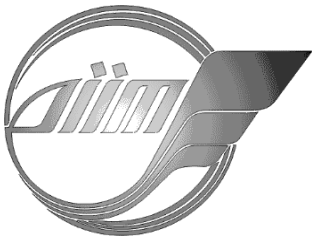
МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ



Дніпровський національний університет
залізничного транспорту імені академіка В. Лазаряна

Кафедра «Комп'ютерні інформаційні технології»

Курсова робота

з дисципліни «Об'єктно-орієнтоване програмування»

на тему: «Інформаційна база навчального закладу»

Виконав:
студент гр.ПЗ1911
Сафонов Д. Є.
Прийняла:
Демидович І. М.

Дніпро, 2021

Завдання на курсову роботу

Міністерство освіти і науки України
Дніпровський національний університет залізничного транспорту
імені академіка В. Лазаряна

Факультет: Комп’ютерні технології і системи
Кафедра: Комп’ютерні інформаційні технології
Спеціальність: 121 Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Зав. кафедрою "КІТ"
проф. Шинкаренко В.І.
"____"_____2021р.

З А В Д А Н Н Я
до курсової роботи

студента Сафонова Д. Є.

1. Тема проекту: Інформаційна база навчального закладу
2. Дата видачі завдання _____
3. Перелік питань до розробки:
    1. Опис предметної області та розробка специфікацій (вимог до програми).
        1.1. Постановка задачі.
        1.2. Вимоги до програми.
    2. Розробка об’єктно-орієнтованої моделі.
        2.1. Опис відповідальності класів.
        2.2. Опис відношень між класами.
        2.3. Діаграма класів.
        2.4. Діаграма послідовностей.
    3. Розробка та опис інтерфейсної частини класів.
    4. Розробка файлів реалізації класів. Проектування основних алгоритмів.
    5. Тестування програми.
    6. Відлагодження програми.
    7. Приклад роботи програми.
    8. Аналіз результатів. Переваги програми за рахунок застосування об’єктно-орієнтованої парадигми.
    9. Текст програми.
4. Термін виконання курсового проекту 31 травня 2021 року

Керівник курсового проектування Демидович І. М.
Завдання прийняв до виконання Сафонов Д. Є.

Зміст

Вступ

Об'єктно-орієнтоване програмування(ООП) — парадигма програмування, яка базується на концепті “об’єктів”, які можуть складатися із коду та даних(стану): даних у формі полів(атрибутів), та коду у вигляді процедур(методів).

Одна з особливостей об'єктів — можливість змінювати свій стан у процесі виконання методів(mutability), це виконується за допомогою неявної передачі посилання на об’єкт через аргумент this або self(в залежності від використаної мови програмування). Насправді коли ми викликаємо метод об’єкту деякого класу, це те саме що викликати функцію із модуля, де один з аргументів буде структурою із набором атрибутів аналогічним цьому класу.

Програми створені із допомогою об'єктно-орієнтованих мов програмування розроблюються як об’єкти, які взаємодіють між собою. Також ОО мови можуть бути базовані на прототипах, класах, акторах. Більшість мов базовані на класах, але існують популярні мови базовані на прототипах(наприклад ECMAScript(JavaScript)).

Більшість популярних об’єктно-орієнтованих мов програмування не є чистими, а складаються із декількох парадигм, найчастіше з: імперативною, процедурною, функціональною.

Об'єктно-орієнтовані мови програмування мають спільні риси із не ОО мовами: змінні, які можуть зберігати примітивні типи даних або об’єкти, процедури, які називаються методами, але виконують ту саму функцію(з цієї точки зору можна вважати що програма написана на імперативній мові програмування є класом, а якщо запустити її — отримаємо об’єкт, викликавши його процедури ми можемо змінити його стан, або отримати його). Інкапсуляція також не є рисою тільки об’єктно-орієнтованих мов, вона є мова програмування, які мають концепт модуля(декілька процедур, констант тощо можна об’єднати в один модуль та зробити деякі з цих об’єктів приватними(не у тому самому сенсі що в ООП)). Поліморфізм також існує в інших парадигмах, і не тільки у тому вигляді що в ООП(поліморфізм підтипів).

Через дуже велику популярність об’єктно-орієнтованого підходу у розробці програм були виділені практики, які можна використовувати у дуже різних випадках для покращення коду(з точки зору читабельності, розширюваності тощо), річ про патерни розробки, у 1994 навіть вийшла книга “Шаблони проєктування: Елементи повторно використовуваного об'єктно-орієнтованого програмного забезпечення”, автори книги: Еріх Гамма (англ. Erich Gamma), Річард Хелм (англ. Richard Helm), Ральф Джонсон (англ. Ralph Johnson), Джон Вліссідес (англ. John Vlissides). Колектив авторів також відомий як «Банда чотирьох» (англ. Gang of Four; GoF). Книга описує 23 популярних проблеми та патерни, які допомагають їх вирішити.

1. Опис предметної області та розробка специфікацій

1.1. Постановка задачі

Основною задачею курсового проекту є розробка програми, яка допомогла б з організацією навчального процесу, а саме програма має надавати:

- розклад занять
- інформацію про предмети
- інформацію про групи/класи, студентів, викладачів

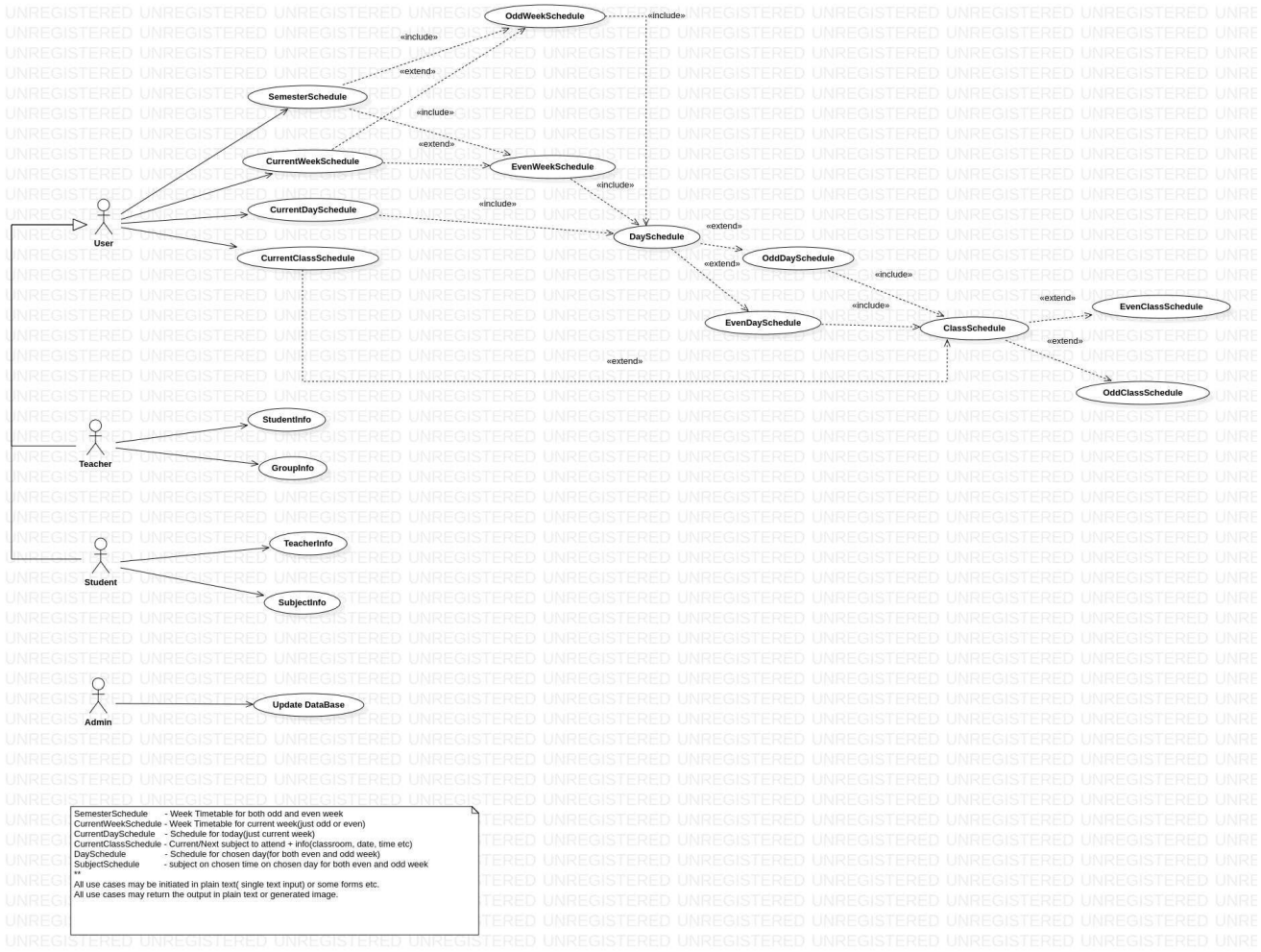


Рисунок 1: Діаграма usecase

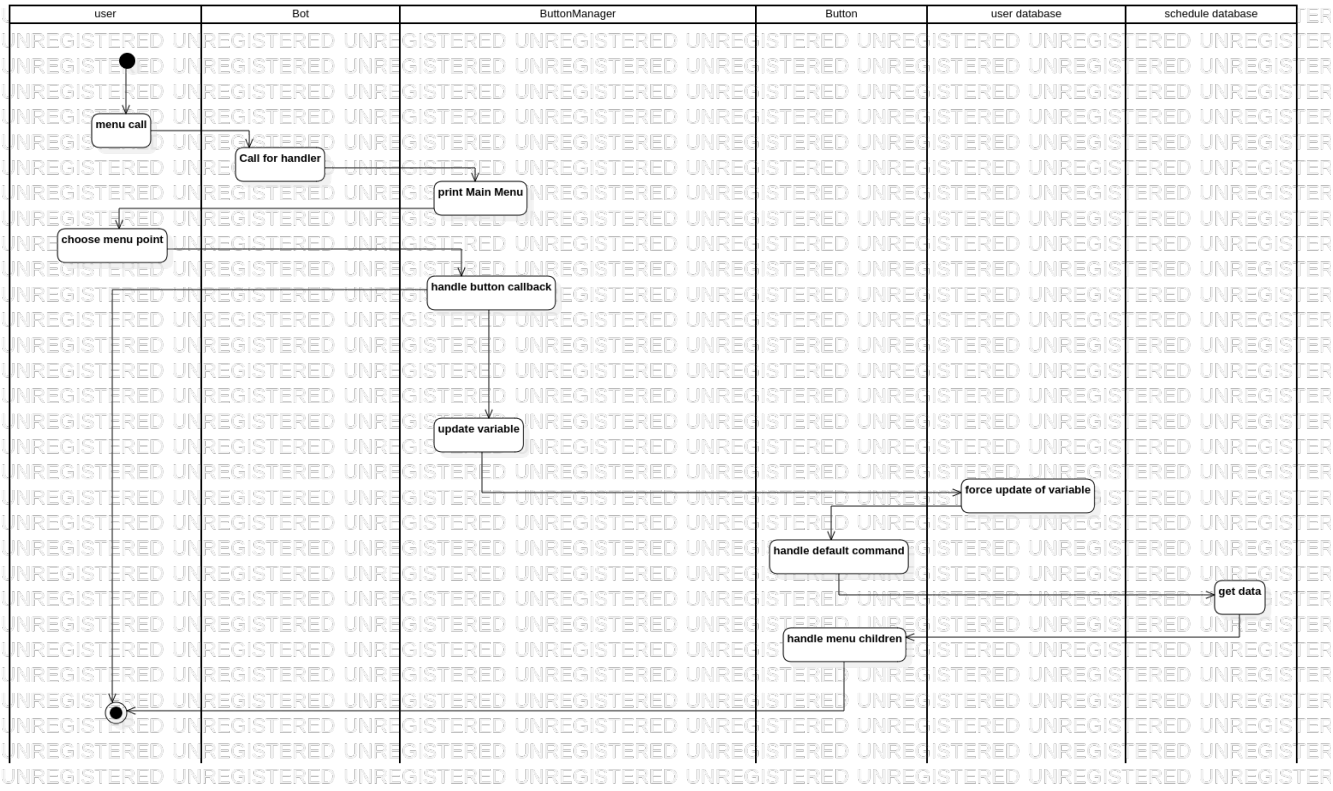


Рисунок 2: Діаграма activity(для одного циклу програми)

1.2. Вимоги до програми

Програма має задовільняти наступні вимоги:

- Стійкість до некоректних команд
- Модульність
- Стандартизація

1.3. Методологія програмування

Об’єктно-орієнтоване програмування(ООП) — підхід у програмуванні, заснований на поєднані даних та коду у об’єктах, і створенні програми із таких об’єктів.

1.4. Середовище розробки і мова програмування

Курсовий проект був написаний із використанням мов python, SQL у текстовому редакторі vscode(Visual Studio Code).

Ця мова була обрана за простоту та гибкість.

Вибір середовища був з двох текстових редакторів: vim та vscode, і хоч перший досить швидкий, він є дещо складнішим у роботі з відносно великими проєктами.

1.5. Вимоги до вхідних даних

Вхідні дані мають подаватися у вигляді текстовик команд, але із допомогою фронт-енд кінцевий користувач не обов’язково має вводити їх у такому вигляді.

1.6. Вимоги до вихідних даних

Вихідні дані будуть подаватися у вигляді об’єктів, які будуть трансформовані у текст або картинки.

1.7. Вимоги до функціональності

Програма має дозволяти:

- Подивитися розклад занять:
  - поточний(яка пара/урок наступний)
  - на сьогодні(має змінюватися щодня та щотижня(чисельник/знаменник))
  - на тиждень(можна обрати чисельник/знаменник або обидва)
- Подивитися інформацію про предмет:
  - Викладачів
  - Групи або класи, які вивчають цей предмети
- Подивитися інформацію про викладачів та студентів

2. Розробка об’єктно-орієнтованої моделі

2.1. Опис розподілу відповідальностей між класами та зв’язків між ними

Клас	Відповідальність	Зв'язки
Button	Виконання дії після натискання на кнопку	Спадкування: **LeafButton**(Нащадок) **Menu**(Нащадок) Агрегація: **ButtonManager**(Button е складовою ButtonManager) Агрегація: **Menu**(Button е складовою Menu)
LeafButton	Виведення результату після натискання на кнопку	Спадкування: **Button**(Батьківський клас) Агрегація: **Menu**(Menu може мати LeafButton в собі)
Menu	Виведення нового меню після натискання на кнопку	Спадкування: **Button**(Батьківський клас) Агрегація: **LeafButton**(Menu може мати LeafButton в собі) **Menu**(Menu може мати Menu в собі)
ButtonManager	Складне меню на основі Menu, LeafButton, Button	Композиція: **Button**(ButtonManager може мати в собі LeafButton та/або Menu) **Schedule**(Schedule е складовою ButtonManager) **UserDbManager**(UserDbManager е складовою ButtonManager) Залежність: **Bot**(Bot залежить від ButtonManager)

Bot	Головний клас, який контролює взаємодію з користувачем	Залежність: **ButtonManager**(Bot залежить від ButtonManager) Агрегація: **Command**(Command є складовою Bot)
Command	Асоціація ключового слова із обробником та описом команди	Агрегація: **Bot**(Command є складовою Bot)
UserDbManager	Зручний інтерфейс до бази даних користувачів	Агрегація: **ButtonManager**(UserDbManager є складовою ButtonManager) Залежність: **User**(UserDbManager залежить від User)
User	Зберігання стану користувача	Залежність: **User**(UserDbManager залежить від User)
Schedule	Зручний інтерфейс до бази даних розкладу	Композиція: **Schedule**(Schedule є складовою ButtonManager)
MultiPageMenu	Багатосторінкове меню	Спадкування: **Menu**(Батьківський клас) **CalendarMenu**(Нащадок) **MultiPageListMenu**(Нащадок)
ListMenu	Односторінкове меню зі списку опцій	Спадкування: **Menu**(Батьківський клас)
MultiPageListMenu	Багатосторінкове меню зі списку опцій	Спадкування: **MultiPageMenu**(Батьківський клас)
CalendarMenu	Багатосторінкове меню-календар	Спадкування: **MultiPageMenu**(Батьківський клас)

2.2. Діаграма класів

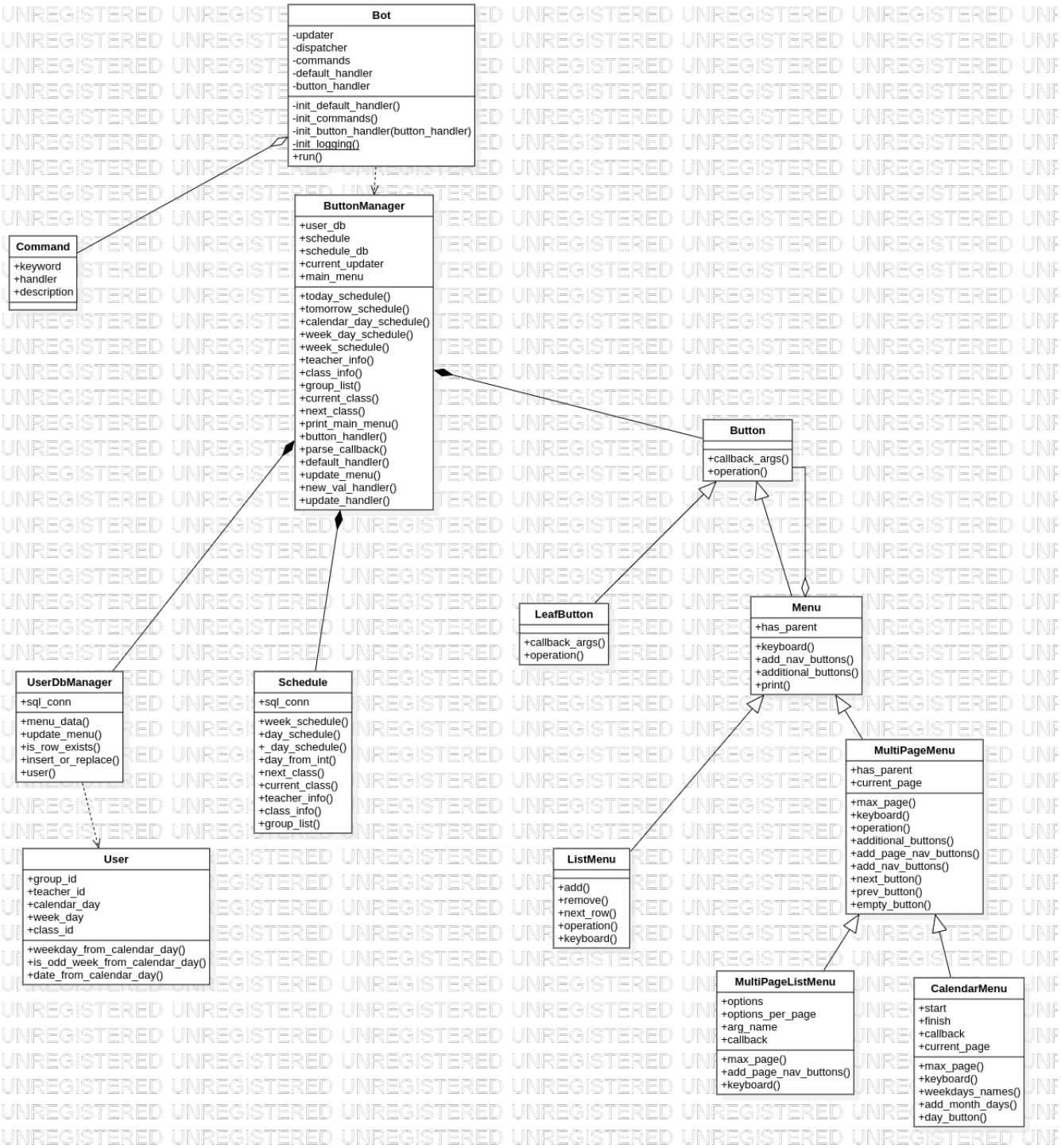


Рисунок 3: Діаграма класів

2.3. Діаграма послідовності

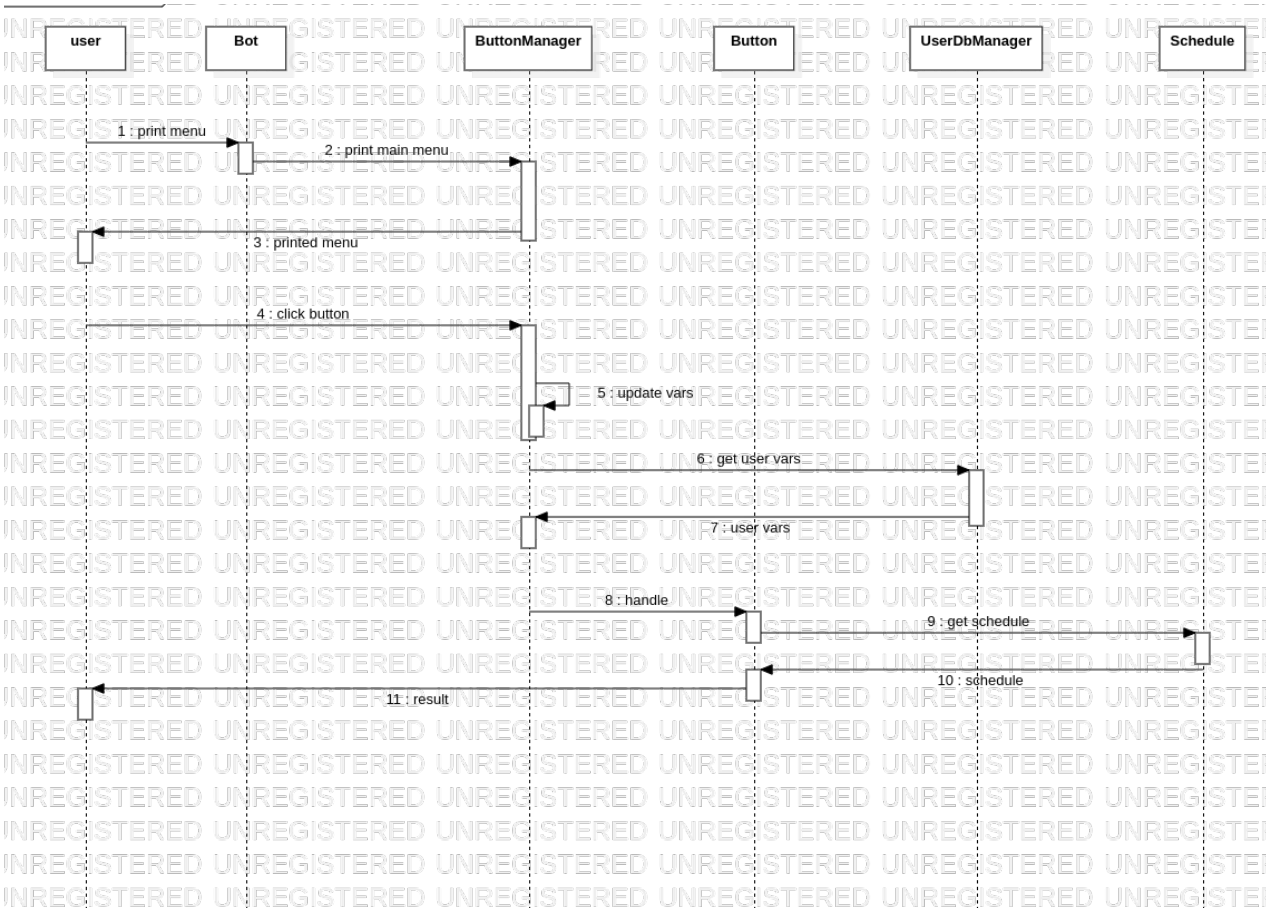


Рисунок 4: діаграма послідовності

3. Розробка та опис інтерфейсної частини класів.

3.1. Bot

Атрибути:

- updater — слухач подій
- dispatcher — обробник подій
- commands — колекція команд доступних до використання
- default_handler — обробник усіх повідомлень
- button_handler — обробник натиску на кнопку

Операції:

- _init_default_handler — ініціалізація обробнику усіх подій
- _init_commands — ініціалізація команд
- _init_button_handler — ініціалізація обробнику натискання на кнопку
- _init_logging — ініціалізація повідомлень про стан програми
- run — запуск бота

3.2. Command

Атрибути:

- keyword — ключове слово за яким викликається команда
- handler — обробник команди
- description — опис команди

3.3. ButtonManager

Атрибути:

- user_db — менеджер бази даних користувачів
- schedule — база даних розкладу
- schedule_db — менеджер бази даних розкладу
- current_updater — поточний оновлювач стану користувача
- main_menu — головне меню

Операції:

- today_schedule — розклад на сьогодні для обраної групи
- tomorrow_schedule — розклад на завтра для обраної групи
- calendar_day_schedule — розклад для обраного календарного дня та групи
- week_day_schedule — розклад для обраного дня тижня та групи
- week_schedule — розклад на весь тиждень
- teacher_info — інформація про обраного викладача
- class_info — інформація про обраний предмет
- group_list — список учнів у групі
- current_class — поточне заняття
- next_class — наступне заняття
- print_main_menu — вивести головне меню
- button_handler — обробити натиск кнопки
- parse_callback — розбити команду на атомарні частини
- _default_handler — обробник простих команд
- _update_menu — оновити історію меню
- new_val_handler — оновити стан користувача відповідно команді
- _update_hanlder — запит на оновлення від користувача

3.4. Schedule

Атрибути:

- sql_conn — база даних

Операції:

- week_schedule — розклад на весь тиждень
- day_schedule — розклад на обраний день тижня
- day_from_int — назва дня з числа(1-7)
- next_class — наступне заняття
- current_class — поточне заняття
- teacher_info — інформація про викладача
- class_info — інформація про предмет
- group_list — список учнів

3.5. UserDbManager

Атрибути:

- sql_conn — база даних

Операції:

- menu_data — історія меню
- update_menu — оновити історію меню
- is_row_exists — перевірити чи є користувач у базі даних
- insert_or_replace — оновити дані користувача, або додати у базу, якщо це новий користувач
- user — інформація про користувача за ідентифікатором

3.6. User

Атрибути:

- group_id — остання обрана група
- teacher_id — ідентифікатор останнього обраного викладача
- calendar_day — останній обраний календарний день
- week_day — останній обраний день тижня
- class_id — останній обраний предмет

Операції:

- weekday_from_calendar_day — день тижня обраного календарного дня
- is_odd_week_from_calendar_day — парність тижня обраної дати
- date_from_calendard_day — об'єкт обраної дати

3.7. Button

Атрибути:

- text — текст на кнопці
- callback — дані, яка повертає кнопка при натиску
- handler — обробник натиску на кнопку
- arg1name — ім’я аргументу, який доцільно оновити перед використанням кнопки
- arg2name — ім’я аргументу, який доцільно оновити перед використанням кнопки

Операції:

- callback_args — додаткові дані, які повертає кнопка
- operation — операція кнопки(також обробник натиску)

3.8. LeafButton

Атрибути:

- text — текст на кнопці
- callback — дані, яка повертає кнопка при натиску
- handler — обробник натиску на кнопку
- arg1name — ім’я аргументу, який доцільно оновити перед використанням кнопки
- arg2name — ім’я аргументу, який доцільно оновити перед використанням кнопки

Операції:

- callback_args — додаткові дані, які повертає кнопка
- operation — операція кнопки(також обробник натиску)

3.9. Menu

Атрибути:

- has_parent — чи е у цього меню батьківське меню(потрібно для визначення чи потрібна кнопка повернення до попереднього меню)

Операції:

- keyboard — створити клавіатуру
- _add_nav_buttons — додати кнопки навігації до клавіатури(вихід та повернення до попереднього)
- _additional_buttons — додаткові кнопки(навігації у тому числі)
- print — вивести клавіатуру(меню)

3.10. MultiPageMenu

Атрибути:

- has_parent — те саме що й в Menu
- current_page — поточний номер сторінки

Операції:

- keyboard — те саме що й в Menu
- operation — те саме що й в Button
- max_page — номер сторінки на одинцю більший за максимальний
- _additional_buttons — те саме що й в Menu
- _add_nav_buttons — те саме що й в Menu
- _add_page_nav_buttons — додати кнопки навігації по сторінкам цього меню
- _next_button — кнопка переходу на наступну сторінку
- _prev_button — кнопка переходу на попередню сторінку
- empty_button — пуста кнопка без тексту, при натиску нічого не змінюється

3.11. ListMenu

Атрибути:

- _children - кнопки-нащадки
- text — текст кнопки
- callback — те саме що й в Button

Операції:

- add — додати кнопку-нащадок
- remove — видалити кнопку
- next_row — перейти до наступного рядку
- operation — те саме що й в Button
- keyboard — те саме що й в Button

3.12. MultiPageListMenu

Атрибути:

- options — опції
- options_per_page — кількість опцій на сторінку
- arg_name — ім’я аргументу, значення якого ми оновлюємо
- callback — те саме що й в Button

Операції:

- max_page — те саме що й в MultiPageMenu
- _add_page_nav_buttons — те саме що й в MultiPageMenu
- keyboard — те саме що й в Button

3.13. CalendarMenu

Атрибути:

- start — стартова дата
- finish — кінцева дата
- current_page — те саме що й в MultiPageMenu
- callback — те саме що й в Button

Операції:

- max_page — те саме що й в MultiPageMenu
- keyboard— те саме що й в Button
- weekdays_names — додати дні тижня до календаря
- _add_month_day — додати дні місяця до календаря
- _day_button — створити кнопку дня за датою

4. Розробка файлів реалізації класів.

4.1. Файли

1. bot.py — class Bot, class Command
2. button_manager.py — class ButtonManager
3. button.py — class Button, LeafButton, Menu, ListMenu
4. calendar_menu.py — class CalendarMenu
5. main.py — вхідна точка
6. multipage_list_menu.py — class MultiPageListMenu
7. multipage_menu.py — class MultiPageMenu
8. schedule.py — class Schedule
9. user_db_manager.py — class UserDbManager
10. user.py — class User

4.2. Діаграми Нассі-Шнейдермана

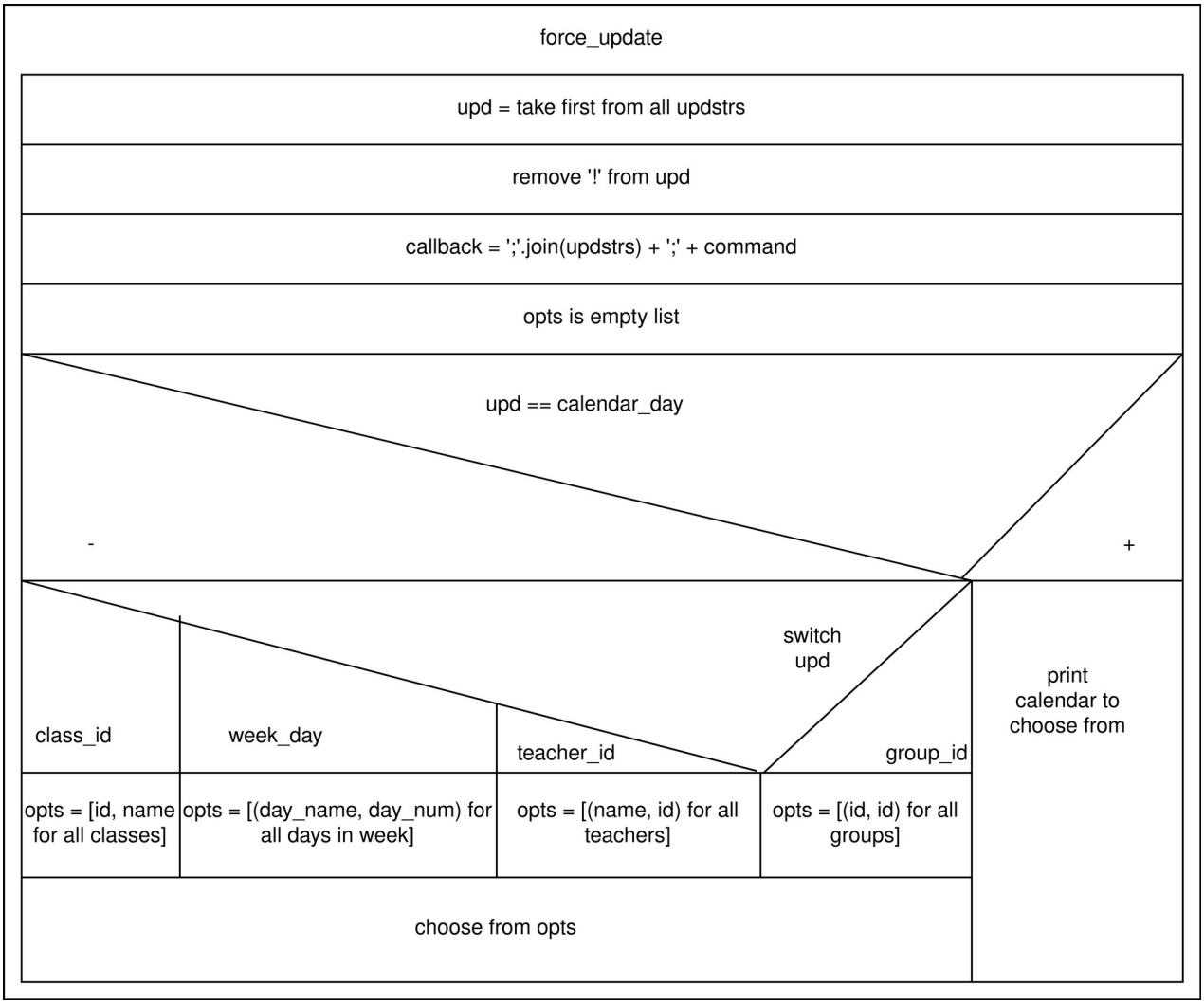
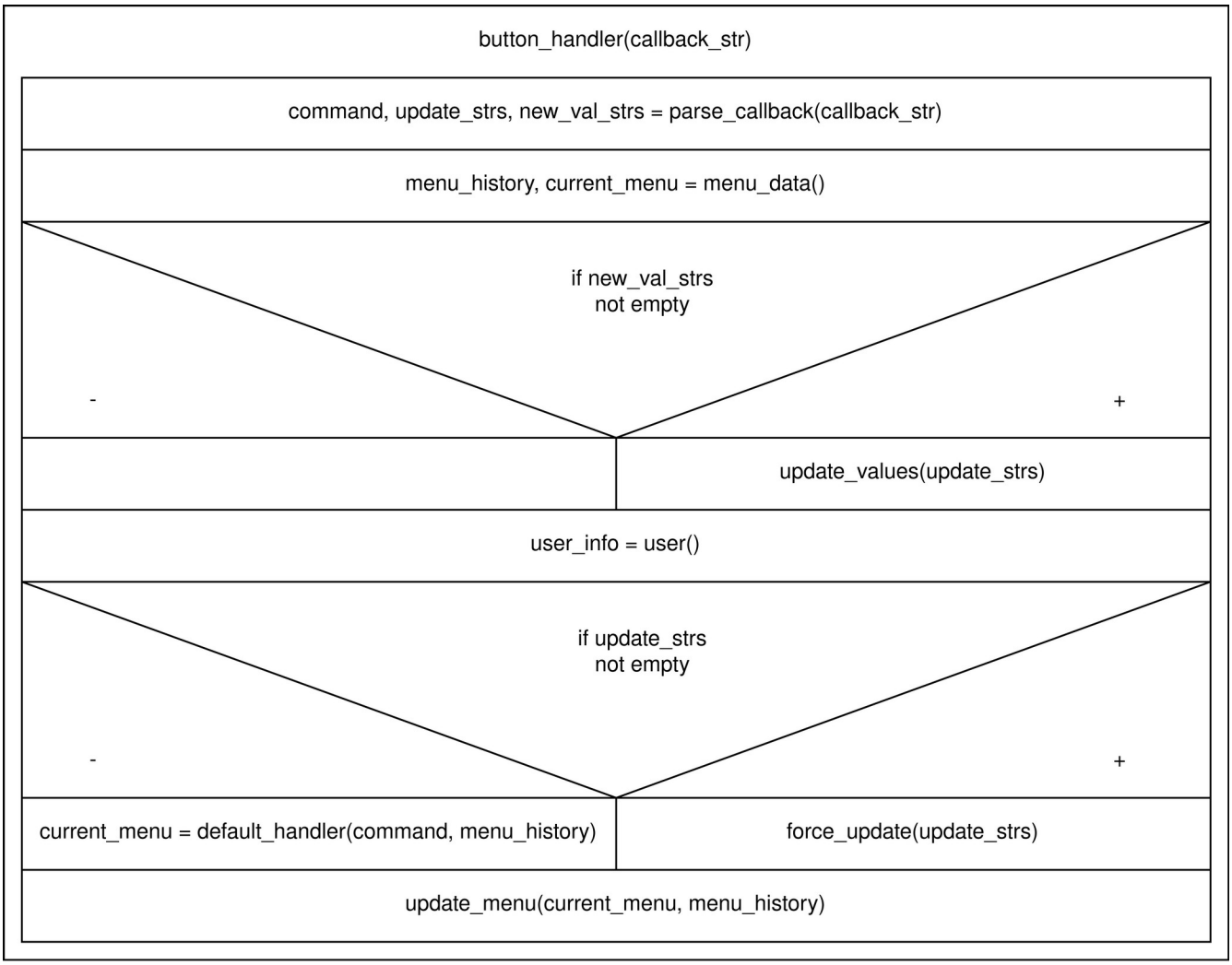


Рисунок 5: діаграма Нассі-Шнейдермана для запиту оновлення від користувача



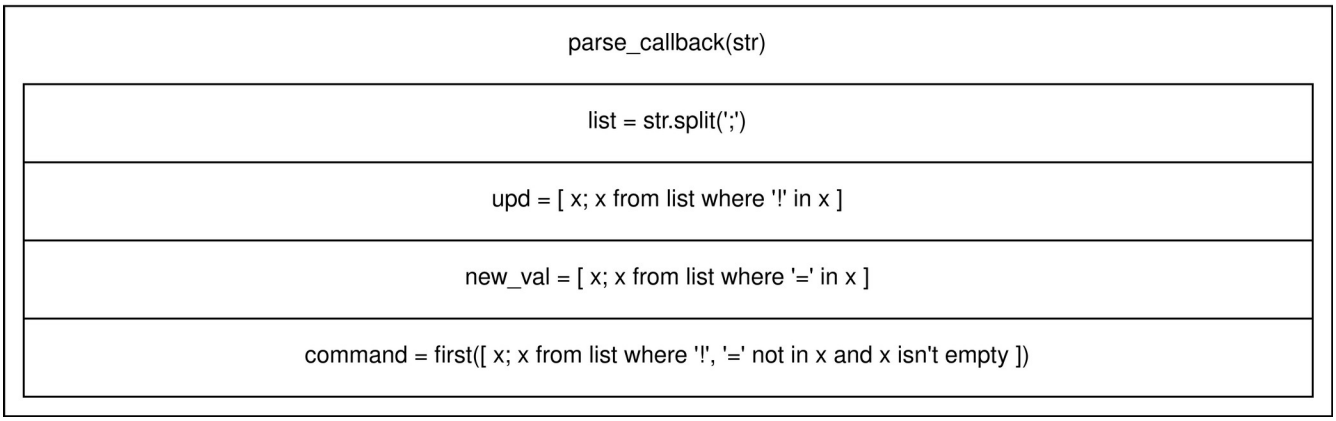


Рисунок 6: Діаграми Нассі-Шнейдермана для обробки натиску на кнопку та розбору команди

5. Тестування програми.

5.1. Чорна скринька

Метод тестування чорною скринькою полягає у тестуванні функціоналу програмного забезпечення без інформації про його внутрішню будову. Такий метод може бути застосований на усіх рівнях тестування: від тестування атомарних об'єктів до тестування цілої програми. Також його іноді називають “Тестуванням на специфікацію”.

5.2. Біла скринька

Метод тестування чорною скринькою полягає у тестуванні програмного забезпечення відносного його внутрішньої будови, але такий метод може бути застосований тільки при попередньо розробленому програмному дизайні.

Обидва методи мають свої недоліки та переваги. Наприклад метод чорної скриньки дозволяє перевірити логічні помилки програміста, коли біла скринька виявляє розходження щодо розробленого дизайну.

Помилок не було виявлено через те, що вхідні дані дуже дискретні.

6. Приклад роботи програми.

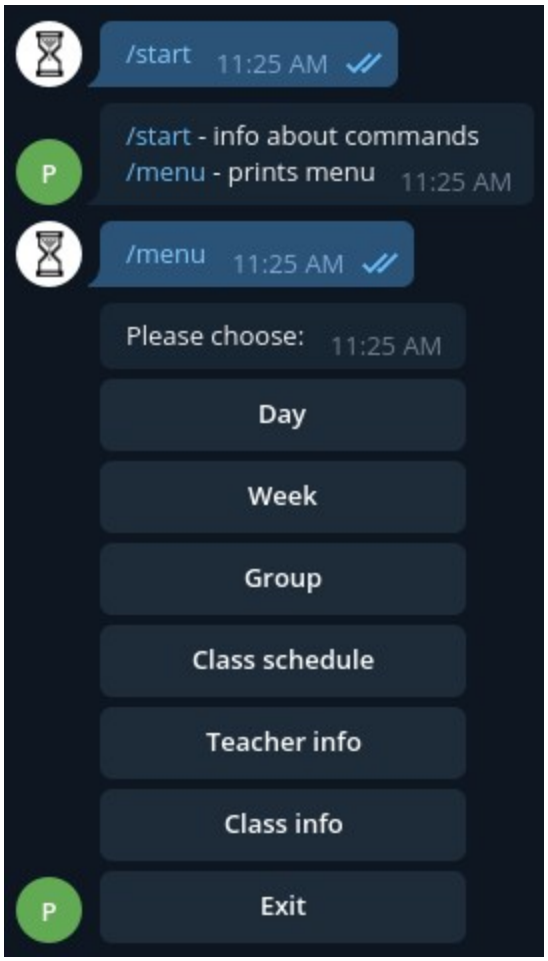


Рисунок 7: Головне меню

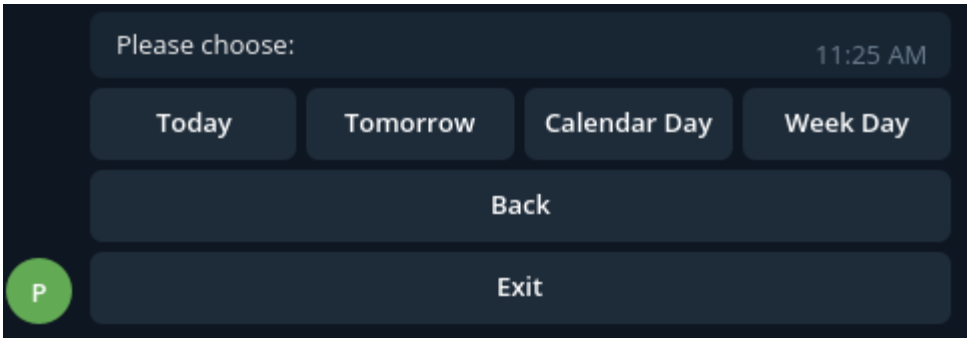


Рисунок 8: Внутрішнє меню

Рисунок 9: Багатосторінкове меню

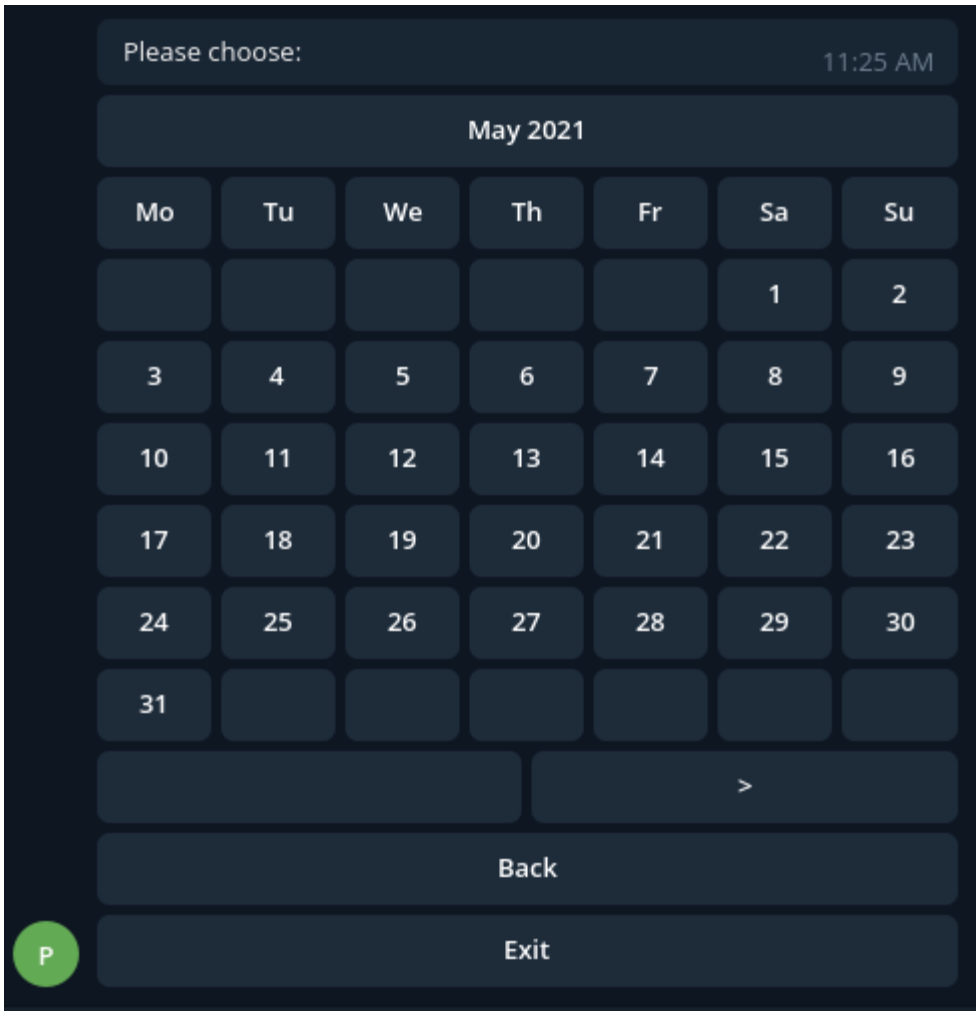


Рисунок 10: Календар

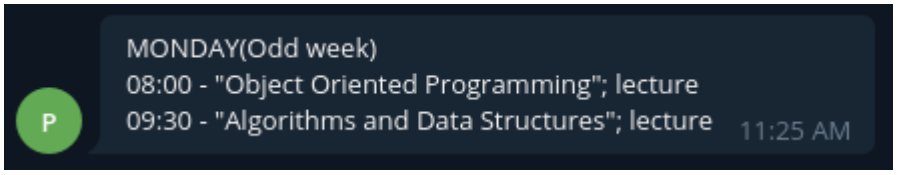


Рисунок 11: Результат

7. Аналіз результатів.

Об’єктно-орієнтоване програмування дозволяє набагато легше використовувати частки коду тому, що при розробці вони одразу мають бути структурованими. Також ця парадигма сприяє скороченню великих кусків коду, тому що він має бути розділений на маленькі методи. ООП існує вже досить давно і є самою популярною парадигмою програмування, тож було знайдено багато шаблонів, які можуть допомогти у багатьох ситуаціях. Одною з переваг ООП є поєднання коду та даних у об'єкти які мають стан, таким чином при розробці легше слідкувати за, наприклад, аргументами функцій.

Але головними перевагами ООП є:

- абстракція — виділення у складному об’єкті основного, потрібного для вирішення задачі
- інкапсуляція — поєднання коду та даних, можливо укриття деякої їх частини
- успадкування — дозволяє виносити код з багатьох споріднених класів у один батьківський
- поліморфізм(підтипів) — дозволяє використовувати класи з точки зору їх батьківських родичей.

8. Висновки

У розробленій програмі більш допомогли усі переваги ООП. Абстракція і інкапсуляція — при створенні класів. Успадкування та поліморфізм при створення їх ієрархій. Також були використані наступні патерни:

- “Компонувальник” — Меню складається із кнопок, які у свою чергу можуть бути меню
- “Ланцюжок обов’язків” — Меню намагається обробити команду, але якщо не може — передає до дочірніх кнопок, які також можуть передати до своїх.

9. Література

10. Додатки

“bot.py”

```
"""
TELEGRAM BOT
"""

import logging
from typing import Callable, List
from dataclasses import dataclass
from telegram.ext import (Updater,
CommandHandler,
MessageHandler,
Filters,
CallbackQueryHandler)
from user import User


@dataclass
class Command:
"""TELEGRAM BOT COMMAND HANDLER"""
keyword: str
handler: Callable[[User], str]
description: str


class Bot:
"""
TELEGRAM BOT
"""
def __init__(
self,
token: str,
commands: List[Command],
default_handler=None, button_handler=None
):
self._updater = Updater(token=token, use_context=True)
self._dispatcher = self._updater.dispatcher

self._commands = commands
self._init_commands()

self._default_handler = default_handler
self._init_default_handler()

self._init_button_handler(button_handler)
```

```
self._init_logging()


def _init_default_handler(self):
if self._default_handler is not None:
handler = MessageHandler(
Filters.text & (~Filters.command),
self._default_handler
)
self._dispatcher.add_handler(handler)


def _init_commands(self):
for command in self._commands:
handler = CommandHandler(command.keyword, command.handler)
self._dispatcher.add_handler(handler)


def _init_button_handler(self, button_handler):
if not button_handler is None:
handler = CallbackQueryHandler(button_handler)
self._dispatcher.add_handler(handler)


@staticmethod
def _init_logging():
logging.basicConfig(
format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
level=logging.DEBUG
)


def run(self):
"""START UPDATING BOT"""
self._updater.start_polling()
```

“button_manager.py”

```
"""
Button manager for timetable bot, using button module
"""

import datetime
from typing import List
from sqlite3 import Connection
from telegram import CallbackQuery, Message, Update
from telegram.ext import CallbackContext
from button import LeafButton, ListMenu
from user import User
from multipage_list_menu import MultiPageListMenu
from schedule import Schedule
from calendar_menu import CalendarMenu
from user_db_manager import UserDbManager


class ButtonManager:
"""
Button manager for timetable bot, using button module
"""
def __init__(self, user_db: Connection, schedule_db: Connection):
self.user_db = UserDbManager(user_db)
self.schedule = Schedule(schedule_db)
self.schedule_db = schedule_db
self.current_updater = None
self.main_menu = ListMenu('Menu', 'menu')

# main_menu init
self.day_menu = ListMenu('Day', 'day', self.main_menu)
self.main_menu.next_row()
self.week_menu = ListMenu('Week', 'week', self.main_menu)
self.main_menu.next_row()
self.group_menu = ListMenu('Group', 'group', self.main_menu)
self.main_menu.next_row()
self.class_menu = ListMenu(
'Class schedule', 'class_sch', self.main_menu
)
self.main_menu.next_row()
self.teacher_menu = LeafButton(
'Teacher info', 'teacher_info', self.main_menu,
self.teacher_info, 'teacher_id'
)
self.main_menu.next_row()
self.student_menu = LeafButton(
'Class info', 'class_info', self.main_menu,
```

```
self.class_info, 'class_id'
)
self.main_menu.next_row()

# main_menu.day_menu init
self.today_button = LeafButton(
'Today', 'today', self.day_menu,
self.today_schedule, 'group_id'
)
self.tomorrow_button = LeafButton(
'Tomorrow', 'tomorrow', self.day_menu,
self.tomorrow_schedule, 'group_id'
)
self.calendar_day_button = LeafButton(
'Calendar Day', 'calendar_day_button', self.day_menu,
self.calendar_day_schedule, 'group_id', 'calendar_day'
)
self.week_day_menu = ListMenu('Week Day', 'weekday_button', self.day_menu)

# main_menu.day_menu.week_day_menu init
self.wholeweek_day_button = LeafButton(
'Whole Week(Odd & Even) Day', 'whole_week_day', self.week_day_menu,
self.week_day_schedule(None), 'group_id', 'week_day'
)
self.oddweek_day_button = LeafButton(
'Odd Week Day', 'odd_week_day', self.week_day_menu,
self.week_day_schedule(True), 'group_id', 'week_day'
)
self.evenweek_day_button = LeafButton(
'Even Week Day', 'even_week_day', self.week_day_menu,
self.week_day_schedule(False), 'group_id', 'week_day'
)

# main_menu.week_menu init
self.whole_week_button = LeafButton(
'Whole Week(Odd & Even)', 'whole_week', self.week_menu,
self.week_schedule(None), 'group_id'
)
self.odd_week_button = LeafButton(
'Odd Week', 'odd_week', self.week_menu,
self.week_schedule(True), 'group_id'
)
self.even_week_button = LeafButton(
'Even Week', 'even_week', self.week_menu,
self.week_schedule(False), 'group_id'
)
```

```
# main_menu.group_menu init
self.all_students_button = LeafButton(
'All Students', 'all_students', self.group_menu,
self.group_list(None), 'group_id'
)
self.subgroup1_button = LeafButton(
'Subgroup1', 'subgroup1', self.group_menu,
self.group_list(1), 'group_id'
)
self.subgroup2_button = LeafButton(
'Subgroup2', 'subgroup2', self.group_menu,
self.group_list(2), 'group_id'
)

# main_menu.class_menu init
self.current_class_button = LeafButton(
'Current class', 'curr_class', self.class_menu,
self.current_class, 'group_id'
)
self.current_class_button = LeafButton(
'Next class', 'next_class', self.class_menu,
self.next_class, 'group_id'
)


def today_schedule(self, user: User) -> str:
"""schedule for today from underlying database(self.schedule)"""
day = datetime.datetime.today().weekday() + 1
if day > 7:
day -= 7
is_odd_week = datetime.datetime.today().isocalendar().week % 2 == 1
return self.schedule.day_schedule(day, user.group_id, is_odd_week)


def tomorrow_schedule(self, user: User) -> str:
"""schedule for tomorrow from underlying database(self.schedule)"""
day = datetime.datetime.today().weekday() + 2
if day > 7:
day -= 7
is_odd_week = (datetime.datetime.today()
+ datetime.timedelta(days=1)).isocalendar().week % 2 == 1
return self.schedule.day_schedule(day, user.group_id, is_odd_week)


def calendar_day_schedule(self, user: User) -> str:
"""schedule for calendar day from underlying database(self.schedule)"""
return self.schedule.day_schedule(
```

```
user.weekday_from_calendar_day(),
user.group_id,
user.is_odd_week_from_calendar_day()
)


def week_day_schedule(self, is_odd_week: bool):
"""schedule for week day from underlying database(self.schedule)"""
def wds(user) -> str:
return self.schedule.day_schedule(
user.week_day,
user.group_id,
is_odd_week
)
return wds


def week_schedule(self, is_odd_week: bool):
"""schedule for week from underlying database(self.schedule)"""
def week_sch(user) -> str:
return self.schedule.week_schedule(
user.group_id,
is_odd_week
)
return week_sch


def teacher_info(self, user: User):
"""wrapper for schedule.teacher_info(user.teacher_id)"""
return self.schedule.teacher_info(user.teacher_id)


def class_info(self, user: User):
"""wrapper for schedule.teacher_info(user.class_id)"""
return self.schedule.class_info(user.class_id)


def group_list(self, subgroup: int):
"""wrapper"""
def gr_ls(user):
return self.schedule.group_list(user.group_id, subgroup)
return gr_ls


def current_class(self, user: User):
"""wrapper"""
```

```
now = datetime.datetime.now()
return self.schedule.current_class(
now.weekday() + 1,
now.hour,
now.minute,
user.group_id,
now.isocalendar().week % 2 == 1
)


def next_class(self, user: User):
"""wrapper"""
now = datetime.datetime.now()
return self.schedule.next_class(
now.weekday() + 1,
now.hour,
now.minute,
user.group_id,
now.isocalendar().week % 2 == 1
)


def print_main_menu(self, message: Message):
"""
creates new message with main menu keyboard
"""
self._update_menu(self.main_menu.callback, [], message.chat_id)
self.main_menu.operation(
message,
self.main_menu.callback,
self.user_db.user(message.chat_id)
)


def button_handler(self, update: Update, _context: CallbackContext):
"""
handles callback buttons
"""
query = update.callback_query
query.answer()
command, update_strs, new_val_strs = self._parse_callback(query.data)
chat_id = update.effective_chat.id
menu_history, current_menu = self.user_db.menu_data(chat_id)
if new_val_strs:
self._new_val_handler(new_val_strs, chat_id)

user_info = self.user_db.user(chat_id)
```

```
if update_strs:
self._update_handler(
update_strs, command, menu_history,
current_menu, query, user_info
)
else:
current_menu = self._default_handler(
command, query, menu_history, user_info, current_menu
)
self._update_menu(current_menu, menu_history, chat_id)


@staticmethod
def _parse_callback(callback_str: str):
callback_list = callback_str.split(';')
command_str = next(filter(
lambda str_: '!' not in str_ and '=' not in str_ and str_ != '',
callback_list
))
update_strs = [update for update in callback_list if '!' in update]
new_val_strs = [new_val for new_val in callback_list if '=' in new_val]
return command_str, update_strs, new_val_strs


def _default_handler(
self,
command: str, query: CallbackQuery, menu_history: List[str],
user_info: User, current_menu: str
):
"""
Handles callbacks that weren't handled by update or new val handlers,
these are either built-in callbacks or menu buttons
"""
if command == 'pass':
pass
elif command == 'exit':
query.delete_message()
elif command == 'back' and menu_history:
current_menu = menu_history.pop()
self.main_menu.operation(query.message, current_menu, user_info)
elif command == 'next_page' and self.current_updater is not None:
self.current_updater.current_page += 1
self.current_updater.operation(query.message, command, user_info)
elif command == 'prev_page' and self.current_updater is not None:
self.current_updater.current_page -= 1
self.current_updater.operation(query.message, command, user_info)
elif self.main_menu.operation(query.message, command, user_info):
```

```
menu_history.append(current_menu)
current_menu = command
return current_menu


def _update_menu(
self, current_menu: str, menu_history: List[str], id_: int
):
self.user_db.update_menu(current_menu, menu_history, id_)


def _new_val_handler(self, new_val_strs: List[str], chat_id: int):
for new_val_str in new_val_strs:
varname, new_val = new_val_str.split('=')
varname = varname.upper()

if not new_val.isdigit():
new_val = "'" + new_val + "'"
self.user_db.insert_or_replace(varname, chat_id, new_val)


def _update_handler(
self,
update_strs: List[str],
command_str: str,
menu_history: List[str],
current_menu: str,
query: CallbackQuery,
user_info: User
):
upd = update_strs[0]
update_strs.remove(upd)
upd = upd.replace('!', '')
callback = ';'.join(update_strs) + ';' + command_str
opts = []
if upd == 'group_id':
rows = self.schedule_db.execute("""SELECT DISTINCT GROUP_ID
FROM SCHEDULE""")

for row in rows:
id_ = row[0]
opts.append((id_, id_))
self.current_updater = MultiPageListMenu(
opts, upd.upper(), callback, True
)
elif upd == 'teacher_id':
rows = self.schedule_db.execute(
```

```
"""SELECT TEACHER_ID, FIRSTNAME, LASTNAME
FROM TEACHER"""
)
for row in rows:
id_ = row[0]
name = row[1] + ' ' + row[2]
opts.append((name, id_))
self.current_updater = MultiPageListMenu(
opts, upd.upper(), callback, True
)
elif upd == 'calendar_day':
today = datetime.date.today()
next_year = datetime.date(today.year + 1, today.month, today.day)
self.current_updater = CalendarMenu(
today, next_year, True, callback
)
elif upd == 'week_day':
opts = [
('Monday', '1'),
('Tuesday', '2'),
('Wednesday', '3'),
('Thursday', '4'),
('Friday', '5'),
('Saturday', '6'),
('Sunday', '7'),
]
self.current_updater = MultiPageListMenu(
opts, upd.upper(), callback, True
)
elif upd == 'class_id':
rows = self.schedule_db.execute(
"""SELECT CLASS_ID, NAME
FROM CLASS"""
)
for row in rows:
name = row[1]
id_ = row[0]
opts.append((name, id_))
self.current_updater = MultiPageListMenu(
opts, upd.upper(), callback, True
)
if current_menu not in menu_history:
menu_history.append(current_menu)
print()
print(f'upd={upd}')
print()
self.current_updater.operation(query.message, None, user_info)
```

“button.py”

```
"""
Button hierarchy for telegram bot menu
"""

from __future__ import annotations
from abc import ABC, abstractmethod
from typing import List, Callable
from telegram import InlineKeyboardMarkup, InlineKeyboardButton, Message
from user import User


class Button(ABC):
"""
Component from Composite Pattern,
either prints result of its handler or prints new menu when clicked
(unless its 'Exit' or 'Back', these just exit the menu or go to the previous
respectively)
"""


def callback_args(self) -> str:
"""returns additional data to be added to callback"""
return ''


@abstractmethod
def operation(self, message: Message, command: str, user: User) -> bool:
"""
Changes parameter message, performed if command == callback
True -> changed menu
Fales -> didnt change menu
None -> Bad command(Nothing happend)
"""


class LeafButton(Button):
"""
Leaf from Composite Pattern
Doesnt print new menu when clicked,
just calls handler and prints the result
arg2 is not None only if arg1 is not None
"""
def __init__(
self,
text: str, callback: str, parent: ListMenu,
handler: Callable[[User], str]=None,
```

```
arg1name: str=None, arg2name: str=None
):
"""added to parent automatically"""
self.text = text
self.callback = callback
self.handler = handler
self.arg1name = arg1name
self.arg2name = arg2name
if parent is not None:
parent.add(self)


def callback_args(self) -> str:
cb_args = ''
if self.arg1name is not None:
cb_args += ';!' + self.arg1name
if self.arg2name is not None:
cb_args += ';!' + self.arg2name
return cb_args


def operation(self, message: Message, command: str, user: User) -> bool:
"""
call for the handler
"""
if command == self.callback:
new_text = f'leaf button: {self.text}'
if self.handler is not None:
new_text = self.handler(user)
message.edit_text(new_text)
message.edit_reply_markup(reply_markup=None)
# raises BadRequest, but everything works as intended

return False


class Menu(Button):
"""
Composite from the pattern of the same name
Prints menu with all the children when clicked, also prints 'Exit' option,
and 'Back' if has parent, if 'Back' is clicked than parent is called(printed)
"""
def __init__(self, has_parent):
self._has_parent = has_parent


@property
```

```
def has_parent(self) -> bool:
"""Needed to check if 'Back' button is needed"""
return self._has_parent


@abstractmethod
def keyboard(self) -> List[List[InlineKeyboardButton]]:
"""keyboard base"""


def _add_nav_buttons(self, keyboard: List[List[InlineKeyboardButton]]):
if self.has_parent:
keyboard.append(
[InlineKeyboardButton('Back', callback_data='back')]
)
keyboard.append(
[InlineKeyboardButton('Exit', callback_data='exit')]
)


def _additional_buttons(self, keyboard: List[List[InlineKeyboardButton]]):
self._add_nav_buttons(keyboard)


def print(self, message: Message):
"""prints menu"""
keyboard = self.keyboard()
self._additional_buttons(keyboard)
markup = InlineKeyboardMarkup(keyboard)
message.edit_reply_markup(reply_markup=markup)


class ListMenu(Menu):
"""
Composite from the pattern of the same name
Prints menu with all the children when clicked, also prints 'Exit' option,
and 'Back' if has parent,
if 'Back' is clicked than parent is called(printed)
"""
def __init__(self, text: str, callback: str, parent: ListMenu=None) -> None:
Menu.__init__(self, parent is not None)
self._children: List[List[Button]] = [[]]
self.text = text
self.callback = callback
if parent is not None:
parent.add(self)
```

```
def add(self, button: Button) -> None:
"""adds button to the current row"""
self._children[-1].append(button)


def remove(self, button: Button) -> None:
"""removes button if its in the keyboard"""
for row in self._children:
row.remove(button)


def next_row(self):
"""go to the next row(if current row is empty does nothing)"""
if self._children[-1]:
self._children.append([])


def operation(self, message: Message, command: str, user: User) -> bool:
"""
Tries to handle command or delegates it to self._children
"""
if command == self.callback:
self.print(message)
return True
else:
for row in self._children:
for button in row:
if button.operation(message, command, user):
return True


def keyboard(self) -> List[List[InlineKeyboardButton]]:
"""Returns markup keyboard from children"""
return list(map(
lambda row: list(map(
lambda button: InlineKeyboardButton(
button.text, callback_data=(button.callback +
button.callback_args())
),
row
)),
self._children
))
```

“calendar_menu.py”

```
"""
Menu to choose calendar day
"""

import calendar
from datetime import date
from typing import List
from telegram import InlineKeyboardButton
from multipage_menu import MultiPageMenu


class CalendarMenu(MultiPageMenu):
"""
MENU TO CHOOSE CALENDAR DAY, callback is <CALENDAR_DAY=YYYY/MM/DD>
"""
def __init__(self,
start: date, finish: date, has_parent: bool, callback: str
):
MultiPageMenu.__init__(self, has_parent)
self.start = start
self.finish = finish
self.callback = callback

self.current_page = 0


def max_page(self) -> int:
return ((self.finish.year - self.start.year) * 12
+ (self.finish.month - self.start.month))


def keyboard(self) -> List[List[InlineKeyboardButton]]:
keyboard: List[List[InlineKeyboardButton]] = []

month = self.start.month + self.current_page - 1# 0 based
year = int(self.start.year + month / 12)
month = month % 12
month += 1# 1 based

keyboard.append([InlineKeyboardButton(
'%s %s' % (calendar.month_name[month], year), callback_data='pass'
)])
keyboard.append(self.weekdays_names())
self._add_month_days(keyboard, year, month)
return keyboard
```

```
@staticmethod
def weekdays_names():
"""
returns row of buttons with weekday names with empty callbacks('pass')
"""
row = []
for day in ["Mo", "Tu", "We", "Th", "Fr", "Sa", "Su"]:
row.append(InlineKeyboardButton(day, callback_data='pass'))
return row


def _add_month_days(
self, keyboard: List[List[InlineKeyboardButton]], year: int, month: int
):
my_calendar = calendar.monthcalendar(year, month)
for week in my_calendar:
row = []
for day in week:
if day == 0:
row.append(self.empty_button())
else:
row.append(self._day_button(year, month, day))
keyboard.append(row)


def _day_button(self, year: int, month: int, day: int):
"""returns button for non 0 day"""
return InlineKeyboardButton(
day,
callback_data='CALENDAR_DAY=%s/%s/%s' % (
year, month, day
) + ';' + self.callback
)
```

“main.py”

```
"""
MAIN
"""

import sqlite3
from telegram import Update
from telegram.ext import CallbackContext
from bot import Bot, Command
from button_manager import ButtonManager


if __name__ == '__main__':
users_db = sqlite3.connect('users.db', check_same_thread=False)
schedule_db = sqlite3.connect('schedule.db', check_same_thread=False)
commands = []
button_mgr = ButtonManager(users_db, schedule_db)


def start(update: Update, context: CallbackContext):
"""/start handler"""
descriptions = ''
for command in commands:
descriptions += '/' + command.keyword + ' - ' + command.description + '\n'
context.bot.send_message(
chat_id=update.effective_chat.id,
text=descriptions
)
commands.append(Command(
'start',
start,
'info about commands'
))


def menu_handler(update: Update, _context: CallbackContext):
"""
main menu
"""
msg = update.message.reply_text('Please choose: ')
button_mgr.print_main_menu(msg)
commands.append(Command(
'menu',
menu_handler,
'prints menu'
))
```

```
bot = Bot(
'1815999083:AAFClF7cEZq6IjXTxGNA07WQ5xLvZsKs6LY',
commands, None, button_mgr.button_handler#menu_manager.button_handler
)
bot.run()
```

“multipage_list_menu.py”

```
"""
ListMenus
"""
from typing import List, Tuple
from telegram import InlineKeyboardButton
from multipage_menu import MultiPageMenu

class MultiPageListMenu(MultiPageMenu):
"""
Forces user to fill in new data(choice from long lists)
"""
def __init__(
self,
options: List[Tuple[str, str]], arg_name: str, callback: str,
has_parent: bool,
options_per_page: int=10
):
MultiPageMenu.__init__(self, has_parent)
self.options = options
self.options_per_page = options_per_page
self.arg_name = arg_name
self.callback = callback

self._current_page = 0


def max_page(self) -> int:
return len(self.options) / self.options_per_page


def _add_page_nav_buttons(self, keyboard: List[List[InlineKeyboardButton]]):
if len(self.options) > self.options_per_page:
super()._add_page_nav_buttons(keyboard)


def keyboard(self) -> List[List[InlineKeyboardButton]]:
keyboard: List[List[InlineKeyboardButton]] = []
for i in range(self.options_per_page):
current_option_n = self.current_page * self.options_per_page + i
if current_option_n >= len(self.options):
break
current_option = self.options[current_option_n]
text = current_option[0]
callback = f'{self.callback};{self.arg_name}={current_option[1]}'

keyboard.append([InlineKeyboardButton(
```

```
text,
callback_data=callback
)])
return keyboard
```

“multipage_menu.py”

```
"""
Additional class to choose arguments for LeafButtons(update User state)
"""
from typing import List
from abc import abstractmethod
from telegram import Message, InlineKeyboardButton
from button import Menu
from user import User

class MultiPageMenu(Menu):
"""Multi page menu with scrolling"""
def __init__(self, has_parent):
Menu.__init__(self, has_parent)


@property
def has_parent(self) -> bool:
"""Needed to check if 'Back' button is needed"""
return self._has_parent


@property
def current_page(self) -> int:
"""current page"""
return self._current_page


@current_page.setter
def current_page(self, val: int):
self._current_page = val


@abstractmethod
def max_page(self) -> int:
"""maxpage"""


@abstractmethod
def keyboard(self) -> List[List[InlineKeyboardButton]]:
"""keyboard base"""


def operation(self, message: Message, command: str, user: User):
self.print(message)
```

```
def _additional_buttons(self, keyboard: List[List[InlineKeyboardButton]]):
self._add_page_nav_buttons(keyboard)
self._add_nav_buttons(keyboard)


def _add_nav_buttons(self, keyboard: List[List[InlineKeyboardButton]]):
if self.has_parent:
keyboard.append(
[InlineKeyboardButton('Back', callback_data='back')]
)
keyboard.append(
[InlineKeyboardButton('Exit', callback_data='exit')]
)


def _add_page_nav_buttons(self, keyboard: List[List[InlineKeyboardButton]]):
keyboard.append([self._prev_button(), self._next_button()])


def _next_button(self):
if self.current_page + 1 < self.max_page():
return InlineKeyboardButton(
'>',
callback_data='next_page'
)
else:
return self.empty_button()


def _prev_button(self):
if self.current_page > 0:
return InlineKeyboardButton(
'<',
callback_data='prev_page'
)
else:
return self.empty_button()


@staticmethod
def empty_button():
"""returns button with no text and 'pass' callback"""
return InlineKeyboardButton(' ', callback_data='pass')
```

```
"""
Read only manager for schedule database
"""
import datetime
from itertools import cycle, dropwhile, takewhile
from sqlite3 import Connection

class Schedule:
"""
Read only manager for schedule database
"""
def __init__(self, sql_conn: Connection):
self.sql_conn = sql_conn


def week_schedule(self, group_id: int, is_odd_week=None):
"""
schedule for week, whole/odd/even
"""
result = ''
for day in range(1, 6):
result += self.day_schedule(day, group_id, is_odd_week) + '\n'
return result


def day_schedule(self, day: int, group_id: int, is_odd_week=None):
"""
schedule for day, whole/odd/even
"""
odd_week_str = 'Whole week'
if is_odd_week is not None:
odd_week_str = 'Odd week' if is_odd_week else 'Even week'
result = self.day_from_int(day) + f'({odd_week_str})' + '\n'
for row in self._day_schedule(day, group_id, is_odd_week):
time = row[0]
is_lecture = row[1]
class_id = row[2]
class_odd_week = row[3]

classname = next(
self.sql_conn.execute(
'SELECT NAME from CLASS where CLASS_ID = '
+ str(class_id)
)
)[0]
```

```
type_ = 'lecture' if is_lecture else 'practice'
result += f'{time} - "{classname}"; {type_}'
if is_odd_week is None and class_odd_week is not None:
result += f'({"Odd" if class_odd_week == 1 else "Even"})'
result += '\n'
return result + '\n'


def _day_schedule(self, day: int, group_id: int, is_odd_week=None):
week_constraint = ''
if is_odd_week is not None:
boolean = 1 if is_odd_week else 0
week_constraint = f'AND (ODD_WEEK = {boolean} OR ODD_WEEK IS NULL)'

return self.sql_conn.execute(
'''SELECT TIME, IS_LECTURE, CLASS_ID, ODD_WEEK
from SCHEDULE
where DAY = %s
%s AND GROUP_ID = %s
ORDER BY TIME''' % (
day,
week_constraint,
group_id
)
)


@staticmethod
def day_from_int(day: int):
"""returns string with day name from int"""
days = [
'',# 1 based
'MONDAY',
'TUESDAY',
'WEDNESDAY',
'THURSDAY',
'FRIDAY',
'SATURDAY',
'SUNDAY',
]
return days[day]


def next_class(
self, day: int, hour: int, minute: int, group_id, is_odd_week=None
):
"""day, hour, minute is <current_time>"""
```

```
if 1 <= day <= 7:
counter = 0
def count(_):
nonlocal counter
counter += 1
return counter <= 7

for day_ in takewhile(
count,
dropwhile(lambda n: n < day, cycle(range(1, 8)))
):
# check one week forward(8th day is the same as today)
for row in self._day_schedule(day_, group_id, is_odd_week):
time = row[0]
is_lecture = row[1]
class_id = row[2]

time_hour, time_min = time.split(':')
time_hour = int(time_hour)
time_min = int(time_min)

if (
(time_hour == hour and time_min > minute)
or time_hour > hour
):
classname = next(
self.sql_conn.execute(
'SELECT NAME from CLASS where CLASS_ID = '
+ str(class_id)
)
)[0]

is_lecture_str = "lecture" if is_lecture else "practice"

return ('Your next class is %s(%s)\n' % (
classname,
is_lecture_str
) + f'on {self.day_from_int(day_)}, at {time}'
)


def current_class(
self, day: int, hour: int, minute: int, group_id: int, is_odd_week=None
):
"""returns stringified info about current class
day, hour, minute is <current_time>"""
if 1 <= day <= 7:
```

```
for row in self._day_schedule(day, group_id, is_odd_week):
time = row[0]
is_lecture = row[1]
class_id = row[2]

time_hour, time_min = time.split(':')
time_hour = int(time_hour)
time_min = int(time_min)

time_now = datetime.time(hour, minute)
time_class_start = datetime.time(time_hour, time_min)
time_class_finish = time_class_start + datetime.timedelta(
minutes=80
)

if time_class_start < time_now < time_class_finish:
classname = next(
self.sql_conn.execute(
'SELECT NAME from CLASS where CLASS_ID = '
+ str(class_id)
)
)[0]

is_lecture_str = "lecture" if is_lecture else "practice"

return ('Your next current class is %s(%s)\n' % (
classname,is_lecture_str
))
return "You don't have a class rigth now"


def teacher_info(self, id_):
"""Stringified info about teacher by their id"""
row = next(self.sql_conn.execute(
f"""SELECT FIRSTNAME, LASTNAME
FROM TEACHER
WHERE TEACHER_ID = {id_}"""
))
result = f'{row[0]} {row[1]}' + '\n'
rows = self.sql_conn.execute(
f"""SELECT NAME,
LECTURER_ID,
INSTRUCTOR1_ID,
INSTRUCTOR2_ID
FROM CLASS
WHERE LECTURER_ID = {id_} OR
INSTRUCTOR1_ID = {id_} OR
```

```
INSTRUCTOR2_ID = {id_}"""
)

for row in rows:
subresult = ''
classname = row[0]
lecturer_id = row[1]
ins1_id = row[2]
ins2_id = row[3]

if lecturer_id == id_:
subresult += 'is lecturer'
if ins1_id == id_ or ins2_id == id_:
if subresult != '':
subresult += ' and instructor'
else:
subresult += 'is instructor'
subresult += f' for "{classname}"'
result += subresult + '\n'
return result


def class_info(self, id_):
"""Stringified info about class by id"""
row = next(self.sql_conn.execute(
f"""SELECT NAME,
LECTURER_ID,
INSTRUCTOR1_ID,
INSTRUCTOR2_ID,
LECTURE_ROOM_ID,
ROOM1_ID,
ROOM2_ID
FROM CLASS
WHERE CLASS_ID = {id_}"""
))
name = row[0]
lecturer_id = row[1]
ins1_id = row[2]
ins2_id = row[3]
lec_room = row[4]
room1 = row[5]
room2 = row[6]

lecturer = next(self.sql_conn.execute(
f"""SELECT FIRSTNAME, LASTNAME
FROM TEACHER
WHERE TEACHER_ID = {lecturer_id}"""
```

```
))
ins1 = next(self.sql_conn.execute(
f"""SELECT FIRSTNAME, LASTNAME
FROM TEACHER
WHERE TEACHER_ID = {ins1_id}"""
))
ins2 = None
if ins2_id is not None:
ins2 = next(self.sql_conn.execute(
f"""SELECT FIRSTNAME, LASTNAME
FROM TEACHER
WHERE TEACHER_ID = {ins2_id}"""
))

result = ''
result += name + '\n'
result += f'lecturer: {lecturer[0]} {lecturer[1]}' + '\n'
result += f'instructor 1: {ins1[0]} {ins1[1]}' + '\n'
if ins2 is not None:
result += f'instructor 2: {ins2[0]} {ins2[1]}' + '\n'
result += f'lecture room: {lec_room}' + '\n'
result += f'room 1: {room1}' + '\n'
if room2 is not None:
result += f'room 2: {room2}' + '\n'
return result


def group_list(self, id_: int, subgroup: int=None):
"""stringified list group"""
subgroup_constraint = ''
if subgroup is not None:
subgroup_constraint = f' AND SUBGROUP = {subgroup}'
rows = self.sql_conn.execute(
f"""SELECT FIRSTNAME,
LASTNAME,
SUBGROUP
FROM STUDENT
WHERE GROUP_ID = {id_}""" + subgroup_constraint)

result = ''
for row in rows:
name = row[0] + ' ' + row[1]
subgr = row[2]

result += name
if subgroup is None:
result += f'(subgroup: {subgr})'
```

```
result += '\n'
return result
```

“user_db_manager.py”

```
"""
Manager for user info
"""
from sqlite3 import Connection
from typing import List
from user import User

class UserDbManager:
"""
Manager for user info
"""
def __init__(self, sql_conn: Connection):
self.sql_conn = sql_conn


def menu_data(self, id_: int) -> (List[str], str):
"""returns tuple (menu_history, current_menu)"""
row = next(self.sql_conn.execute(f"""SELECT MENU_HISTORY, CURRENT_MENU
FROM USER
WHERE ID = {id_}"""
))
menu_history_str = row[0]
menu_history = (menu_history_str or '').split(';')
current_menu = row[1]
return menu_history, current_menu


def update_menu(self, current_menu: str, menu_history: List[str], id_: int):
"""inserts or replaces info about users state"""
menu_history_str = ';'.join(menu_history)
if self.is_row_exists(id_):
self.sql_conn.execute(
f"""UPDATE USER
SET CURRENT_MENU = '{current_menu}',
MENU_HISTORY = '{menu_history_str}'
WHERE ID = {id_}"""
)
else:
self.sql_conn.execute(f"""REPLACE INTO USER
(CURRENT_MENU, MENU_HISTORY, ID)
VALUES
('{current_menu}', '{menu_history_str}', {id_})"""
)
self.sql_conn.commit()
```

```
def is_row_exists(self, id_: int):
"""check if row with id exists"""
row = next(self.sql_conn.execute(
f"""SELECT * FROM USER WHERE ID = {id_}"""
), None)
return row is not None


def insert_or_replace(self, varname: str, id_: int, new_val: str):
"""inserts or updates varname with new_val"""
if self.is_row_exists(id_):
self.sql_conn.execute(
f"""UPDATE USER
SET {varname} = {new_val}
WHERE ID = {id_}"""
)
else:
self.sql_conn.execute(f"""REPLACE INTO USER
(ID, {varname})
VALUES
({id_}, {new_val})"""
)
self.sql_conn.commit()


def user(self, id_):
"""returns user by their id"""
row = next(self.sql_conn.execute("""SELECT GROUP_ID,
TEACHER_ID,
CALENDAR_DAY,
WEEK_DAY,
CLASS_ID
FROM USER
WHERE ID = %s""" % id_
))
return User(row[0], row[1], row[2], row[3], row[4])
```

“user.py”

```
"""
user dataclass
"""

import datetime
from dataclasses import dataclass

@dataclass
class User:
"""
Easier access to user specific data(args for commands)
"""
group_id: int
teacher_id: int
calendar_day: str
week_day: int
class_id: int


def weekday_from_calendar_day(self):
"""1 based day from calendar day"""
return self.date_from_calendar_day().weekday() + 1


def is_odd_week_from_calendar_day(self):
"""is week odd(year wise) from calendar day"""
return self.date_from_calendar_day().isocalendar().week % 2 == 1


def date_from_calendar_day(self):
"""datetime.date object from calendar_day"""
year, month, day = self.calendar_day.split('/')
return datetime.date(int(year), int(month), int(day))
```