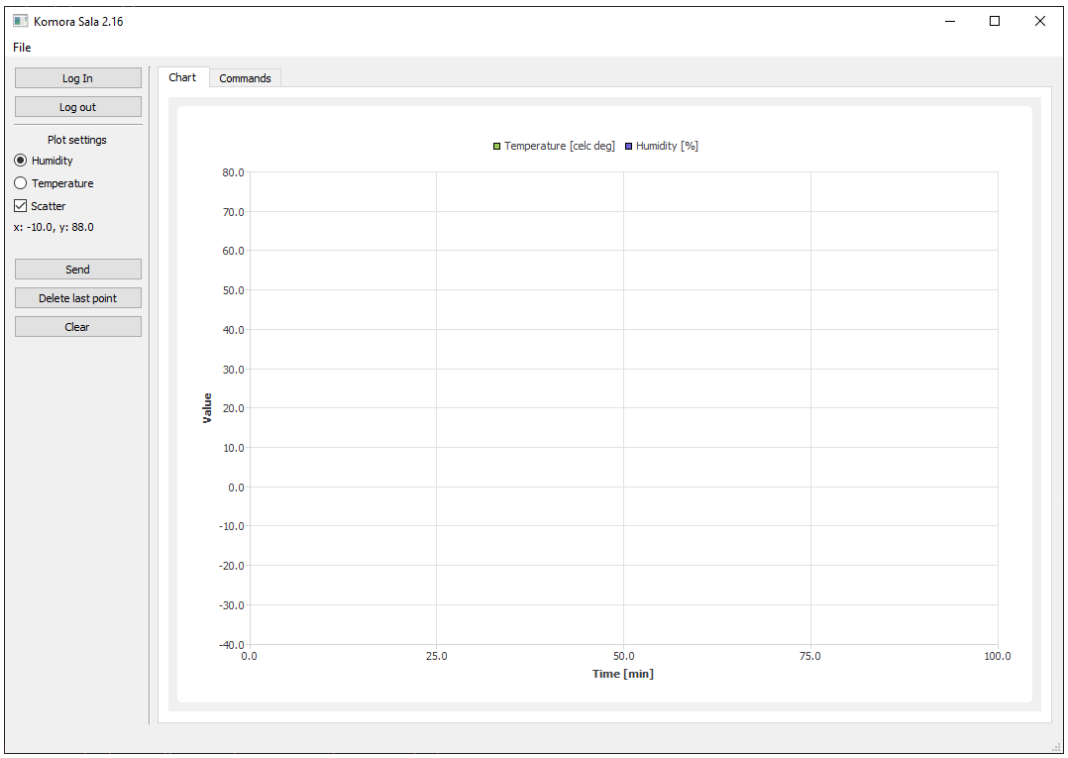
Komunikacja z komorą

Gdy aplikacja użytkownika wyśle komendę lub skrypt do serwera, to zostanie on obsłużony i przekazany do komory. Stanie się to za pomocą protokołu Ethernet i biblioteki *socket*, które są opisane w sekcji Ethernet na stronie 7.

Skrypt do wysyłania komend do komory zawarty jest w pliku `chamber.py`. Zawiera on informacje o adresie IP komory oraz o porcie. Wysyła komendę, a następnie wypisuje odpowiedź komory, aby mogła zostać odczytana przez aplikację użytkownika. W dodatku za pomocą polecenia `with open()` z opcją `"a"` (ang. *append*) do pliku `command_script_history.txt` zapisywane są wysłane komendy, otrzymane odpowiedzi oraz data i godzina tych zdarzeń.

Skrypt pythonowy do wysyłania skryptów do komory mieści się w pliku `chamber_script.py`. Jest rozszerzeniem poprzedniego skryptu o funkcje `send(val : str)` i `sleep(mins : float)`. Pierwsza z nich wysyła pojedynczą komendę do komory, a druga odczekuje zadaną liczbę minut. Z wywołań obu tych funkcji składa się sparsowany skrypt, który utworzony został przez aplikację użytkownika na podstawie wykresu.

4 Działanie systemu



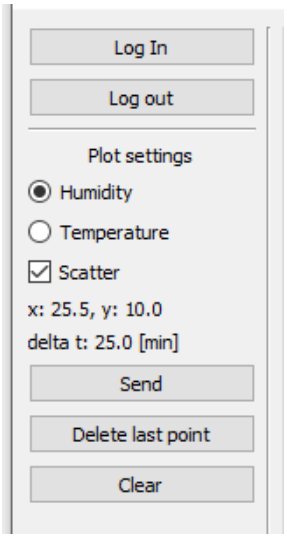
Rysunek 15: Ekran startowy aplikacji

Aplikacja została uruchomiona i przetestowana na dwóch systemach operacyjnych: Windows 10 oraz Ubuntu 22.04 LTS. Działanie aplikacji na obu systemach jest identyczne i nie wymagało praktycznie żadnych istotnych zmian w kodzie źródłowym. Ważna jest jednak wersja używanego języka Python oraz bilbiotek Qt. Używano Pythona w wersji 3.7.9 oraz PyQt5, PySide2 i PyQtChart w wersji 5.15.x. Dla bibliotek PyQt bardzo ważna jest zgodność wersji, ponieważ w innym przypadku często dochodzi do różnych błędów, które są trudne do zlokalizowania i wynikają z różnic w

wersjach bibliotek. Zdecydowano się korzystać z PyQt5 zamiast PyQt6 oraz z Pythona 3.7.9 zamiast Pythona 3.11.x ze względu na mniejszą liczbę bugów w starych wersjach. W korzystaniu z konkretnych wersji podczas rozwijania programu bardzo pomagało wirtualne środowisko *venv*.

Z punktu widzenia użytkownika Windowsa, czy Linuksa jedyną odczuwalną różnicą są nieco inne kontrolki charakterystyczne dla obu systemów. Uruchomienie aplikacji na systemie macOS byłoby również możliwe, ale nie jest przewidywane, aby istniała potrzeba uruchamiania tej aplikacji na tym systemie.

Zakładka Chart



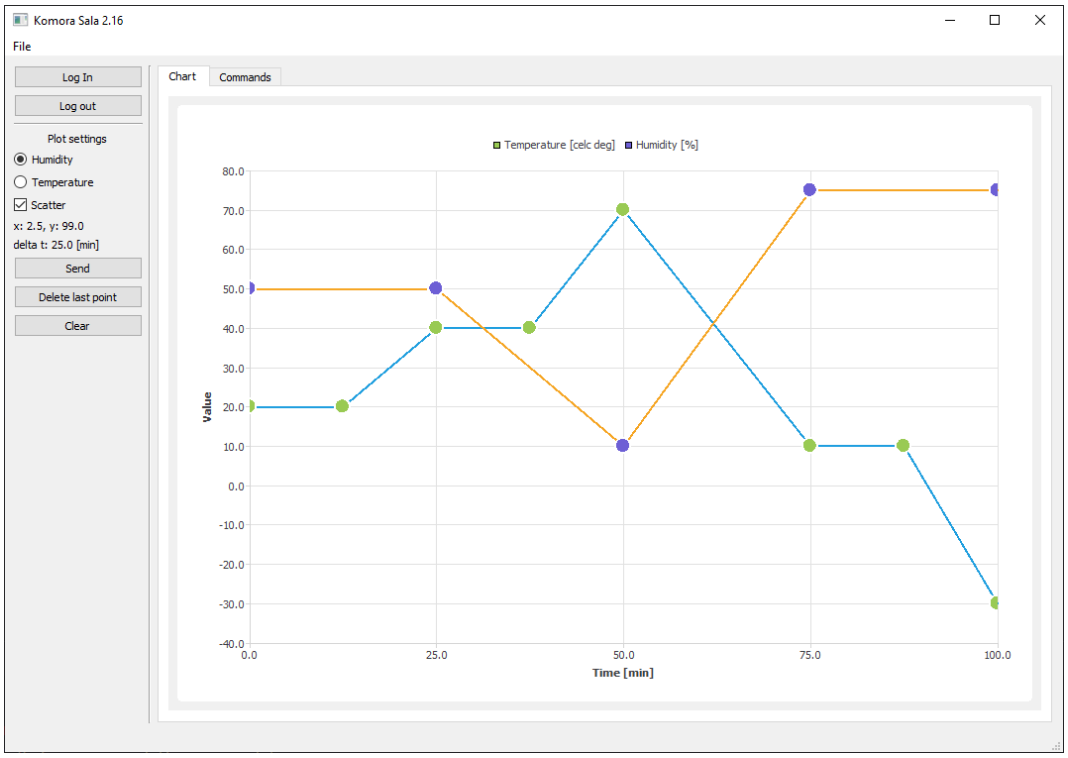
Rysunek 16: Pasek boczny aplikacji w trybie rysowania wykresu

Po uruchomieniu aplikacji ukazuje się jej okno, które zostało przedstawione na rysunku 15. Boczny pasek (rysunek 16.) umożliwia logowanie i wylogowanie użytkownika. Poniżej znajdują się kontrolki do zarządzania wykresem (których nie ma w zakładce *Commands* programu). Pierwsze od góry to przełączniki typu radio. W jednym momencie może być aktywna tylko jedna opcja. W ten sposób użytkownik wybiera, czy edytuje wykres wilgotności, czy temperatury. Kolejną opcją jest przycisk wyboru (ang. *checkbox*). Jego zadaniem jest włączanie lub wyłączanie rysowania punktów za pomocą kropek na wykresie. Skutek tej kontrolki widoczny jest na rysunkach 17 i 18.

Poniżej kontrolki *Scatter* wyświetlane są dwie linijki tekstu. Obie mają za zadanie ułatwienie użytkownikowi dokładnego rysowania wykresu. Pierwsza z nich aktualizuje się podczas każdego ruchu kursorem myszy na wykresie. Współrzędna x wskazuje czas w minutach, a współrzędna y wskazuje wartość wilgotności lub wartość temperatury. Druga linijka tekstu informuje użytkownika o odstępie czasowym pomiędzy dwoma ostatnio narysowanymi przez niego punktami. Dzięki temu użytkownik wie, jaki okres czasu będzie trwała wybrana przez niego akcja.

Ostatnie trzy kontrolki paska bocznego aplikacji z rysunku 16. służą odpowiednio do wgrywania skryptu do komory, usuwania ostatniego punktu z wykresu (w zależności od wyboru edycji wykresu wilgotności albo temperatury) oraz do czyszczenia całego wykresu.

Na rysunku 17. (oraz rysunku 18.) pokazano przykładowy wykres, którego stworzenie umożliwia aplikacja.Wykres ten po sparsowaniu staje się skryptem interpretowalnym dla aplikacji. Skrypt wygenerowany na podstawie tych dwóch rysunków widocznym jest w tabeli 1. Lewa kolumna to wygenerowane na podstawie narysowanego wykresu komendy. Prawa kolumna to komendy sparsowane przez aplikację. W tym przykładzie użytkownik ustawił temperaturę na $20°C$ i zażyczył sobie, aby taka temperatura była utrzymywana przez 12,5 min. Wygenerowało to komendę `set_temperature(20.0)` oraz `sleep(12.5)`. W prawej kolumnie widzimy sparsowaną wersję tych komend, czyli odpowiednio `send("a0 20.00")` oraz `sleep(12.5)`. Funkja `sleep()` nie zmieniła się, a funkcja `set_temperature()` została przetłu-



Rysunek 17: Okno aplikacji z przykładowym wykresem

Rysunek 18: Fragment okna aplikacji przedstawiający wykresy z wyłączoną opcją *Scatter*

maczona na łańcuch znaków `"a0 20.00"`. Jest to łańcuch zgodny z dokumentacją komory wygenerowany na podstawie pliku konfiguracyjnego XML. Takie komendy trafiają z aplikacji użytkownika na serwer ubuntu i tam są obsługiwane przez skrypt opisany na stronie 27.

Po upływie 12,5 min użytkownik ustawił stopniowy wzrostu temperatury. Wzrost ten został obliczony zgodnie ze wzorem (2) ze strony 24. Stąd też została wygenerowana komenda `set_temperature_gradient_up(1.6)`, a następnie przetłumaczona na `send("u1 1.60")`. Gradient ten miał być utrzymany przez kolejne 12,5 min, aby się zatrzymać na $40°C$. Po pomnożeniu wartości gradientu przez jego czas trwania otrzymujemy wzrost temperatury $1{,}6\frac{°C}{\text{min}} * 12{,}5 \text{ min} = 20°C$. Poprzednia temperatura wynosiła $20°C$, więc w rezultacie komora powinna uzyskać temperaturę $40°C$, co jest zgodne z wykresem.

Tok postępowania dla wykresu wilgotności jest analogiczny. Wygenerowane komendy dla wilgotności widoczne są w tabeli 1.

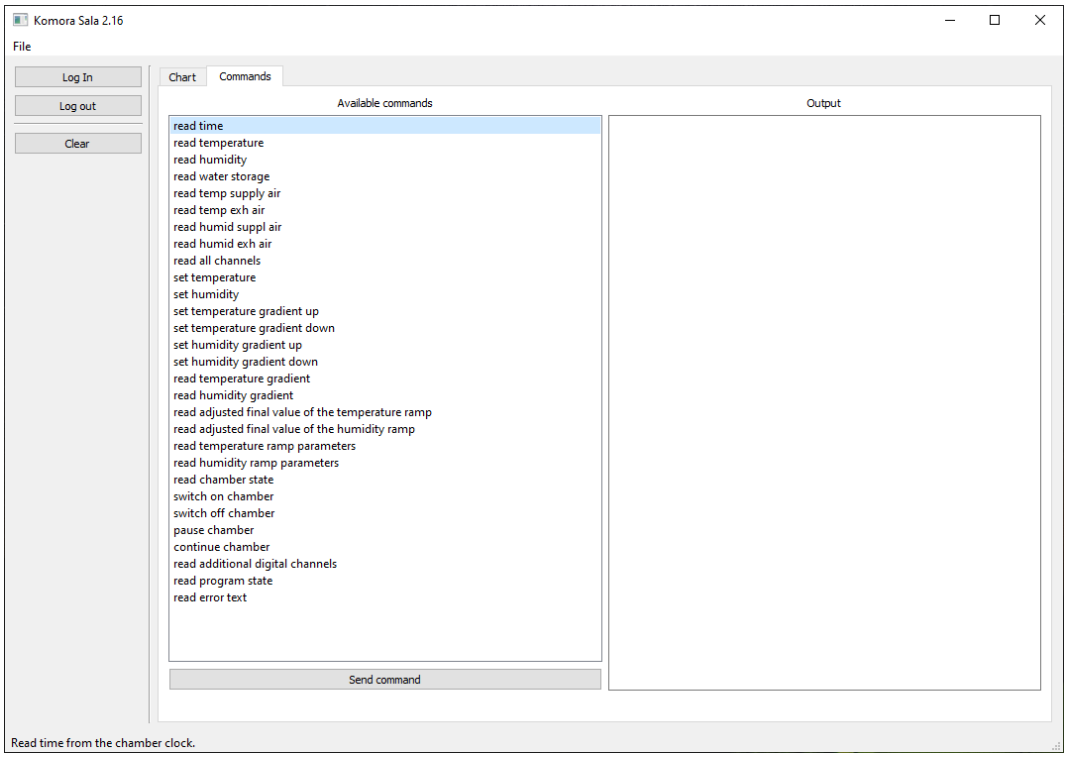
Tabela 1: Komendy wygenerowane na podstawie wykresu z rysunku 17.

Komendy przed parsowaniem	**Komendy po parsowaniu**
Wykres temperatury	
`set_temperature(20.0)`	`send("a0 20.00")`
`sleep(12.5)`	`sleep(12.5)`
`set_temperature_gradient_up(1.6)`	`send("u1 1.60")`
`sleep(12.5)`	`sleep(12.5)`
`set_temperature(40.0)`	`send("a0 40.00")`
`sleep(12.5)`	`sleep(12.5)`
`set_temperature_gradient_up(2.4)`	`send("u1 2.40")`
`sleep(12.5)`	`sleep(12.5)`
`set_temperature_gradient_down(2.4)`	`send("d1 2.40")`
`sleep(25.0)`	`sleep(25.0)`
`set_temperature(10.0)`	`send("a0 10.00")`
`sleep(12.5)`	`sleep(12.5)`
`set_temperature_gradient_down(3.2)`	`send("d1 3.20")`
`sleep(12.5)`	`sleep(12.5)`
Wykres wilgotności	
`set_humidity(50.0)`	`send("a1 50.00")`
`sleep(25.0)`	`sleep(25.0)`
`set_humidity_gradient_down(1.6)`	`send("d2 1.60")`
`sleep(25.0)`	`sleep(25.0)`
`set_humidity_gradient_up(2.6)`	`send("u2 2.60")`
`sleep(25.0)`	`sleep(25.0)`
`set_humidity(75.0)`	`send("a1 75.00")`
`sleep(25.0)`	`sleep(25.0)`

Zakładka Commands

Rysunek 19. przedstawia zakładkę *Commands* okna aplikacji. Zakładka ta jest przeznaczona do odczytywania logów oraz manualnego wysyłania komend do komory.

Jedną ze znaczących zmian w stosunku do zakładki *Chart* jest znacznie uboższy lewy pasek aplikacji, który zawiera jedynie przyciski służące do zalogowania, wylogowania użytkownika oraz oddzielony kreską przycisk służący do czyszczenia okna logów *Output*.

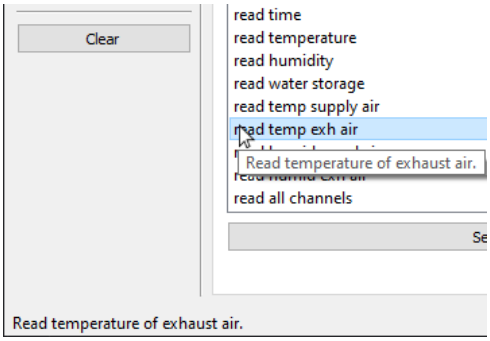


Rysunek 19: Zakładka *Commands* okna aplikacji

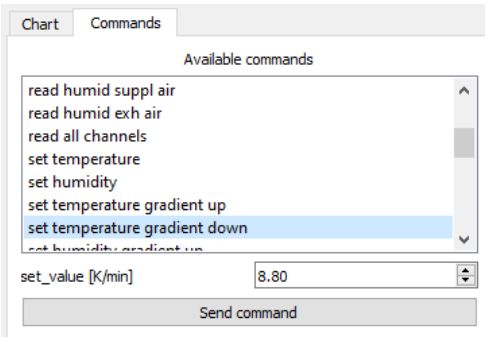
Główna część okna podzielona jest na dwa sektory: lewy o nazwie *Available commands* oraz prawy *Output*. Pierwszy z nich to obiekt klasy `QListWidget`. Jest listą zawierającą komendy wczytane z pliku konfiguracyjnego XML. Drugi z nich to okno logów, które informuje użytkownika o stanie aplikacji, błędach oraz odpowiedziach komory.

W dolnej części okna znajduje się pasek statusu (ang. *status bar*), który wyświetla komunikaty dla użytkownika. W momencie wybrania komendy pojawia się w nim opis komendy, który zostaje wczytany z podwęzła `<description>` węzła `<command>` wypełnioniego w pliku konfiguracyjnym XML. Ten sam opis pokazuje się również po najechaniu kursorem myszy na komendę (ang. *tooltip*), co pokazane jest na rysunku 20.

Istnieją dwa sposoby, aby manualnie wysłać komendę do komory. Pierwszy sposób to podwójne kliknięcie na element listy reprezentujący komendę. Drugim sposobem jest kliknięcie przycisku *Send command*. Gdy komenda wymaga wprowadzenia wartości od użytkownika (komenda typu „*set*"), pod listą komend i nad przyciskiem wysyłania pojawia się przewijane pole numeryczne (ang. *spinbox* lub *spinner*), co pokazane szostało na rysunku 21. Można je edytować za pomocą przewijania kółka myszy po najechaniu kursorem, za pomocą przycisków góra i dół lub manualnie wpisując wartość liczbową w to pole.

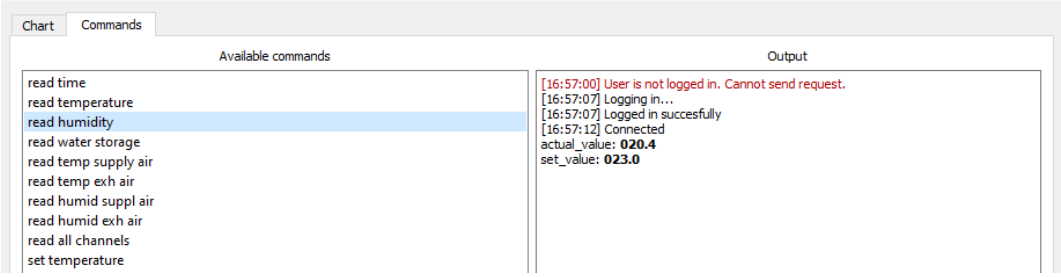


Rysunek 20: Fragment okna aplikacji przedstawiający działanie paska statusu oraz dymka z podpowiedzią



Rysunek 21: Fragment okna aplikacji przedstawiający kontrolki służące do manualnego wysyłania komend

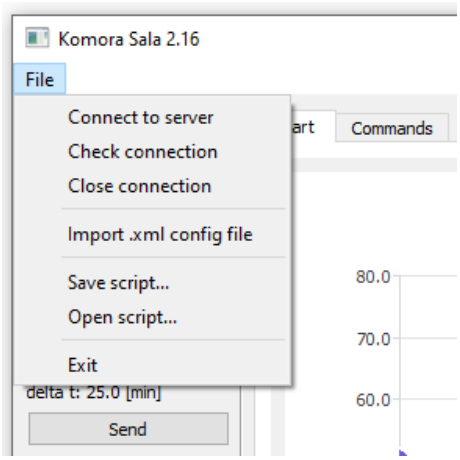
Na rysunku 22. przedstawiono fragment okna aplikacji, który pokazuje sposób wyświetlania logów w oknie *Output*. Logi zawierają godzinę ich pojawienia się, aby zwiększyć ich czytelność oraz poinformować użytkownika o czasie wystąpienia zdarzenia. Błędy są wyświetlane w kolorze czerwonym. Zwykłe informacje mają kolor czarny. Odpowiedzi od komory są natomiast pogrubione. Każdy z logów wyświetla się również w pasku statusu znajdującym się w dolnej części okna na czas 4 sekund.



Rysunek 22: Fragment okna aplikacji przedstawiający okienko do logowania wiadomości

W prawym oknienku *Output* na rysunku 22. pokazany jest przykładowy wynik działania programu w formie logów. Najpierw użytkownik podjął próbę wysłania komendy będąc niezalogowanym. Aplikacja poinformowała go o tym drukując czerwony log. Gdy użytkownik zalogował się za pomocą okna logowania z rysunku 24., został powiadomiony o pomyślnym przebiegu tego procesu, co widoczne jest w kolejnych linijkach logów. Po manualnym wysłaniu komendy `read humidity`, otrzymano odpowiedź od komory o aktualnej wilgotności wewnątrz (20.4%) oraz o wilgotności ustawionej (23.0%).

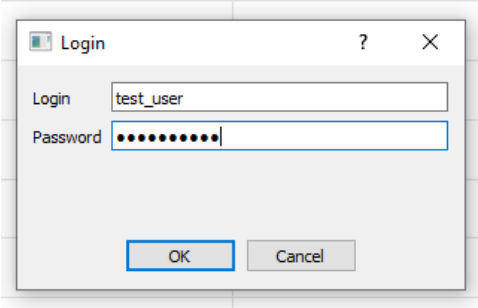
Lista rozwijana File



Rysunek 23: Fragment okna aplikacji z rozwiniętym submenu *File*

W lewym górnym rogu aplikacji znajduje się rozwijane menu *File*. Znajdują się tam następujące opcje:

- *Connect to server* - otwiera okienko logowania użytkownika (rysunek 24.). Zawiera pole do wprowadzenia loginu użytkownika oraz pole do wprowadzania hasła z maskowaniem znaków,
- *Check connection* - sprawdza, czy aplikacja posiada aktywne połączenie z serwerem. Informacja o tym jest wyświetlana w pasku statusu oraz w oknie logów *Output* (przykład na rysunku 22. - linijka o treści `[16:57:12] Connected`),
- *Close connection* - zamyka połączenie z serwerem,
- *Import .xml config file* - otwiera systemowe okno dialogowe, w którym należy wybrać ścieżkę do pliku konfiguracyjnego XML. Jest to opcja używana w momencie, w którym użytkownik chce zaimportować własny plik konfiguracyjny,
- *Save script...* - otwiera systemowe okno dialogowe, w którym należy podać ścieżkę, w której program ma zapisać wykres do pliku z rozszerzeniem .csv,



Rysunek 24: Okienko logowania użytkownika

- *Open script...* - otwiera systemowe okno dialogowe, które umożliwia otwarcie pliku z rozszerzeniem .csv zawierającego zapisany wykres,
- *Exit* - zamyka aplikację. Tę samą akcję można wykonać przez wciśnięcie przycisku „x” w prawym górnym rogu aplikacji lub za pomocą skrótu klawiszowego „ctrl + Q”.

5 Dalszy rozwój

Założenia projektu zostały ustalone z uwzględnieniem ograniczeń czasowych na jego przygotowanie. Dlatego wśród nich zabrakło wielu drobnych aspektów, takich jak np. ergonomia aplikacji użytkownika. W tym rozdziale opisano kilka pomysłów na jej rozwinięcie. Jednym z nich byłoby usprawnienie i uatrakcyjnienie interfejsu graficznego. Jednym z takich usprawnień byłoby np. uruchomienie procesu logowania w osobnym wątku, aby nie blokować GUI na czas logowania. Okno aplikacji powinno mieć tryb ciemny, który jest mniej męczący dla oczu. Przydałaby się również lepsza obsługa wykresu do tworzenia skryptów. Punkty mogłyby dawać się przeciągać lewym przyciskiem myszy - obecnie punkty można jedynie tworzyć w danym punkcie i usunąć w przypadku nietrafienia w pożądane miejsce. Wykres mógłby być przewijany dodatkowo za pomocą przeciągnięcia myszą, a nie tylko za pomocą kółka myszy. Dodatkowo dobrze byłoby, gdyby przybliżanie i oddalanie wykresu odbywało się w stosunku do kursora myszy, a nie środka ekranu. Dobrym pomysłem byłoby umieszczenie pod wykresem tabeli zawierającej wszystkie naniesione punkty. To wygodny sposób na kontrolowanie i dokładne edytowanie wartości punktów. Można by również dodać opcjonalną funkcjonalność śledzenia postępów pracy komory i nanoszenia ich na żywo na wykres skryptowy. Dzięki temu użytkownik mógłby łatwo zweryfikować poprawność przebiegu badań i doświadczeń. Kolejną funkcjonalnością byłaby możliwość sterowania kilkoma komorami z jednej aplikacji. Obecnie wymaga to modyfikacji pliku konfiguracyjnego XML. Znacznie wygodniej byłoby mieć możliwość szybkiego wyboru interesującej nas komory np. z rozwijanej listy w interfejsie graficznym. Przydatnym byłoby też tworzenie konta na podstawie *Active Directory* - tzw. usługi katalogowej od firmy Microsoft, czyli bazy danych zawierającej informacje m. in. o kontach użytkowników [23]. Ostatnia przykładowa funkcjonalność to możliwość rezerwacji komory na określony czas w kalendarzu (np. Microsoft Outlook), co usprawniłoby pracę z komorą w przypadku większej liczby pracowników.

6 Podsumowanie

Autorowi udało się spełnić wszystkie założenia projektu. System zapewnia podstawowe funkcjonalności pozwalające na pracę z komorą klimatyczną. Umożliwia zarówno manualne sterowanie komorą klimatyczną, jak i automatyzację jej pracy za pomocą skryptów. System składa się z aplikacji użytkownika oraz serwera, z którym aplikacja łączy się za pomocą sieci internetowej. Aplikacja jest generyczna dzięki plikowi konfiguracyjnemu XML. W razie potrzeby pracy z inną komorą obsługującą komunikację przez Ethernet, wystarczy zmodyfikować lub napisać odpowiedni plik konfiguracyjny.

Bibliografia

[1] Katalog producenta komór klimatycznych. https://www.cts-umweltsimulation.de/en/products/climate-c.html Dostęp w dniu 9.12.2022 r.

[2] Historia języka Python. https://docs.python.org/3/license.html Dostęp w dniu 9.12.2022 r.

[3] Instrukcja Linux - man ssh. https://linux.die.net/man/1/ssh Dostęp w dniu 6.12.2022 r.

[4] Dokumentacja biblioteki Paramiko. https://docs.paramiko.org/en/stable/api/client.html Dostęp w dniu 06.12.2022 r.

[5] Opis XML w3c. https://www.w3.org/XML/ Dostęp w dniu 6.12.2022 r.

[6] Dokumentacja języka Python - xml.etree.ElementTree. https://docs.python.org/3/library/xml.etree.elementtree.html Dostęp w dniu 6.12.2022 r.

[7] Oficjalna strone RiverbankComputing - PyQt intro. https://riverbankcomputing.com/software/pyqt/intro Dostęp w dniu 9.12.2022 r.

[8] Dokumentacja Qt, rozdział Sygnały i Sloty. https://doc.qt.io/qt-6/signalsandslots.html Dostęp w dniu 9.12.2022 r.

[9] Dokumentacja języka Python, rozdział unittest. https://docs.python.org/3/library/unittest.html

[10] K. Sacha - Inżynieria oprogramowania, s. 124

[11] Przewodnik programu Visual Paradigm. https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/ Dostęp w dniu 9.12.2022 r.

[12] Oficjalna strona internetowa Ubuntu. https://ubuntu.com/desktop Dostęp w dniu 6.12.2022 r.

[13] Instrukcja Ubuntu Manuals - crontab. https://manpages.ubuntu.com/manpages/trusty/man5/crontab.5.html Dostęp w dniu 6.12.2022 r.

[14] Dokumentacja języka Python. https://docs.python.org/3/library/socket.html Dostęp w dniu 6.12.2022 r.

[15] Artykuł "The Meaning of Underscores in Python. https://dbader.org/blog/meaning-of-underscores-in-python Dostęp w dniu 6.12.2022 r.

[16] Jerzy Grębosz: Symfonia C++, Kraków: Edition 2000, 2005, s. 416.

[17] Allen B. Downey: Think Python: How to Think Like a Computer Scientist, 2nd Edition, s. 125

[18] Dokumentacja języka Python. https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions Dostęp w dniu 17.12.2022 r.

[19] Dokumentacja biblioteki PyQt5. https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QMainWindow.html Dostęp w dniu 9.12.2022 r.

[20] Dokumentacja frameworka do testów jednostkowych unittest. https://docs.python.org/3/library/unittest.html Dostęp w dniu 10.12.2022 r.

[21] Allen B. Downey: Think Python: How to Think Like a Computer Scientist, Version 1.1.19, s. 7

[22] Instrukcja Ubuntu Manuals - nmcli. https://manpages.ubuntu.com/manpages/jammy/en/man1/nmcli.1.html Dostęp w dniu 28.12.2022 r.

[23] Dokumentacja usługi Active directory. https://learn.microsoft.com/pl-pl/troubleshoot/windows-server/identity/active-directory-overview Dostęp w dniu 30.12.2022 r.