МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра РЭС

ОТЧЕТ

по учебной практике

Тема: Рефакторинг кода. Проектирование встроенных приложений умного дома

Студент гр. 1181 Шишков Д.А.

Руководитель Проценко И.М.

Санкт-Петербург

2023

ЗАДАНИЕ

НА УЧЕБНУЮ ПРАКТИКУ

Студент Шишков Д.А.
Группа 1181
Тема практики: Рефакторинг кода. Проектирование встроенных приложений умного дома.

Задание на практику:

Произвести рефакторинг (улучшение качества кода без изменения его поведения) графического приложения, написанного на ЯП Python с использованием библиотеки PyQT5. А так же привести теорию неблокирующего взаимодействия и реализовать функциональность выбора ручного/автоматического обновления данных и UI программы.

Сроки прохождения практики: 27.03.2023-02.06.2023

Дата сдачи отчета: 08.03.2023

Дата защиты отчета: 08.03.2023

Студент		Шишков Д.А.
Руководитель		Проценко И.М.

АННОТАЦИЯ

Целью данной работы является развитие профессиональных навыков в понимании и доработке чужого программного кода, написанного на языке программирования Python. При сохранении имеющегося функционала программы, её код был реорганизован, разбит на классы, сетевое взаимодействие переписано с использованием `QNetworkAccessManager`, что сделало его неблокирующим, исправлены баги и реализован недостающий функционал, такой как периодическое обновление данных и чтение начального состояния отображаемых данных из config-файла.

SUMMARY

The purpose of this work is to develop professional skills in understanding and refining other people's code written in the Python programming language. While preserving the existing functionality of the program, its code was reorganized, split into classes, the networking was rewritten using `QNetworkAccessManager`, making it non-blocking, bugs were fixed and missing functionality was implemented, such as periodic data update and reading the initial state of the displayed data from the config-file.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Современные бизнес-процессы предполагают ускоренное прототипирование программных продуктов для ускорения выхода нового функционала на рынок. Однако, в какой-то момент возникает необходимость оптимизировать и улучшить поддерживаемость данного кода. Для отработки подобных задач была организована следующая учебная практика. Был получен исходный код приложения, написанного на Python с применением библиотеки PyQT5 и кода стенда, с которым оно производит взаимодействие по сети. Аппаратно-програмный комплекс разработан Проценко И.М. Для группы из трёх студентов была поставлена задача произвести рефакторинг кода и реализовать недостающий функционал. Автор данного отчёта взял на себя переработку сетевого стека программы, для чего также пришлось внести некоторые архитектурные изменения.

1. ТЕОРИЯ НЕБЛОКИРУЮЩЕГО ВЗАИМОДЕЙСТВИЯ

1.1. Краткая историческая справка

Традиционно, программы, разрабатываемые для первых ЭВМ были лишь последовательностью инструкцией, поочерёдно исполняемых процессором. При этом, даже несмотря на наличие условных переходов, поведение программы полностью определялось входными данными и самим кодом. Таким образом, после запуска оператор никак не мог воздействовать на её работу.

Добавление периферийных устройств ввода, позволило добавить интерактивность, однако, традиционная парадигма разработки программ была неудобна для обработки поступающей информации. Значительным шагом вперёд стало введение операций прерывания. Они позволяют останавливать выполнение основного кода программы и передавать управление отдельным участкам кода, отвечающим за обработку поступающих сигналов. После их завершения выполнение передавалось обратно в основной блок программы.

Однако, если обработка прерываний занимает большое количество циклов процессора, а так же если они случаются слишком часто, выполнение основного кода программы значительно замедляется. Оставляя за рамками данной работы весь путь, пройденный инженерами для разработки и внедрения параллельного выполнения инструкций процессора, а также появление мультипроцессорности, скажем, что для нас важен тот факт, что отход от последовательного выполнения одной программы породил проблему синхронизации разных потоков.

Для её решения были введены новые техники такие, как семафоры, мьютексы. Это усложнило написание программ, так как если в логике взаимодействия потоков были допущены ошибки, они могли приводить к таким ситуациям, как состояние гонки — когда результат зависит от того, в каком

порядке выполнились части кода, или взаимным блокировкам (deadlock), когда у нескольких потоков образуются циклические зависимости от общих ресурсов.

Несмотря на вышеперечисленные минусы, невозможно представить современное достаточно сложное ПО не использующее многозадачность. Это позволило значительно повысить производительность компьютеров и интегрировать их во многих сферах, где раньше казалось, что это невозможно, или нерентабельно.

При этом, блокировки — не единственная существующая парадигма. Для решения проблем синхронизации также были разработаны модель акторов, программная транзакционная память (software transactional memory, STM). Первая подразумевает асинхронный обмен сообщениями между «акторами» - вычислительными сущностями. Вторая аналогична транзакционному механизму в базах данных — когда набор операций применяется над общими данными атомарно.

1.2. Графический пользовательский интерфейс

Одной из технологий, невозможных без многозадачности является графический пользовательский интерфейс (GUI). Действительно, любое взаимодействие пользователя с программой прерывало бы основные вычисления, а так как такие инструменты ввода как мышь практически постоянно генерируют прерывания, на остальные действия совсем не осталось бы машинного времени.

В большинстве современных архитектур логика взаимодействия пользователя с GUI реализуется с помощью так называемого событийного цикла (event loop). Так, главный процесс запускает бесконечный цикл, в каждой итерации которого происходит опрос элементов интерфейса и любых иных сущностей (event providers), которые могут послать сообщение. Его исполнение блокируется пока на вход не поступит новое сообщение. Для полученных сообщений вызываются соответствующие им обработчики. При этом если все доступные потоки исполнения (workers) заняты, то сообщение попадает в

очередь (message queue), откуда будет забрано когда освободится worker. При этом детали реализации, такие как асинхронность выполнения обработчиков, получения сообщений и их приоритизация абстрагируются простым интерфейсом, благодаря которому разработчик может эффективно разрабатывать интерактивные приложения.

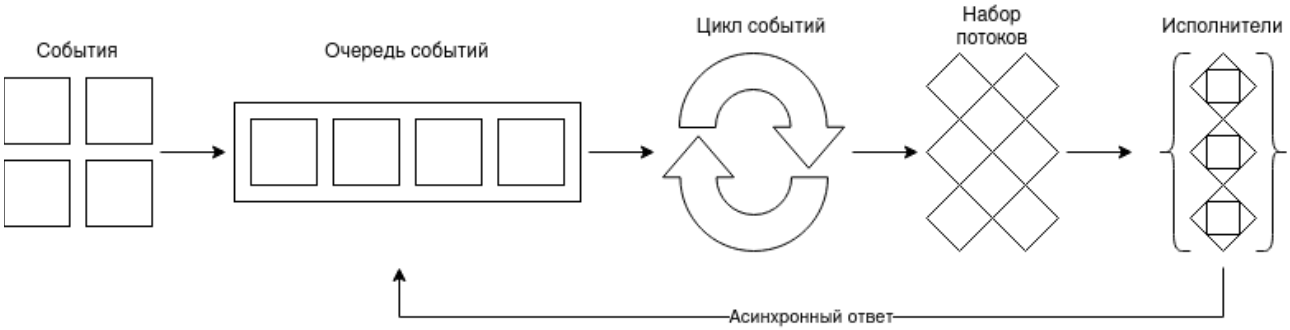

Рис. 1 Цикл событий

На самом деле, описанная конструкция применяется не только при разработке GUI. Так, event loop лёг в основу языка программирования JavaScript и серверной среды его исполнения Node.js в частности, шине системных сообщений D-Bus, где используется GLib.

1.3. Параллелизм в Python

Многопоточность в Python обычно применяется с использованием модуля `threading`. Он позволяет параллельно выполнять требовательные к процессору операции. Однако, из-за известной проблемы наиболее популярной реализации интерпретатора CPython, глобальной блокировки интерпретатора (GIL), делающей невозможной интерпретацию параллельно выполнению кода.

Чаще всего в разрабатываемом на этом языке приложениях необходимо ускорить множественные долгие операции ввода-вывода. Например, сетевое взаимодействие, работа с файловой системой, сигналы операционной системой. Для этих задач применяется модуль `asyncio`, реализующий event loop в Python.

Он предоставляет несколько высокоуровневых сущностей, позволяющих запланировать задачи для неблокирующего выполнения и дождаться их завершения там, где нужен их результат. Корутины (Coroutines) — это

функции, обозначенные ключевым словом `async`. Внутри них, используя ключевое слово `await` можно дожидаться выполнения других корутин. Задачи (Tasks) — это обёртки, используемые для параллельного запуска корутин на event loop.

Для решения типовых задач на базе `asyncio` были разработаны дополнительные модули. Так, для асинхронных сетевых запросов применяется модуль `aiohttp`, а для работы с файлами `aiofiles`.

2. РАЗРАБОТКА СЕТЕВОГО СТЕКА В PYQT5

2.1. QT и библиотека PyQT5

Одним из самых распространённых инструментов для разработки кроссплатформенного GUI является библиотека QT. Она использует парадигму ООП. Легла в основу окружения рабочего стола KDE, портирована на операционные системы Microsoft Windows, множество систем семейства UNIX, Android, Mac OS и другие. Для неё также представлена утилита QT Designer для графической разработки шаблона интерфейса оконного приложения.

Главным языком разработки для QT является C++, чьи возможности дополнены метаобъектным компилятором, для поддержки слотов и сигналов (о них позже). Однако, также существуют биндинги — библиотеки, реализующие интерфейс между языками, для Java, Ruby и т. д. В том числе и для Python — модуль PyQT5. Он позволяет описывать логику на Python, но в сущности программа всё ещё выполняется ядром QT. Поэтому зачастую более эффективным подходом является использование встроенных в QT классов вместо возможностей Python.

2.2. События, сигналы и слоты в QT

Интерфейс приложения описывается древовидной структурой в формате XML, так называемым ui-файлом. В основе QT лежит описанная в предыдущем разделе модель event loop. Пользовательское взаимодействие описывается объектами события, такими как `QMouseEvent`. Сначала событие ассоциируется с самым глубоким элементом в дереве компонентов. Если в объекте, описывающем этот элемент отсутствует обработчик данного события, оно «всплывает» (bubbles) к родительскому объекту и так происходит, пока оно не будет обработано.

Описанный выше подход удобен когда элемент самостоятельно обрабатывает событие. Однако, он неудобен когда обработчик берётся из другого объекта, так как тогда нужно следить за тем, чтобы этот объект не был удалён из памяти. Для решения данной проблемы в QT был введён механизм сигналов и слотов. Каждый объект может в ответ на свои события посылать сигналы и предоставлять обработчики сигналов в виде слотов. Привязка слотов к сигналам осуществляется при помощи внутренних механизмов QT, обеспечивая безопасную с точки зрения работы с памятью абстракцию. Разработчик может как подписываться на встроенные в объекты QT сигналы, так и создавать свои. То же касается слотов.

Действительно полезным данный подход оказывается в многопоточных приложениях, которые в QT реализуются с использованием `QThread`. Благодаря сигналам и слотам разные потоки могут обмениваться между собой данными.

2.3. Сетевое взаимодействие в QT, `QNetworkAccessManager`

Для работы с сетью в QT предусмотрен класс `QNetworkAccessManager`. Он позволяет делать асинхронные запросы, выполняемые в другом потоке и синхронизироваться с ним для получения ответа с помощью механизма слотов и сигналов. HTTP запрос описывается классом `QNetworkRequest`, URL передаётся в его конструктор в виде `QUrl`. Объект, описывающий ответ — `QnetworkReply`, возвращается методами `QNetowrkAccessManager`’а названными соответственно типам HTTP запросов, например, `get` или `post`. Он имеет сигнал `finished`, к которому можно подключить собственный обработчик (слот, функцию). Обычно достаточно одного `QnetowrkAccessManager`’а на приложение, поэтому его можно рассматривать как центральный хаб, оркеструющий все HTTP запросы в приложении.

Всё вышеперечисленное справедливо и для PyQT5, поэтому вместо модулей `requests` или `aiohttp` имеет смысл для соблюдения лучших практик данного стека использовать эту технологию.

3. ОПИСАНИЕ ПРОДЕЛАННОЙ РАБОТЫ

3.1. Переход к классовой структуре приложения

Как было сказано выше, QT предполагает разработку в парадигме ООП (подробнее эта тема раскрыта в отчётах моих коллег), поэтому для консистентности изначальный код был реорганизован как методы класса, наследующегося от `QMainWindow`. Логика работы с графиком при помощи библиотеки `pyqtgraph` была вынесена в отдельный класс `Plot`, экземпляр которого хранится в главном классе в поле `plot`. Объект, хранящий элементы интерфейса помещён в поле `ui`, `QnetowrkAccessManager` в поле `nam`.

3.2. Интерфейс сетевого взаимодействия

На основании оригинального кода, а также C++ кода для стенда была составлена спецификация HTTP API для взаимодействия макета и программы-интерфейса. Её представление в формате Swagger OpenAPI приведено в приложении А. Всего, предполагается два типа HTTP запросов: GET для получения состояния стенда и POST для отправки управляемых параметров на макет. На рис. 2 приведена схема сетевого взаимодействия приложения и стенда.

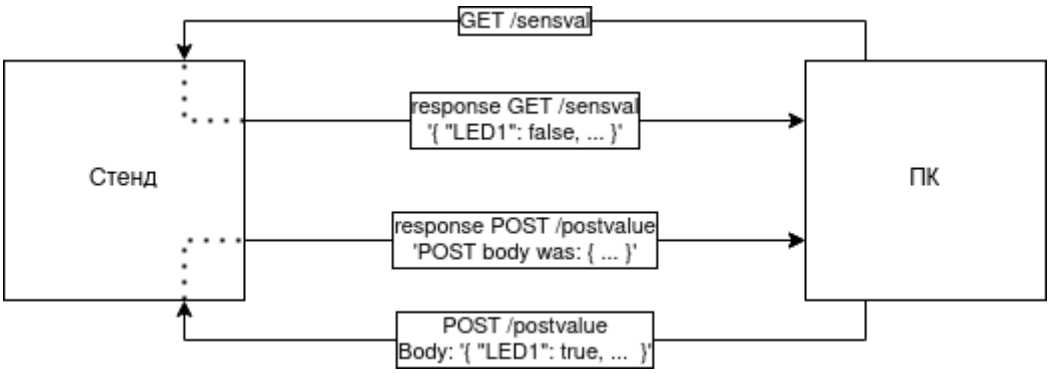

Рис. 2 Схема HTTP запросов

Так как в качестве транспортного протокола был выбран TCP/IP, в программе необходимо указать адрес макета. Это может быть либо IP, либо

доменное имя, которое будет разрешено средствами ОС. Для их ввода в программе предусмотрены два поля ввода — одно для URL (помимо адреса также является конфигурируемым и путь) GET-запроса, другое для URL POST-запроса. Как и в изначальной реализации, GET и POST запросы могут быть вызваны нажатием соответствующих кнопок. Но в добавок к этому, был реализован опциональный функционал автоматического обновления данных. Раз во введённое пользователем количество секунд на стенд посылается GET запрос. А POST запрос отправляется каждый раз когда пользователь меняет состояние макета.

При POST запросе на стенд отправляются булевые значения трёх лампочек и объекты из трёх целых чисел, определяющие цвета восьми RGB светодиодов. При GET запросе макет, помимо этих данных, также посылает булевые значения для трёх переключателей, целочисленные значения освещённости, освещённости окружения, освещённости по трём цветовым каналам, дробные значения температуры, давления и акселерометра по трём координатным осям.

3.3. Некоторые сведения о применяемых конструкциях

Язык программирования Python поддерживает такую синтаксическую конструкцию как декоратор. Она позволяет добавлять функциональность объекту не меняя его реализацию. Это напоминает называющийся так же паттерн проектирования в парадигме объектно-ориентированного программирования. Декоратор является функцией, которая принимает на вход декорируемую функцию и возвращает другую, которая теперь будет вызываться вместо изначальной. Таким образом можно выполнять какие-либо действия до или после выполнения оригинальной функции, изменять значения её аргументов, или даже менять её интерфейс. Декораторы также можно применять к классам. В таком случае на вход поступает экземпляр класса, который модифицируется и возвращается.

Синтаксис применения декоратора имеет следующий вид:

```
@my_decorator
def func(arg):
 pass


@class_decorator
class MyClass():
 def __init__(self):
  pass
```

В случае если в декоратор передаются аргументы, тогда добавляется дополнительный уровень вложенности, так что функция, принимающая аргумент возвращает декоратор, который сохранит в своей области видимости аргумент за счёт замыкания.

Пример:

```
def arg_decorator(arg):
 def decorator(func):
  def wrapper(*args, **kwargs):
   func(*ags, **kwargs)
   do_decorator_things(arg)
  return wrapper
 return decorator


@arg_decorator(5)
def do_something(another_arg):
 pass
```

Декораторы также можно применять к методам класса. Однако, сам декоратор не является членом класса. Поэтому для того, чтобы получать доступ к его экземпляру, можно в списке аргументов функции-обёртки указать `self`, но тогда нельзя забывать далее передать её в оригинальную функцию.

3.4. Реализация сетевых запросов в PyQT5

Что бы не вызвало сетевой запрос, его выполняют методы `send_message()` и `get_value_from_macket()` для POST и GET запросов соответственно. Обе функции получают URL запроса из соответствующего текстового поля, формируют объект `QNetworkRequest` и передают его в соответствующий метод `QnetworkAccessManager`'а. Однако, в методе для POST-запроса также при помощи метода `compose_post_json_data()` собираются данные для отправки и передаются вторым аргументом. `QNetworkReply` записывается в соответствующее типу запроса поле класса. После этого осуществляется привязка соответствующего обработчика.

При помощи декоратора `with_cancel(reply_name: str)` отменяется предыдущий запрос того же типа, вызывая слот `QNetworkReply.abort()`, так как при каждом новом запросе данные в старом теряют свою актуальность. Аргумент, `reply_name` позволяет задать название атрибута класса, в котором хранится объект `QNetworkReply`, с помощью которого контролируется выполнение запроса и из которого получаются данные, или тип ошибки. Соответствующий код на Python приведён в приложении Б, листинг 1.

К сигналам `finished` вышеописанных `QnetworkReply GET_reply` и `POST_reply` привязываются методы (слоты) `handle_get_reply()` и `handle_post_reply()`. Первый считывает данные из `GET_reply` и преобразует их объект Python, `dict data`. Они также выводятся в текстовое поле журнала запросов. Далее, они отображаются в интерфейсе подобно тому, как это сделано в конструкторе, где программа инициализируется из config-файла. Для POST запроса обработчик ещё проще — в нём данные, которые вернул сервер просто выводятся в журнал.

Однако, запрос может завершиться с ошибкой, или вовсе быть отменён. Для обработки данных случаев был введён декоратор `with_err_handling(reply_name: str)`. В нём из `QNetworkReply` берётся код ошибки и если он соответствует успешному завершению запроса, то

соответствующая запись добавляется в журнал и вызывается оригинальный обработчик. Иначе, если ошибка сообщает об отмене запроса, пишем это в журнал. Если произошло переподключение сети, сообщается об этом и предлагается повторить запрос. Если подключение оказалось невозможным, в журнал помещается соответствующая запись. Если же сервер вернул ошибку по HTTP, выводится ещё и её код. Описанный код приведён в приложении Б, листинг 2.

3.5. Реализация автоматической синхронизации данных

В задании на практику требовалось реализовать функционал выбора ручного/автоматического обновления данных и UI программы. Было принято решение при включенной автосинхронизации отправлять изменения, сделанные пользователем сразу, а получать обновления со стенда периодически. Такой вариант работы более интуитивен, а также избавляет от проблемы синхронизации и задержек GET и POST запросов, которые нужны были бы если бы оба запроса запускались по таймеру.

При запуске программы в конструкторе класса главного окна из config-файла считываются начальные значения интервала для таймера и автосинхронизации. В поле `timer` сохраняется экземпляр класса `QTimer`, который посылает сигнал `timeout` раз в заданное количество миллисекунд. К нему привязывается метод `get_value_from_macket()`. Описанный код приведён в приложении Б, листинг 3.

За переключение режима автообновления отвечает чекбокс. В его слот `stateChanged` подключается функция, которая, если чекбокс становиться отмеченным включает таймер с заданным в поле ввода периодом. В нём отображается время в секундах, поэтому его требуется домножить на 1000, чтобы перевести в миллисекунды. Также, чтобы в первый раз запустить обновление не дожидаясь завершения таймера, сигнал `timeout` посылается вручную. Если чекбокс выключается, тогда таймер останавливается.

Для автоматической отправки пользовательского ввода на сервер был

реализован декоратор `with_autosend`. В нём, сначала вызывается декорируемая функция, после чего, если включено автообновление, вызывается метод `send_message()`. Данный декоратор добавляется ко всем методам, обрабатывающим пользовательский ввод (кроме опля ввода периода автообновления). Описанный код приведён в приложении Б, листинг 4.

3.6. Прочие дополнения и исправления

Изначально, в config-файле присутствовала часть данных для отображения в интерфейсе. Поэтому было принято решение дополнить его и инициализировать состояние приложения из него.

Добавление обработчиков для сигналов переключения ламп осуществляется в цикле. Так как в слот необходимо передать индекс лампы, была использована конструкция `lambda`. При этом, чтобы сработало замыкание конкретного значения индекса на данной итерации, пришлось передавать его как её аргумент с инициализацией значения по умолчанию как показано ниже

```
for i in range(...):
... lambda val, i=i: self.handle_toggle_lamp(i, val)
```

В функции `main`, вызываемой при старте программы сначала создаётся экземпляр класса QApplication. Он используется для контроля за работой программы и отвечает за корректное её завершение с сообщением кода ошибки. Окно программы также создаётся в области видимости главной функции для того, чтобы приложение знало, когда можно удалить его.

Для тестирования приложения, в том числе сетевого стека в условиях отсутствия доступа к стенду было использовано ПО Mockoon для запуска тестового веб-сервера, реализующего реальный интерфейс, представленный в формате OpenAPI.

ЗАКЛЮЧЕНИЕ

Полученное приложение было запущено как напрямую через интерпретатор Python, так и будучи собранным в один исполняемый файл при помощи `pyinstaller` (подробнее в файле README.md). В качестве веб сервера использовался Mockoon. В результате тестирования ошибки не выявлены, весь функционал оригинального приложения реализован в полном объёме.

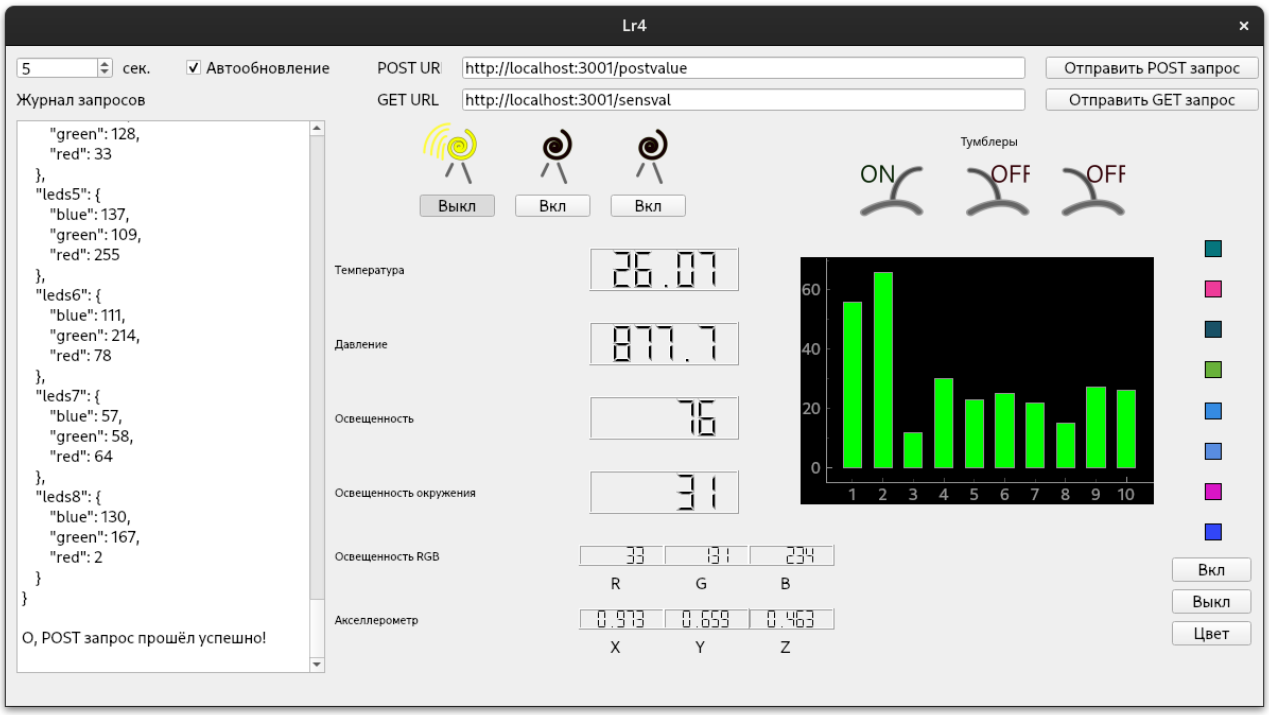

Рис. 3 Скриншот окна работающего приложения

В ходе работы были отработаны навыки понимания чужого кода и совместной разработки, применена система контроля версий git. Произошло ознакомление с библиотекой QT, применены и углублены знания Python, методов разработки приложений с неблокирующими операциями.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Таненбаум Э. Архитектура компьютера. 5-е изд: Питер, 2007. 844 с.
2. Документация Python для модуля threading // Python 3.11.3 documentation. URL: https://docs.python.org/3/library/asyncio.html (дата обращения 07.06.2023).
3. Документация Python для модуля asyncio // Python 3.11.3 documentation. URL: https://docs.python.org/3/library/threading.html (дата обращения 07.06.2023).
4. Документация для класса QThread // Qt Core 5.15.14. URL: https://doc.qt.io/qt-5/qthread.html (дата обращения 08.06.2023).
5. Документация для класса QnetworkAccessManager // Qt for Python. URL: https://doc.qt.io/qtforpython-5/PySide2/QtNetwork/QNetworkAccessManager.html (дата обращения 26.05.2023).
6. Спецификация формата OpenAPI // OpenAPI Specification - Version 3.0.3 | Swagger. URL: https://swagger.io/specification/ (дата обращения 08.06.2023).

ПРИЛОЖЕНИЕ А

OPENAPI СПЕЦИФИКАЦИЯ API


```
{
 "openapi": "3.0.0",
 "info": {
  "title": "Mocks",
  "version": "1.0.0"
 },
 "servers": [
  {
   "url": "http://localhost:3001/"
  }
 ],
 "components": {
  "schemas": {
   "ValuesRGB": {
    "type": "object",
    "properties": {
     "red": {
      "type": "number"
     },
     "green": {
      "type": "number"
     },
     "blue": {
      "type": "number"
```

```
      }
     },
     "required": [
      "red",
      "green",
      "blue"
     ]
    }
   }
  },
 "paths": {
  "/sensval": {
   "get": {
    "description": "",
    "responses": {
     "200": {
      "description": "",
      "content": {
       "application/json": {
        "schema": {
         "type": "object",
         "properties": {
          "LED1": {
           "type": "boolean"
          },
          "LED2": {
           "type": "boolean"
```

```
},
"LED3": {
 "type": "boolean"
},
"button1State": {
 "type": "boolean"
},
"button2State": {
 "type": "boolean"
},
"button3State": {
 "type": "boolean"
},
"temperature": {
 "type": "number",
 "format": "float"
},
"pressure": {
 "type": "number",
 "format": "float"
},
"ambient_light": {
 "type": "number",
 "format": "int32"
},
"red_light": {
 "type": "number",
```

```
 "format": "int32"
},
"green_light": {
 "type": "number",
 "format": "int32"
},
"blue_light": {
 "type": "number",
 "format": "int32"
},
"lightness": {
 "type": "number",
 "format": "int32"
},
"acceleration_x": {
 "type": "number",
 "format": "float"
},
"acceleration_y": {
 "type": "number",
 "format": "float"
},
"acceleration_z": {
 "type": "number",
 "format": "float"
},
"leds1": {
```

```
      "$ref": "#/components/schemas/ValuesRGB"
    },
    "leds2": {
      "$ref": "#/components/schemas/ValuesRGB"
    },
    "leds3": {
      "$ref": "#/components/schemas/ValuesRGB"
    },
    "leds4": {
      "$ref": "#/components/schemas/ValuesRGB"
    },
    "leds5": {
      "$ref": "#/components/schemas/ValuesRGB"
    },
    "leds6": {
      "$ref": "#/components/schemas/ValuesRGB"
    },
    "leds7": {
      "$ref": "#/components/schemas/ValuesRGB"
    },
    "leds8": {
      "$ref": "#/components/schemas/ValuesRGB"
    }
  },
  "example": {
    "LED1": false,
    "LED2": true,
```

```
"LED3": false,
"button1State": true,
"button2State": false,
"button3State": true,
"temperature": 25,
"pressure": 100,
"ambient_light": 100,
"red_light": 200,
"green_light": 300,
"blue_light": 400,
"lightness": 500,
"acceleration_x": 0.1,
"acceleration_y": 0.2,
"acceleration_z": 0.3,
"leds1": {
 "red": 100,
 "green": 150,
 "blue": 200
},
"leds2": {
 "red": 100,
 "green": 150,
 "blue": 200
},
"leds3": {
 "red": 100,
 "green": 150,
```

```
 "blue": 200
},
"leds4": {
 "red": 100,
 "green": 150,
 "blue": 200
},
"leds5": {
 "red": 100,
 "green": 150,
 "blue": 200
},
"leds6": {
 "red": 100,
 "green": 150,
 "blue": 200
},
"leds7": {
 "red": 100,
 "green": 150,
 "blue": 200
},
"leds8": {
 "red": 100,
 "green": 150,
 "blue": 200
}
```

```
        }
       }
      }
     },
     "headers": {}
    }
   }
  }
 },
"/postvalue": {
 "post": {
  "description": "",
  "requestBody": {
   "content": {
    "application/json": {
     "schema": {
      "type": "object",
      "properties": {
       "LED1": {
        "type": "boolean"
       },
       "LED2": {
        "type": "boolean"
       },
       "LED3": {
        "type": "boolean"
       },
```

```
    "leds1": {
     "$ref": "#/components/schemas/ValuesRGB"
    },
    "leds2": {
     "$ref": "#/components/schemas/ValuesRGB"
    },
    "leds3": {
     "$ref": "#/components/schemas/ValuesRGB"
    },
    "leds4": {
     "$ref": "#/components/schemas/ValuesRGB"
    },
    "leds5": {
     "$ref": "#/components/schemas/ValuesRGB"
    },
    "leds6": {
     "$ref": "#/components/schemas/ValuesRGB"
    },
    "leds7": {
     "$ref": "#/components/schemas/ValuesRGB"
    },
    "leds8": {
     "$ref": "#/components/schemas/ValuesRGB"
    }
   }
  }
}
```

```
      }
    },
    "responses": {
      "200": {
        "description": "",
        "content": {
          "text/plain": {
            "example": "POST body was:\n{ }"
          }
        },
        "headers": {}
      }
    }
  }
 }
}
```

ПРИЛОЖЕНИЕ Б

ЛИСТИНГИ С КОДОМ ПРОГРАММЫ

```
def collect_lamps_state(self) -> dict[str, bool]:
  """
  Composes proper object structure with lamps state
  """
  lamps_state = {}
  for i in range(1,4):
    lamps_state[f"LED{i}"] = getattr(self.ui, f"label_lamp_on{i}").isVisible()
  return lamps_state

def compose_post_json_data(self) -> dict:
  json_data = {}
  json_data.update(self.rgb_leds_state)
  json_data.update(self.collect_lamps_state())

  return json_data

def log_post_request(self, url: str, json_data: dict):
  json_str = json.dumps(json_data, separators=(',', ':'))
  data_str = 'Я отправляю текст на: ' + url + '\n'+ json_str
  self.ui.textEdit_message.append(data_str)

def with_cancel(reply_name: str): # retuns decorator with argument enclosed
  """
  Decorator for cancelling previous request of same type
  """
```

```
  def inner(func): # function decorator
    def wrapper(self): # the fuction that will be called as handler
      if hasattr(self, reply_name):
        getattr(self, reply_name).abort() # abort currently pending request

      func(self) # calling actual handler

    return wrapper
  return inner

@with_cancel('POST_reply')
def send_message(self):
  """
  POST запрос
  """
  # Get inputed url
  url = self.ui.lineEdit_POST_URL.text()
  # compose body
  json_data = self.compose_post_json_data()
  self.log_post_request(url, json_data)
  data = QJsonDocument(json_data)
  # Create request object
  request = QNetworkRequest(QUrl(url))
  # Set request headers
  request.setHeader(QNetworkRequest.ContentTypeHeader, 'application/json')
  request.setRawHeader(b'Accept', b'text/plain')
  # Do POST request and store its reply object
  self.POST_reply = self.nam.post(request, data.toJson())
```

```
  # Set callback for request finishing signal
  self.POST_reply.finished.connect(self.handle_post_reply)


@with_cancel('GET_reply')
def get_value_from_macket(self):
  """
  GET запрос
  """


  url = self.ui.lineEdit_GET_URL.text()
  request = QNetworkRequest(QUrl(url))
  self.GET_reply = self.nam.get(request)
  # Set callback for request finishing signal
   self.GET_reply.finished.connect(self.handle_get_reply)
```

Листинг 1. Код отправки сетевых запросов

```
def with_err_handling(reply_name: str): # retuns decorator with argument
enclosed
  """
  Decorator for error and cancellation of request handling
  """
  operation = reply_name.split('_')[0]
  def inner(func): # function decorator
    def wrapper(self): # the fuction that will be called as handler
      reply = getattr(self, reply_name) # gets actual reply by its name (ex:
self.GET_reply)
      err = reply.error()
      if err == QNetworkReply.NetworkError.NoError:
        self.ui.textEdit_message.append(f'O, {operation} запрос прошёл
успешно!')
```

```
            func(self) # calling actual handler
        elif err == QNetworkReply.NetworkError.OperationCanceledError:
            self.ui.textEdit_message.append(f"{operation} запрос был отменён, так как не успел выполниться до нового вызова")
        elif err == QNetworkReply.NetworkError.TemporaryNetworkFailureError:
            self.ui.textEdit_message.append(f"Произошла временная ошибка при {operation} запросе, повторите запрос ещё раз")
        else:
            msg = f"Ошибка при {operation} запросе: "
            if (err in CONNECTION_ERRORS):
                self.ui.textEdit_message.append(msg + 'не удалось установить подключение к серверу')
            else:
                status_code = reply.attribute(QNetworkRequest.Attribute.HttpStatusCodeAttribute)
                self.ui.textEdit_message.append(msg + f'сервер вернул статус код {status_code}')
    return wrapper
  return inner


@with_err_handling('GET_reply')
def handle_get_reply(self):
  res = self.GET_reply.readAll().data()
  data = json.loads(res)
  self.ui.textEdit_message.append(json.dumps(data, separators=(',', ':')))
  # Update lamps
  for i in range(1,4):
    self.handle_toggle_lamp(i, data[f"LED{i}"])
  self.update_buttons(self.convert_buttons_state(data))
```

```
  # Update LCDs
  self.update_lcds(data)
  # Append dot to plot and set corresponding LCD
  self.plot.update(data["temperature"])
  self.ui.lcd_temperature.display(data["temperature"])
  self.update_colors(data)


@with_err_handling('POST_reply')
def handle_post_reply(self):
  res = self.POST_reply.readAll().data()
   self.ui.textEdit_message.append(res.decode())
```

Листинг 2. Код обработки ответа на сетевые запросы

```
# Init timer and connect data fetching with interval from defaults
self.timer = QTimer(self)
self.timer.setInterval(conf["defaultUpdateInterval"])
self.timer.timeout.connect(self.get_value_from_macket)
# Init autoupdate interval input
self.ui.spinBox_autoupdate.setValue(conf["defaultUpdateInterval"] // 1000)
# Setup autoupdate interval updating on input change
self.ui.spinBox_autoupdate.valueChanged.connect(lambda i: self.timer.setInte
rval(i * 1000))
# Setup autoupdate toggler
self.ui.checkBox_autoupdate.stateChanged.connect(self.handle_toggle_autoupda
te)
self.ui.checkBox_autoupdate.setChecked(conf["startWithAutoupdate"])
```

Листинг 3. Код инициализации логики автообновления данных

```
def handle_toggle_autoupdate(self):
  """
  Toggles timer
  """
  if self.ui.checkBox_autoupdate.isChecked():
```

```
    interval_s = self.ui.spinBox_autoupdate.value()
    self.timer.start(interval_s * 1000)
    self.timer.timeout.emit() # Trigger timer event immediately
  else:
    self.timer.stop()


def with_autosend(func):
  """
  Decorator for doing automatic post request after function invocation
  """
  def wrapper(self, *args, **kwargs): # the fuction that will be called as handler args - positional arguments, kwargs - named arguments
    func(self, *args, **kwargs) # calling actual handler
    if self.ui.checkBox_autoupdate.isChecked():
      self.send_message()
  return wrapper
```

Листинг 4. Код переключения режима автообновления и декоратор для автоотправки данных на стенд