МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ "ЛЬВІВСЬКА ПОЛІТЕХНІКА"

Кафедра інформаційних систем та мереж

Лабораторна робота №3

з дисципліни

«Спеціалізовані мови програмування»

на тему

«Розробка ASCII ART генератора для візуалізації текстових даних»

Виконав:

ст. гр. РІ-32

Олег ВОЛОДЬКО

Прийняв:

доц. каф. ІСМ

Сергій ЩЕРБАК

Львів - 2024

**Мета**

Створення додатка Генератора ASCII-арту.

**Хід виконання роботи**

**Завдання 1: Введення користувача**

Створено Python-програму, яка приймає введення користувача для слова або фрази, яку треба перетворити в ASCII-арт.

**Завдання 2: Бібліотека ASCII-арту**

Інтегрувано бібліотеку ASCII-арту (наприклад, pyfiglet або art) у програму для генерації ASCII-арту з введення користувача.

**Завдання 3: Вибір шрифту**

Дозволено користувачам вибирати різні стилі шрифтів для свого ASCII-арту. Надано список доступних шрифтів та дозволити їм вибрати один.

**Завдання 4: Колір тексту**

Реалізувано опцію вибору користувачем кольору тексту для їхнього ASCII-арту. Підтримано основний вибір кольорів (наприклад, червоний, синій, зелений).

**Завдання 5: Форматування виводу**

Створений ASCII-арт правильно відформатований та вирівнюється на екрані для зручності читання.

**Завдання 6: Збереження у файл**

Додано функціональність для збереження створеного ASCII-арту у текстовому файлі, щоб користувачі могли легко завантажувати та обмінюватися своїми творіннями.

**Завдання 7: Розмір ARTу**

Дозволено користувачам вказувати розмір (ширина і висота) ASCII-арту, який вони хочуть створити. Масштабувати текст відповідно.

**Завдання 8: Вибір символів**

Дозволено користувачам вибирати символи, які вони хочуть використовувати для створення ASCII-арту (наприклад, '@', '#', '\*', тощо).

**Завдання 9: Функція попереднього перегляду**

Реалізувано функцію попереднього перегляду, яка показує користувачам попередній перегляд їхнього ASCII-арту перед остаточним збереженням.

**Завдання 10: Інтерфейс, зрозумілий для користувача**

Створено зручний для користувача інтерфейс командного рядку для додатка, щоб зробити його інтуїтивно зрозумілим та легким у використанні.

**Код програмного продукту**

//AsciiFabric.py

from shared.classes.DictJsonDataAccess import DictJsonDataAccess

from config.settings\_paths import settings\_path\_lab3

from shared.classes.FolderDataAccess import FolderDataAccess

from labs.lab3.bll.AsciiController import AsciiController

from labs.lab3.bll.ColoramaPainter import ColoramaPainter

from labs.lab3.bll.PyfigletGenerator import PyfigletGenerator

from labs.lab3.ui.AsciiSettings import AsciiSettingsUI

from labs.lab3.ui.AsciiMenu import AsciiMenu

from shared.interfaces.UIInterface import UIInterface

from labs.lab4.bll.CustomPainter import CustomPainter

from labs.lab4.bll.CustomGenerator import CustomGenerator

class AsciiFabric:

"""

AsciiFabric class is a facade for generating ASCII art and managing user interface.

Attributes:

\_\_ascii\_ui (UIInterface): Interface to interact with ASCII art generation menu.

Methods:

\_\_init\_\_(generator, coloring): Initializes the AsciiFabric with a specific generator and coloring tool.

show(): Displays the ASCII art generation menu.

pyfiglet(): Class method to instantiate AsciiFabric with PyfigletGenerator and ColoramaPainter.

custom(): Class method to instantiate AsciiFabric with CustomGenerator and CustomPainter.

"""

def \_\_init\_\_(self, generator=PyfigletGenerator(), coloring=ColoramaPainter()):

settings\_access = DictJsonDataAccess(settings\_path\_lab3)

\_\_arts\_folder = settings\_access.get("\_\_arts\_folder")

settings\_ui: UIInterface = AsciiSettingsUI()

ascii\_ui: UIInterface = AsciiMenu(settings\_ui, arts\_folder= \_\_arts\_folder)

arts\_access = FolderDataAccess(\_\_arts\_folder, True, ".txt")

controller = AsciiController(

generator,

coloring,

arts\_access,

settings\_access,

)

settings\_ui.set\_controller(controller)

ascii\_ui.set\_controller(controller)

self.\_\_ascii\_ui = ascii\_ui

def show(self):

self.\_\_ascii\_ui.show()

@classmethod

def pyfiglet(cls):

generator = PyfigletGenerator()

coloring = ColoramaPainter()

return cls(generator, coloring)

@classmethod

def custom(cls):

generator = CustomGenerator()

coloring = CustomPainter()

return cls(generator, coloring)

from shared.classes.AsciiGenerator import AsciiGenerator

from shared.interfaces.PaintTextInterface import PaintTextInterface

from shared.classes.KeyDataAccess import KeyDataAccess

class AsciiController:

"""

AsciiController is responsible for managing ASCII art generation settings and processes.

It interacts with the ASCII art generator, coloring interface, and settings storage.

:param generator: Object responsible for generating ASCII art.

:type generator: AsciiGenerator

:param coloring: Interface for coloring ASCII art text.

:type coloring: PaintTextInterface

:param arts\_access: Access interface for art storage.

:type arts\_access: KeyDataAccess

:param settings\_access: Access interface for settings storage.

:type settings\_access: KeyDataAccess

"""

def \_\_init\_\_(

self,

generator: AsciiGenerator,

coloring: PaintTextInterface,

arts\_access: KeyDataAccess,

settings\_access: KeyDataAccess,

):

self.\_generator = generator

self.\_coloring = coloring

self.\_arts\_access = arts\_access

self.\_settings = settings\_access

self.art = None

def get\_fonts(self):

return self.\_generator.get\_fonts()

def get\_font(self):

return self.\_settings.get("font")

def is\_font\_correct(self):

current\_font = self.get\_font()

fonts = self.get\_fonts()

if current\_font in fonts:

return True

else:

return False

def set\_font(self, font):

if not font in self.get\_fonts():

raise ValueError("Font not in fonts list")

self.\_settings.set("font", font)

def get\_bright\_symbol(self):

return self.\_settings.get("bright\_symbol")

def get\_empty\_symbol(self):

return self.\_settings.get("empty\_symbol")

def set\_bright\_symbol(self, bright\_symbol):

if len(bright\_symbol) != 1:

raise ValueError("String with wrong lenghts, can't set symbol")

self.\_settings.set("bright\_symbol", bright\_symbol)

def set\_empty\_symbol(self, empty\_symbol):

if len(empty\_symbol) != 1:

raise ValueError("String with wrong lenghts, can't set symbol")

self.\_settings.set("empty\_symbol", empty\_symbol)

def set\_is\_symbols\_replace(self, is\_symbols\_replace):

if is\_symbols\_replace not in [True, False]:

raise ValueError("Try set non-bool value to using symbols")

self.\_settings.set("is\_symbols\_replace", is\_symbols\_replace)

def get\_is\_symbols\_replace(self):

return self.\_settings.get("is\_symbols\_replace")

def get\_colors(self):

return self.\_coloring.get\_colors()

def get\_color(self):

return self.\_settings.get("color")

def set\_color(self, color):

if not color in self.get\_colors():

raise ValueError("Color not in colors list")

self.\_settings.set("color", color)

def get\_min\_width(self):

font = self.get\_font()

return self.\_generator.get\_font\_char\_width(font)

def get\_min\_height(self):

font = self.get\_font()

return self.\_generator.get\_font\_char\_height(font)

def is\_font\_support\_line\_break(self):

font = self.get\_font()

return self.\_generator.is\_font\_break\_lines(font)

def get\_char\_limit(self):

font\_height = self.get\_min\_height()

font\_width = self.get\_min\_width()

height\_limit = self.get\_height()

width\_limit = self.get\_width()

if font\_height > height\_limit:

return 0

if font\_width > width\_limit:

return 0

symbols\_in\_row = int(width\_limit / font\_width)

is\_line\_breaks = self.get\_is\_line\_breaks()

is\_font\_breaks\_line = self.is\_font\_support\_line\_break()

if not is\_line\_breaks or not is\_font\_breaks\_line:

return symbols\_in\_row

cols = int(height\_limit / font\_height)

return symbols\_in\_row \* cols

def get\_max\_possible\_width(self):

return self.\_settings.get("max\_width")

def get\_max\_possible\_height(self):

return self.\_settings.get("max\_height")

def get\_width(self):

return self.\_settings.get("width")

def get\_height(self):

return self.\_settings.get("height")

def set\_height(self, height):

try:

height = int(height)

except:

raise TypeError("Wrong type of width")

max\_height = self.get\_max\_possible\_height()

min\_height = self.get\_min\_height()

if not min\_height <= height <= max\_height:

raise ValueError("Height not in diapason")

self.\_settings.set("height",height)

def set\_width(self, width):

try:

width = int(width)

except:

raise TypeError("Wrong type of width")

max\_width = self.get\_max\_possible\_width()

min\_width = self.get\_min\_width()

if not min\_width <= width <= max\_width:

raise ValueError("Width not in diapason")

self.\_settings.set("width", width)

def get\_alignment(self):

return self.\_settings.get("alignment")

def set\_alignment(self, alignment):

options = ["left", "right", "center"]

if alignment not in options:

raise ValueError("Wrong alignment")

self.\_settings.set("alignment", alignment)

def get\_is\_line\_breaks(self):

return self.\_settings.get("is\_line\_breaks")

def set\_is\_line\_breaks(self, is\_line\_breaks):

if is\_line\_breaks not in [True, False]:

raise ValueError("Try set non-bool value to line breaking")

self.\_settings.set("is\_line\_breaks", is\_line\_breaks)

def is\_line\_broken(self, art: str):

is\_symbol\_replace = self.get\_is\_symbols\_replace()

bright\_symbol = self.get\_bright\_symbol() if is\_symbol\_replace else None

is\_symbol\_detected = False

is\_gap\_detected = False

for line in art.splitlines():

line\_has\_content = (

bright\_symbol in line if is\_symbol\_replace else line.strip()

)

if line\_has\_content:

is\_symbol\_detected = True

if is\_gap\_detected:

return True

else:

if is\_symbol\_detected:

is\_gap\_detected = True

return False

def is\_art\_exist(self):

return bool(self.art)

def get\_art(self):

return self.art

def set\_art(self, art):

if not art:

raise ValueError("No art to set")

self.art = art

def save\_art(self, name):

art = self.get\_art()

if not art:

raise ValueError("No art to save")

if not self.is\_art\_allowed(art):

raise ValueError("Too wide")

self.\_arts\_access.set(name, art)

def remove\_color(string):

result = []

in\_ansi\_sequence = False

for char in string:

if char == "\x1b":

in\_ansi\_sequence = True

elif in\_ansi\_sequence and char == "m":

in\_ansi\_sequence = False

elif not in\_ansi\_sequence:

result.append(char)

return "".join(result)

def is\_art\_allowed(self, art: str):

lines = art.splitlines()

if not lines:

return False

art\_height = len(lines)

art\_weight = len(lines[1])

height\_limit = self.get\_height()

width\_limit = self.get\_width()

if art\_height > height\_limit:

return False

if art\_weight > width\_limit:

return False

is\_line\_breaks = self.get\_is\_line\_breaks()

if not is\_line\_breaks:

is\_line\_broken = self.is\_line\_broken(art)

if is\_line\_broken:

return False

return True

def generate(self, text, width=None):

font = self.get\_font()

if width == None:

width = self.get\_width()

kwargs = {"font": font, "width": width}

art = self.\_generator.generate(text, \*\*kwargs)

return art

def replace(self, art):

bright = self.get\_bright\_symbol()

empty = self.get\_empty\_symbol()

return self.\_generator.replace(art, bright, empty)

def paint(self, art):

color = self.get\_color()

return self.\_coloring.paint(art, color)

def cut(self, text, width, height):

lines = text.split("\n")

lines = lines[0: height - 1]

lines\_to\_return = []

for line in lines:

lines\_to\_return.append(line[0: width - 1])

return "\n".join(lines\_to\_return)

def justify(self, text):

alignment = self.get\_alignment()

width = self.get\_width()

justified\_text = self.\_generator.alignment\_text(text, alignment, width)

return justified\_text

def create\_art(self, text):

art = self.generate(text)

is\_allowed = self.is\_art\_allowed(art)

if not is\_allowed:

raise ValueError("Art breaks the rules")

justified\_art = self.justify(art)

is\_replace = self.get\_is\_symbols\_replace()

if is\_replace:

justified\_art = self.replace(justified\_art)

self.set\_art(justified\_art)

painted\_art = self.paint(justified\_art)

return painted\_art

def create\_cut\_art(self, text):

is\_line\_breaks = self.get\_is\_line\_breaks()

if not is\_line\_breaks:

art = self.generate(text, 999)

else:

art = self.generate(text)

width = self.get\_width()

height = self.get\_height()

cutted\_art = self.cut(art, width, height)

justified\_art = self.justify(cutted\_art)

is\_replace = self.get\_is\_symbols\_replace()

if is\_replace:

justified\_art = self.replace(justified\_art)

painted\_art = self.paint(justified\_art)

return painted\_art

def get\_settings\_info(self):

return self.\_settings.\_\_str\_\_()

from shared.interfaces.PaintTextInterface import PaintTextInterface

from colorama import Fore, Style

class ColoramaPainter(PaintTextInterface):

"""

ColoramaPainter is a class that implements the PaintTextInterface. It provides text painting capabilities using color codes from the `colorama` library.

Attributes:

-----------

color\_map : dict

A mapping between color names (as keys) and their corresponding `colorama` color codes (as values).

Methods:

--------

paint(cls, text, color):

Paints the given text with the specified color if it's supported.

get\_colors(cls):

Returns a list of all supported colors.

"""

color\_map = {

"red": Fore.RED,

"green": Fore.GREEN,

"yellow": Fore.YELLOW,

"blue": Fore.BLUE,

"magenta": Fore.MAGENTA,

"cyan": Fore.CYAN,

"white": Fore.WHITE,

"light\_gray": "\x1b[37m",

"dark\_gray": "\x1b[90m",

"black": Fore.BLACK,

"default": Style.RESET\_ALL,

}

@classmethod

def paint(cls, text, color):

if color in cls.color\_map:

return f"{cls.color\_map[color]}{text}{Style.RESET\_ALL}"

else:

raise ValueError(f"Color '{color}' is not supported.")

@classmethod

def get\_colors(cls):

return list(cls.color\_map.keys())

from shared.classes.AsciiGenerator import AsciiGenerator

from pyfiglet import FigletFont, figlet\_format

class PyfigletGenerator(AsciiGenerator):

"""

PyfigletGenerator class for generating ASCII art using the pyfiglet library.

Methods

-------

generate(data, \*\*kwargs):

Generate ASCII art for the given data using specified fonts and settings.

is\_font\_break\_lines(cls, font):

Determines if a given font will break lines for texts that are too wide.

get\_fonts():

Retrieves a list of available fonts in the pyfiglet library.

get\_font\_char\_height(cls, font\_name):

Returns the height of characters for the specified font.

get\_font\_char\_width(cls, font\_name):

Returns the width of characters for the specified font.

"""

@staticmethod

def generate(data, \*\*kwargs):

return figlet\_format(data, \*\*kwargs)

@classmethod

def is\_font\_break\_lines(cls, font):

test\_line = "testlineforlinebreakingtomuchtofitinsinglelinethatforshure"

kwargs = {"font": font, "width": 50}

art = cls.generate(test\_line, \*\*kwargs)

font\_height = cls.get\_font\_char\_height(font)

rows\_amound = len(art.splitlines())

if rows\_amound > font\_height:

return True

else:

return False

@staticmethod

def get\_fonts():

return FigletFont.getFonts()

@classmethod

def get\_font\_char\_height(cls, font\_name):

try:

font\_data = FigletFont.preloadFont(font\_name)

header\_line = font\_data.splitlines()[0]

header\_parts = header\_line.split()

header\_height = int(header\_parts[2])

except Exception as e:

print(f"Error retrieving font height: {e}")

return None

highest\_symbols = "ILQWIONCZ"

kwargs = {"font": font\_name}

max\_height = max(

len(cls.generate(char, \*\*kwargs).splitlines()) for char in highest\_symbols

)

return max(header\_height, max\_height)

@classmethod

def get\_font\_char\_width(cls, font\_name):

widest\_symbols = "mwGMW"

kwargs = {"font": font\_name}

max\_width = max(

len(cls.generate(char, \*\*kwargs).splitlines()[0]) for char in widest\_symbols

)

return max\_width

from shared.classes.MenuBuilder import MenuBuilder

from shared.interfaces.UIInterface import UIInterface

from shared.classes.Input import StringInput

from labs.lab3.bll.AsciiController import AsciiController

class AsciiMenu(UIInterface):

"""

AsciiMenu class inheriting from UIInterface, responsible for displaying and interacting with the ASCII art generation menu.

Attributes:

\_\_controller (AsciiController): The controller handling ASCII art creation and storage.

\_\_settings\_ui (UIInterface): Interface for the settings menu.

\_\_arts\_folder (str): Directory where art files are stored.

Methods:

\_\_init\_\_(settings\_ui, arts\_folder=None, controller=None):

Initializes the AsciiMenu instance with settings UI, arts folder, and controller.

set\_controller(controller):

Sets the controller for ASCII art generation and rebuilds the menu.

show():

Displays the ASCII art menu.

\_\_menu\_build():

Constructs and returns the complete menu for the ASCII art generator.

show\_settings():

Displays the settings UI.

make\_art():

Handles the creation of ASCII art based on user input.

save\_art():

Saves the generated ASCII art to a text file.

get\_art():

Retrieves and returns the currently generated ASCII art, if any.

"""

def \_\_init\_\_(self, settings\_ui: UIInterface, arts\_folder = None, controller: AsciiController = None):

self.\_\_controller = controller

self.\_\_settings\_ui = settings\_ui

self.\_\_arts\_folder = arts\_folder

def set\_controller(self, controller: AsciiController):

self.\_\_controller = controller

self.\_\_menu = self.\_\_menu\_build()

def show(self):

self.\_\_menu.show()

def \_\_menu\_build(self):

return (

MenuBuilder()

.set\_title("Ascii generator")

.set\_warning("Wrong input!")

.set\_input\_text("Choose: ")

.set\_dynamic\_title(self.get\_art)

.add\_option("1", "1. Make art\n", self.make\_art)

.add\_option("2", "2. Save art\n", self.save\_art)

.add\_option("3", "3. Settings", self.show\_settings)

.add\_stop\_options(["0", "Exit", "exit"], "0. Exit")

.build()

)

def show\_settings(self):

self.\_\_settings\_ui.show()

def make\_art(self):

is\_font\_correct = self.\_\_controller.is\_font\_correct()

if not is\_font\_correct:

print("Can't generate example, no such font\n CHANGE FONT TO AVAILABLE")

return

char\_width = self.\_\_controller.get\_char\_limit()

if char\_width == 0:

print("Limits too low, can't create even 1 symbol, change settings")

return

message = (

f"Make art up to {char\_width} chars"

+ "\nIf want more change width, height, font or line breaking in settings"

+ "\n Input text : "

)

input = StringInput().input(message, [1, char\_width], "Too long")

art = self.\_\_controller.generate(input)

if len(art) < 1:

try:

int(input)

print("Art empty, problem might be with numbers")

except:

print("Art empty, problem might be with punctuation marks")

return

if not art:

print("Art can't created")

return

art = self.\_\_controller.create\_art(input)

def save\_art(self):

art = self.\_\_controller.get\_art()

if not art:

return

print(art)

message = "Write name: "

limit = 30 # hardcode variable of key lenght

input = StringInput().input(message, [1, limit], "Too long")

self.\_\_controller.save\_art(input)

print(f"Save your art in {self.\_\_arts\_folder}/{input}.txt")

def get\_art(self):

if self.\_\_controller.is\_art\_exist():

return "Current art \n" + self.\_\_controller.get\_art()

else:

return "No current art"

from shared.classes.MenuBuilder import MenuBuilder

from shared.classes.Input import (

VariantsInput,

BoolInput,

StringInput,

NumberBetweenInput,

)

from labs.lab3.bll.AsciiController import AsciiController

class AsciiSettingsUI:

"""

AsciiSettingsUI

This class represents the user interface for ASCII art settings. It is responsible for displaying various settings menus and options to the user and obtaining their input to configure the ASCII art generation.

Methods

-------

\_\_init\_\_(self, controller: AsciiController = None)

Initializes the AsciiSettingsUI instance with an optional AsciiController instance.

set\_controller(self, controller: AsciiController)

Sets the AsciiController instance for this UI and rebuilds the main menu.

show(self)

Displays the main menu to the user.

\_\_menu\_build(self)

Builds and configures the main settings menu.

set\_symbols(self)

Displays the menu for configuring symbol replacements used in ASCII art.

get\_symbols\_replacement\_info(self)

Retrieves current symbol replacement settings from the controller.

set\_alignment(self)

Allows the user to set text alignment for the ASCII art.

set\_bright\_symbol(self)

Allows the user to set the symbol used for bright areas in the ASCII art.

set\_empty\_symbol(self)

Allows the user to set the symbol used for empty areas in the ASCII art.

set\_is\_replace\_symbols(self)

Allows the user to enable or disable symbol replacement.

set\_color(self)

Allows the user to set the color for the ASCII art.

set\_font(self)

Allows the user to set the font used in the ASCII art.

set\_width(self)

Allows the user to set the width of the ASCII art.

set\_height(self)

Allows the user to set the height of the ASCII art.

set\_line\_breaking(self)

Allows the user to enable or disable line breaking (word wrapping) during ASCII art generation.

see\_example(self)

Provides an example of the current settings applied to a sample text.

"""

def \_\_init\_\_(self, controller: AsciiController = None):

self.\_\_controller = controller

def set\_controller(self, controller: AsciiController):

self.\_\_controller = controller

self.\_\_menu = self.\_\_menu\_build()

def show(self):

self.\_\_menu.show()

def \_\_menu\_build(self):

return (

MenuBuilder()

.set\_title("Setting for ascii")

.set\_input\_text("Choose: ")

.set\_warning("No such setting")

.set\_dynamic\_title(self.see\_example)

.add\_option(

"1", "1. Replacing symbols\n", self.set\_symbols

)

.add\_option("2", "2. Change font\n", self.set\_font)

.add\_option("3", "3. Change color\n", self.set\_color)

.add\_option("4", "4. Change width\n", self.set\_width)

.add\_option("5", "5. Change height\n", self.set\_height)

.add\_option(

"6", "6. Change alignment\n", self.set\_alignment

)

.add\_option(

"7", "7. Change line breaking (word wrapping)\n", self.set\_line\_breaking

)

.add\_stop\_options(["0", "Exit", "exit"], "0. Exit")

.build()

)

def set\_symbols(self):

symbols\_menu = (

MenuBuilder()

.set\_title("Replacing symbols")

.set\_input\_text("Choose: ")

.set\_warning("No such setting")

.set\_dynamic\_title(self.get\_symbols\_replacement\_info)

.add\_option\_without\_attributes(

"1", "1. Set using this symbols\n", self.set\_is\_replace\_symbols

)

.add\_option\_without\_attributes(

"2", "2. Set bright symbols\n", self.set\_bright\_symbol

)

.add\_option\_without\_attributes(

"3", "3. Set empty symbol\n", self.set\_empty\_symbol

)

.add\_stop\_options(["0", "Exit", "exit"], "0. Exit")

.build()

)

symbols\_menu.show()

def get\_symbols\_replacement\_info(self):

bright = self.\_\_controller.get\_bright\_symbol()

empty = self.\_\_controller.get\_empty\_symbol()

is\_replace\_symbols = self.\_\_controller.get\_is\_symbols\_replace()

return (

f"Current symbols:"

+ f"\n bright: {bright}"

+ f"\n empty: {empty}"

+ f"\n replacing: {is\_replace\_symbols}"

)

def set\_alignment(self):

options = ["left", "right", "center"]

options\_str = "/".join(options)

message = f"Choose alignment ({options\_str}): "

result = VariantsInput().input(message, options, "Wrong alignment")

self.\_\_controller.set\_alignment(result)

def set\_bright\_symbol(self):

message = "Bright symbol: "

result = StringInput.input(message, [1, 1])

self.\_\_controller.set\_bright\_symbol(result)

def set\_empty\_symbol(self):

message = "Empty symbol: "

result = StringInput.input(message, [1, 1])

self.\_\_controller.set\_empty\_symbol(result)

def set\_is\_replace\_symbols(self):

message = "Replace (y/n)? "

result = BoolInput.default(message)

self.\_\_controller.set\_is\_symbols\_replace(result)

def set\_color(self):

colors = self.\_\_controller.get\_colors()

colors\_str = " ,".join(colors)

color = VariantsInput().input(

f"Colors: {colors\_str}\n Choose color:", colors, "Wrong color!"

)

self.\_\_controller.set\_color(color)

def set\_font(self):

fonts = self.\_\_controller.get\_fonts()

fonts\_str = ""

in\_row = 5

last = 0

for index in range(0, len(fonts), in\_row):

row = ", ".join(fonts[index : index + in\_row])

fonts\_str += "\n" + row

font = VariantsInput.input(fonts\_str + "\nChoose font: ", fonts)

self.\_\_controller.set\_font(font)

def set\_width(self):

min\_width = self.\_\_controller.get\_min\_width()

max\_width = self.\_\_controller.get\_max\_possible\_width()

cur\_width = self.\_\_controller.get\_width()

message = (

f"Current width: {cur\_width}. Range: {min\_width}-{max\_width}\n Set width: "

)

width = NumberBetweenInput().input(

message, [min\_width, max\_width], "Wrong width"

)

self.\_\_controller.set\_width(width)

def set\_height(self):

min\_height = self.\_\_controller.get\_min\_height()

max\_height = self.\_\_controller.get\_max\_possible\_height()

cur\_height = self.\_\_controller.get\_height()

message = f"Current height: {cur\_height}. Range: {min\_height}-{max\_height}\n Set height: "

height = NumberBetweenInput().input(

message, [min\_height, max\_height], "Wrong height"

)

self.\_\_controller.set\_height(height)

def set\_line\_breaking(self):

is\_font\_support = self.\_\_controller.is\_font\_support\_line\_break()

if is\_font\_support:

supporting = "current font support"

else:

supporting = "current font not support"

message = f"Break line/wrap word ({supporting}) (y/n)? "

result = BoolInput.default(message)

self.\_\_controller.set\_is\_line\_breaks(result)

def see\_example(self):

is\_font\_correct = self.\_\_controller.is\_font\_correct()

if not is\_font\_correct:

return "Can't generate example, no such font\n CHANGE FONT TO AVAILABLE"

example\_text = "Settings for art"

settings\_info = self.\_\_controller.get\_settings\_info()

art = self.\_\_controller.create\_cut\_art(example\_text)

return settings\_info + f"\n '{example\_text}' string example\n" + art

**Висновки**

Я створив універсальний Генератор ASCII-арту, який дозволить користувачам налаштовувати свої творіння з різними шрифтами, кольорами, розмірами та символами. Я отримав практичний досвід роботи з введенням користувача, зовнішніми бібліотеками, роботою з файлами та дизайном інтерфейсу користувача в Python.