for this chatbox, i would like to firstly provide you some context for answering some questions. plz bear in mind that this chat only for answer questions for GUI.

this is the introduction of requirement ,plz first make a summery for that, and again, bear in mind that it is a context. and next i will also add some support code for context too.

In Assignment 2, you implemented a text-based game of Fancy Sokoban using the Model-View-Controller (MVC) design pattern.

In Assignment 3, the text-based interface will be replaced with a graphical user interface (GUI) using tkinter.

The final game's appearance might vary based on the operating system, as shown in Figure 1.

Unlike previous assignments where user interactions were managed through input calls, Assignment 3 will handle interactions through events like key-presses and mouse clicks.

The solution must adhere to the Apple MVC design pattern discussed in lectures.

The modeling classes from Assignment 2 can be reused for this graphical version due to the MVC pattern.

Enhanced modeling classes have been provided for this assignment.

Additional support code and constants are also provided to assist in the assignment.

You need to implement a series of view classes and the controller class.

Regular Testing: It's crucial to test your GUI manually throughout the coding process.

Gradescope: Regularly upload your work to Gradescope to ensure it passes the tests. Even if your implementation appears correct visually, it might fail Gradescope tests if not implemented correctly.

Implementation Details: Adhere to the details from Section 6. Using your own structure might result in a grade of 0. Minor visual differences in your program might not cause test failures.

Uploading Responsibility: It's your duty to upload to Gradescope frequently to ensure your solution passes.

Assignment Structure: The document provides the necessary classes and methods for your assignment. You're encouraged to create helper methods to reduce code duplication and enhance readability.

Error Handling: Ensure regular gameplay doesn't crash or error your program. If a feature causes issues, comment it out before submission. Solutions that can't run might receive a grade of 0.

Imports: Do not add any imports. Doing so might lead to a deduction of up to 100% of your mark.

Provided Code: You can use any code provided by the teaching staff of this course for this semester. This includes support files, sample solutions, lecture, or tutorial code. Ensure the code is styled appropriately and is a correct approach to the problem.

Tkinter Documentation: For additional assistance with tkinter, refer to documentation on effbot2 and New Mexico Tech.

4. Provided Code:

a2 support.py: This is the support code file from assignment 2.

model.py: Contains modeling classes for the Sokoban game. It's an enhanced version of the solution to assignment 2, introducing features like coins and a basic shop for purchasing potions. It also offers methods for game resetting and more.

a3 support.py: Contains support code for the assignment. It provides:

Useful constants.

A function (get image) to create, resize, and optionally cache images.

AbstractGrid: An abstract view class derived from tk.Canvas that offers base functionality for various view classes.

maze files/: A folder with example maze files for testing.

images/: A folder with images for the assignment.

5. Recommended Approach:

GUI program development often requires working on multiple interacting classes simultaneously.

Instead of following the task sheet order, it's beneficial to focus on one feature at a time, testing it thoroughly before proceeding.

Each feature might need updates/extensions to the controller and possibly additions to one or more view classes.

Suggested order of features:

play game, main, and title: Set up the window, ensure it displays, and set its title.

Title banner: Display the title banner at the window's top.

FancyGameView: Implement basic tile display, entities display, player movement, and win/loss scenarios.

FancyStatsView: Implement basic (non-functional) display, followed by functional updates.

Shop: Implement basic display and handle item purchases.

6. Implementation:

Implement three view components: FancyGameView, FancyStatsView, and Shop.

Implement FancySokobanView class representing the overall view, and ExtraFancySokoban as the controller class.

The controller class should be developed in parallel with the view classes.

6. Implementation:

Components to Implement:

Three view components: FancyGameView, FancyStatsView, and Shop.

FancySokobanView class: Represents the overall view and manages the smaller components.

ExtraFancySokoban controller class: Instantiates the SokobanModel and FancySokobanView classes, handles events, and facilitates communication between model and view classes.

Approach: The section doesn't dictate a specific order for implementation. The controller class might need parallel development with the view classes. For a recommended order, refer to Section 5.

ChatGPT

Certainly! Here's a summarized version of the provided context:

Context Summary:

Assignment Overview:

Assignment 2: Implemented a text-based game of Fancy Sokoban using the Model-View-Controller (MVC) design pattern.

Assignment 3: Transition from a text-based interface to a graphical user interface (GUI) using tkinter. The game's appearance may differ based on the OS. User interactions shift from input calls to events like key-presses and mouse clicks. The solution should follow the Apple MVC design pattern. Modeling classes from Assignment 2 can be reused, and enhanced modeling classes are provided.

Guidelines:

Testing: Regular manual testing of the GUI is essential.

Gradescope: Upload work frequently to ensure it meets test criteria.

Implementation Details: Strictly follow details from Section 6. Deviations might result in a grade of 0.

Uploading: It's the student's responsibility to upload work to Gradescope.

Assignment Structure: The document outlines necessary classes and methods. Helper methods are encouraged.

Error Handling: Ensure the game doesn't crash. If a feature is problematic, comment it out before submission.

Imports: No additional imports allowed.

Provided Code: Use any code provided by the course's teaching staff. Ensure it's styled correctly.

Tkinter Documentation: Refer to effbot2 and New Mexico Tech for additional tkinter guidance.

Provided Code:

a2 support.py: Support code from assignment 2.

model.py: Enhanced modeling classes for the Sokoban game, introducing features like coins, a shop, game resetting, etc.

a3 support.py: Support code for assignment 3, including constants, image functions, and an abstract view class.

maze files/: Example maze files for testing.

images/: Images for the assignment.

Recommended Approach:

GUI development often involves working on multiple classes. Focus on one feature at a time and test thoroughly.

Suggested feature order: window setup, title banner, basic tile and entity display, player movement, win/loss scenarios, basic stats display, functional stats updates, shop display, and item purchases.

Implementation:

Components: Implement three view components (FancyGameView, FancyStatsView, and Shop), the overall view (FancySokobanView), and the controller class (ExtraFancySokoban).

Approach: The controller class should be developed alongside the view classes. For a recommended order, refer to Section 5.

I hope this summary provides a clear overview of the context. Please provide the support code or any additional information you'd like to share, and I'll be ready to assist further!

User

these are support code: a2\_support:Grid = list[list['Tile']]

Entities = dict[tuple[int, int], 'Entity']

Position = tuple[int, int]

# Tile constants

WALL = 'W'

FLOOR = ' '

GOAL = 'G'

FILLED\_GOAL = 'X'

# Entity constants

CRATE = 'C'

PLAYER = 'P'

STRENGTH\_POTION = 'S'

MOVE\_POTION = 'M'

FANCY\_POTION = 'F'

# Movement constants

UP = 'w'

DOWN = 's'

LEFT = 'a'

RIGHT = 'd'

DIRECTION\_DELTAS = {

UP: (-1, 0),

DOWN: (1, 0),

LEFT: (0, -1),

RIGHT: (0, 1),

}

def read\_file(maze\_file: str) -> tuple[list[list[str]], list[int, int]]:

""" A helper function to read maze files into a basic format.

Parameters:

maze\_file: The path to the maze file (e.g. 'maze\_files/maze1.txt')

Returns:

A tuple containing two items:

1) A simple representation of the maze

2) A list containing the starting values for the player's strength

and moves remaining respectively.

"""

with open(maze\_file, 'r') as file:

lines = file.readlines()

maze = [list(line.strip()) for line in lines[1:]]

player\_stats = [int(item) for item in lines[0].strip().split(' ')]

return maze, player\_stats

a3\_support:import tkinter as tk

from PIL import ImageTk, Image

from typing import Union

MAZE\_SIZE = 450

SHOP\_WIDTH = 200

BANNER\_HEIGHT = 75

STATS\_HEIGHT = 75

FONT = ('Arial', 16, 'bold')

TITLE\_FONT = ('Arial', 18, 'bold')

CRATE\_FONT = ('Arial', 20, 'bold')

def get\_image(

image\_name: str,

size: tuple[int, int],

cache: dict[str, ImageTk.PhotoImage] = None

) -> ImageTk.PhotoImage:

""" Returns the cached image for image\_id if one exists, otherwise creates a

new one, caches and returns it.

Parameters:

image\_name: The path to the image to load.

size: The size to resize the image to, as (width, height).

cache: The cache to use. If None, no caching is performed.

Returns:

The image for the given image\_name, resized appropriately.

"""

if cache is None or image\_name not in cache:

image = ImageTk.PhotoImage(image=Image.open(image\_name).resize(size))

if cache is not None:

cache[image\_name] = image

elif image\_name in cache:

return cache[image\_name]

return image

class AbstractGrid(tk.Canvas):

""" A type of tkinter Canvas that provides support for using the canvas as a

grid (i.e. a collection of rows and columns). """

def \_\_init\_\_(

self,

master: Union[tk.Tk, tk.Frame],

dimensions: tuple[int, int],

size: tuple[int, int],

\*\*kwargs

) -> None:

""" Constructor for AbstractGrid.

Parameters:

master: The master frame for this Canvas.

dimensions: (#rows, #columns)

size: (width in pixels, height in pixels)

"""

super().\_\_init\_\_(

master,

width=size[0] + 1,

height=size[1] + 1,

highlightthickness=0,

\*\*kwargs

)

self.\_size = size

self.set\_dimensions(dimensions)

def set\_dimensions(self, dimensions: tuple[int, int]) -> None:

""" Sets the dimensions of the grid.

Parameters:

dimensions: Dimensions of this grid as (#rows, #columns)

"""

self.\_dimensions = dimensions

def get\_cell\_size(self) -> tuple[int, int]:

""" Returns the size of the cells (width, height) in pixels. """

rows, cols = self.\_dimensions

width, height = self.\_size

return width // cols, height // rows

def pixel\_to\_cell(self, x: int, y: int) -> tuple[int, int]:

""" Converts a pixel position to a cell position.

Parameters:

x: The x pixel position.

y: The y pixel position.

Returns:

The (row, col) cell position.

"""

cell\_width, cell\_height = self.get\_cell\_size()

return y // cell\_height, x // cell\_width

def get\_bbox(self, position: tuple[int, int]) -> tuple[int, int, int, int]:

""" Returns the bounding box of the given (row, col) position.

Parameters:

position: The (row, col) cell position.

Returns:

Bounding box for this position as (x\_min, y\_min, x\_max, y\_max).

"""

row, col = position

cell\_width, cell\_height = self.get\_cell\_size()

x\_min, y\_min = col \* cell\_width, row \* cell\_height

x\_max, y\_max = x\_min + cell\_width, y\_min + cell\_height

return x\_min, y\_min, x\_max, y\_max

def get\_midpoint(self, position: tuple[int, int]) -> tuple[int, int]:

""" Gets the graphics coordinates for the center of the cell at the

given (row, col) position.

Parameters:

position: The (row, col) cell position.

Returns:

The x, y pixel position of the center of the cell.

"""

row, col = position

cell\_width, cell\_height = self.get\_cell\_size()

x\_pos = col \* cell\_width + cell\_width // 2

y\_pos = row \* cell\_height + cell\_height // 2

return x\_pos, y\_pos

def annotate\_position(

self,

position: tuple[int, int],

text: str,

font=None

) -> None:

""" Annotates the cell at the given (row, col) position with the

provided text.

Parameters:

position: The (row, col) cell position.

text: The text to draw.

"""

self.create\_text(self.get\_midpoint(position), text=text, font=font)

def clear(self):

""" Clears all child widgets off the canvas. """

self.delete("all") model:from a2\_support import \*

COIN = '$'

COIN\_AMOUNT = 5

class Tile:

""" Abstract class for a tile in the maze. """

TYPE = 'Abstract Tile'

BLOCKING = False

def is\_blocking(self) -> bool:

""" Returns True iff this tile is blocking. """

return self.BLOCKING

def get\_type(self) -> str:

""" Returns the type of this tile. """

return self.TYPE

def \_\_str\_\_(self) -> str:

return self.get\_type()

def \_\_repr\_\_(self) -> str:

return str(self)

class Floor(Tile):

""" A basic floor tile (non-blocking) in the maze. """

TYPE = FLOOR

class Wall(Tile):

""" A basic wall tile (blocking) in the maze. """

TYPE = WALL

BLOCKING = True

class Goal(Tile):

""" A goal tile onto which crates should be pushed in the maze. """

TYPE = GOAL

def \_\_init\_\_(self) -> None:

""" Constructor for Goal. Goal is initially unfilled. """

super().\_\_init\_\_()

self.\_is\_filled = False

def fill(self) -> None:

""" Fills this goal. """

self.\_is\_filled = True

def unfill(self) -> None:

""" Unfills this goal. """

self.\_is\_filled = False

def is\_filled(self) -> bool:

""" Returns True iff the goal is filled. """

return self.\_is\_filled

def \_\_str\_\_(self):

return FILLED\_GOAL if self.\_is\_filled else self.get\_type()

class Entity:

""" Abstract class for an entity in the maze. """

TYPE = 'Abstract Entity'

MOVABLE = False

def get\_type(self) -> str:

""" Returns the type of this entity. """

return self.TYPE

def is\_movable(self) -> bool:

""" Returns True iff this entity is movable. """

return self.MOVABLE

def \_\_str\_\_(self):

return self.get\_type()

def \_\_repr\_\_(self):

return str(self)

class Crate(Entity):

""" A crate entity in the maze. """

TYPE = CRATE

MOVABLE = True

def \_\_init\_\_(self, strength: int) -> None:

""" Constructor for Crate.

Parameters:

strength: The strength required to push this crate.

"""

super().\_\_init\_\_()

self.\_strength = strength

def get\_strength(self) -> int:

""" Returns the strength required to push this crate. """

return self.\_strength

def \_\_str\_\_(self):

return str(self.\_strength)

class Coin(Entity):

""" A coin entity in the maze, which can be collected by a player to

increase their money.

"""

TYPE = COIN

class Potion(Entity):

""" Abstract class for a potion entity in the maze. """

TYPE = 'Potion'

EFFECT = {}

def effect(self) -> dict[str, int]:

""" Returns the effect of this potion. Keys that may (or may not) exist

in this dictionary are 'strength' and 'moves'.

"""

return self.EFFECT

class StrengthPotion(Potion):

""" A potion that increases the strength of the player. """

TYPE = STRENGTH\_POTION

EFFECT = {'strength': 2}

class MovePotion(Potion):

""" A potion that increases the moves remaining for the player. """

TYPE = MOVE\_POTION

EFFECT = {'moves': 5}

class FancyPotion(Potion):

""" A potion that increases both the strength and moves remaining for the

player.

"""

TYPE = FANCY\_POTION

EFFECT = {'strength': 2, 'moves': 2}

class Player(Entity):

""" A player entity in the maze. """

TYPE = PLAYER

def \_\_init\_\_(self, start\_strength: int, moves\_remaining: int) -> None:

""" Constructor for Player.

Parameters:

start\_strength: The starting strength of the player.

moves\_remaining: The number of moves the player can make.

"""

super().\_\_init\_\_()

self.\_strength = start\_strength

self.\_moves\_remaining = moves\_remaining

self.\_money = 0

def get\_money(self) -> int:

""" Returns the amount of money the player has. """

return self.\_money

def add\_money(self, money: int) -> None:

""" Adds money to the player's total.

Parameters:

money: The amount of money to add. This may be positive or negative.

"""

self.\_money += money

def is\_movable(self) -> bool:

""" Returns True iff the player is movable. """

return self.\_moves\_remaining > 0

def get\_strength(self) -> int:

""" Returns the strength of the player. """

return self.\_strength

def add\_strength(self, strength: int) -> None:

""" Adds strength to the player.

Parameters:

strength: The amount of strength to add.

"""

self.\_strength += strength

def get\_moves\_remaining(self) -> int:

""" Returns the number of moves remaining for the player. """

return self.\_moves\_remaining

def add\_moves\_remaining(self, moves: int) -> None:

""" Adds moves to the player's moves remaining.

Parameters:

moves: The number of moves to add. This may be positive or negative.

"""

self.\_moves\_remaining += moves

def apply\_effect(self, potion\_effect: dict[str, int]) -> None:

""" Applies the effects described in potion\_effect to the player.

Parameters:

potion\_effect: The effect of the potion. Only the keys 'strength'

and 'moves' will be considered.

"""

self.add\_strength(potion\_effect.get('strength', 0))

self.add\_moves\_remaining(potion\_effect.get('moves', 0))

TILE\_IDS\_TO\_CLASS = {

FLOOR: Floor,

WALL: Wall,

GOAL: Goal,

FILLED\_GOAL: Goal,

}

ENTITY\_IDS\_TO\_CLASS = {

CRATE: Crate,

COIN: Coin,

PLAYER: Player,

STRENGTH\_POTION: StrengthPotion,

MOVE\_POTION: MovePotion,

FANCY\_POTION: FancyPotion,

}

def convert\_maze(raw\_maze: list[list[str]]) -> tuple[Grid, Entities, Position]:

""" Converts a raw maze into a proper maze, entities and player position.

Parameters:

raw\_maze: The raw maze from the file.

Returns:

A tuple containing three items:

1) The maze as a list of lists (rows) of tile objects.

2) A dictionary mapping (row, col) positions to the entities at

those positions on the maze. Positions only exist in this

dictionary if there is an entity at that position.

3) The player's starting position.

"""

proper\_maze = []

entities = {}

player\_position = None

for i, row in enumerate(raw\_maze):

new\_row = []

for j, tile\_type in enumerate(row):

tile = TILE\_IDS\_TO\_CLASS.get(tile\_type, Floor)()

if tile\_type == FILLED\_GOAL:

tile.fill()

new\_row.append(tile)

if not TILE\_IDS\_TO\_CLASS.get(tile\_type):

if tile\_type == PLAYER:

player\_position = (i, j)

else:

if tile\_type.isdigit():

tile\_type = int(tile\_type)

entity = Crate(tile\_type)

else:

entity = ENTITY\_IDS\_TO\_CLASS.get(tile\_type)()

entities[(i, j)] = entity

proper\_maze.append(new\_row)

return proper\_maze, entities, player\_position

class SokobanModel:

""" A model for a Sokoban game. """

ITEM\_COSTS = {

STRENGTH\_POTION: 5,

MOVE\_POTION: 5,

FANCY\_POTION: 10,

}

def \_\_init\_\_(self, maze\_file: str) -> None:

""" Constructor for SokobanModel.

Parameters:

maze\_file: The path to the maze file (e.g. 'maze\_files/maze1.txt')

"""

self.\_maze\_file = maze\_file

self.reset()

def reset(self) -> None:

""" Resets the model to its initial state. """

raw\_maze, player\_stats = read\_file(self.\_maze\_file)

self.\_maze, self.\_entities, self.\_player\_position = convert\_maze(

raw\_maze)

self.\_player = Player(\*player\_stats)

self.\_last\_state = {

'maze': [[item for item in row] for row in self.\_maze],

'entities': {key: value for key, value in self.\_entities.items()},

'player\_stats': player\_stats,

'player\_position': self.\_player\_position,

'last\_filled': None,

}

def get\_shop\_items(self) -> dict[str, int]:

""" Returns a dictionary mapping item names to their cost. """

return self.ITEM\_COSTS

def attempt\_purchase(self, item: str) -> bool:

""" Attempts to purchase the given item.

Parameters:

item: The id / type of the item to purchase.

"""

if self.\_player.get\_money() < self.ITEM\_COSTS.get(item):

return False

self.\_player.add\_money(-self.ITEM\_COSTS[item])

self.\_entities[self.\_player\_position] = ENTITY\_IDS\_TO\_CLASS[item]()

self.\_handle\_potion(self.\_player\_position)

return True

def get\_maze(self) -> Grid:

""" Returns the maze. """

return self.\_maze

def get\_dimensions(self) -> tuple[int, int]:

""" Returns the dimensions of the maze as (#rows, #columns). """

return len(self.\_maze), len(self.\_maze[0])

def get\_entities(self) -> Entities:

""" Returns a dictionary mapping (row, col) positions to the entities at

those positions on the maze. Positions only exist in this dictionary

if there is an entity at that position.

"""

return self.\_entities

def get\_player\_position(self) -> Position:

""" Returns the player's current position. """

return self.player\_position

def get\_player\_moves\_remaining(self) -> int:

""" Returns the number of moves remaining for the player. """

return self.\_player.get\_moves\_remaining()

def get\_player\_strength(self) -> int:

""" Returns the player's current strength. """

return self.\_player.get\_strength()

def get\_player\_money(self) -> int:

""" Returns the amount of money the player has. """

return self.\_player.get\_money()

def undo\_move(self) -> None:

""" Undoes the last valid move made by the player. """

self.\_maze = self.\_last\_state['maze']

self.\_entities = self.\_last\_state['entities']

self.\_player\_position = self.\_last\_state['player\_position']

self.\_player = Player(\*self.\_last\_state['player\_stats'])

if self.\_last\_state['last\_filled'] is not None:

row, col = self.\_last\_state['last\_filled']

self.\_get\_tile(row, col).unfill()

def attempt\_move(self, direction: str) -> bool:

""" Attempts to move the player in the given direction.

Parameters:

direction: The direction to move in. This should be one of the

constants UP, DOWN, LEFT or RIGHT, or 'u' for undo.

Returns:

True iff the move was successful.

"""

# Handle undo move

if direction == 'u':

self.undo\_move()

return True

# Make a copy of important information about this state to overwrite

# self.\_last\_state if the move is successful

last\_state = {

'maze': [[item for item in row] for row in self.\_maze],

'entities': {key: value for key, value in self.\_entities.items()},

'player\_stats': (self.\_player.get\_strength(),

self.\_player.get\_moves\_remaining()),

'player\_position': self.\_player\_position,

'last\_filled': None,

}

# Handle directional move

if not DIRECTION\_DELTAS.get(direction):

return False

new\_position = new\_row, new\_col = self.\_get\_new\_position(

self.\_player\_position, direction)

if not self.\_in\_bounds(new\_row, new\_col):

return False

if self.\_get\_tile(new\_row, new\_col).is\_blocking():

return False

# Handle case where there is a crate in the new position

entity\_present = self.\_entities.get(new\_position)

if entity\_present is not None:

if entity\_present.get\_type() == CRATE:

if not self.\_attempt\_push(new\_position, direction):

return False

elif entity\_present.get\_type() == COIN:

self.\_player.add\_money(COIN\_AMOUNT)

self.\_entities.pop(new\_position)

elif isinstance(entity\_present, Potion):

self.\_handle\_potion(new\_position)

self.\_player\_position = new\_position

self.\_player.add\_moves\_remaining(-1)

self.\_last\_state = last\_state

return True

def has\_won(self) -> bool:

""" Returns True iff the player has won the game. """

for row in self.\_maze:

for tile in row:

if tile.get\_type() == GOAL and not tile.is\_filled():

return False

return True

def \_get\_new\_position(self, position: Position, direction: str) -> Position:

""" Returns the new position for an entity if it were to move in the

given direction from the given position. This does not consider

whether the move is valid.

Parameters:

position: The current (row, column) position.

direction: The direction in which to move. This should be one of the

constants UP, DOWN, LEFT or RIGHT.

Returns:

The new (row, column) position for the entity.

"""

delta = DIRECTION\_DELTAS.get(direction)

return position[0] + delta[0], position[1] + delta[1]

def \_get\_tile(self, row: int, col: int) -> Tile:

""" Returns the tile at the given (row, col) position.

Parameters:

row: The row of the tile.

col: The column of the tile.

Returns:

The tile at the given position.

Preconditions:

The given position is in bounds for the maze.

"""

return self.\_maze[row][col]

def \_in\_bounds(self, row: int, col: int) -> bool:

""" Returns True iff the given (row, col) position is in bounds for the

maze.

Parameters:

row: The row of the position.

col: The column of the position.

Returns:

True iff the given position is in bounds for the maze.

"""

return 0 <= row < len(self.\_maze) and 0 <= col < len(self.\_maze[0])

def \_attempt\_push(self, position: Position, direction: str) -> bool:

""" Attempts to push a crate from the given position in the given

direction.

Parameters:

position: The current (row, col) position of the crate.

direction: The direction in which to push the crate. This should be

one of the constants UP, DOWN, LEFT or RIGHT.

Returns:

True iff the crate was successfully pushed.

"""

new\_row, new\_col = self.\_get\_new\_position(position, direction)

tile = self.\_get\_tile(new\_row, new\_col)

# If the new position is out of bounds, or contains a blocking tile or

# entity, return False

if not self.\_in\_bounds(new\_row, new\_col):

return False

if tile.is\_blocking():

return False

if (new\_row, new\_col) in self.\_entities:

return False

# If the player isn't strong enough for this crate, return False

crate\_strength = self.\_entities.get(position).get\_strength()

if crate\_strength > self.\_player.get\_strength():

return False

crate = self.\_entities.pop(position)

# If the crate would fill an unfilled goal, do so and don't add the

# crate back to the entities

if tile.get\_type() == GOAL and not tile.is\_filled():

tile.fill()

self.\_last\_state['last\_filled'] = (new\_row, new\_col)

return True

# Otherwise, add the crate back to the entities

self.\_entities[(new\_row, new\_col)] = crate

return True

def \_handle\_potion(self, position: tuple[int, int]) -> None:

""" Handles applying the effect of a potion at the given position to the

player.

Parameters:

position: The position of the potion.

"""

potion = self.\_entities.pop(position)

self.\_player.apply\_effect(potion.effect())

plz bear in mind the parameters and others, which we will use to answer the questions next. and next i will provide you the code i write for some questions, i hope you firstly revised my code, and then based on my code to do a lot of other questions.

ChatGPT

Understood! Please provide the code you've written for the questions, and I'll review it. After that, I'll assist you with the other questions based on the provided code and the support code you've shared.

User

these are questions based on the context i provided before:6.1 FancyGameView

FancyGameView should inherit from AbstractGrid (see a3 support.py). The FancyGameView is

a grid displaying the game map (e.g. all tiles and entities, including the player). An example of a

completed FancyGameView is shown in Figure 2. The methods you must implement in this class

are:

• init (self, master: tk.Frame | tk.Tk, dimensions: tuple[int, int], size:

tuple[int, int], \*\*kwargs) -> None: Sets up the FancyGameView to be an AbstractGrid

with the appropriate dimensions and size, and creates an instance attribute of an empty

dictionary to be used as an image cache.

4

• display(self, maze: Grid, entities: Entities, player position: Position ):

Clears the game view, then creates (on the FancyGameView instance itself) the images for

the tiles and entities. If an entity is at a specific location, you may assume there is a FLOOR

tile undeneath. If an entity is at a position, the tile image should be rendered beneath the

entity image. You must use the get image function from a3 support.py to create

your images.

6.2 FancyStatsView

FancyStatsView should inherit from AbstractGrid (see a3 support.py). It is a grid with 3

rows and 3 columns. The top row displays the text ‘Player Stats’ in a bold font in the second

column. The second row displays titles for the stats, and the third row displays the values for

those stats. The FancyStatsView should span the entire width of the game and shop combined.

An example of a completedFancyStatsView in the game is shown in Figure 3. The methods you

must implement in this class are:

• init (self, master: tk.Tk | tk.Frame) -> None: Sets up this FancyStatsView to

be an AbstractGrid with the appropriate number of rows and columns, and the appropriate

width and height (see a3 support.py).

• draw stats(self, moves remaining: int, strength: int, money: int) -> None:

Clears the FancyStatsView and redraws it to display the provided moves remaining, strength,

and money. E.g. in Figure 3, this method was called with moves remaining = 10, strength

= 4, and money = 7.Figure 3: FancyStatsView after redrawing with moves remaining set to 10, strength set to 4, and

money set to 7.

6.3 Shop

Shop should inherit from tk.Frame. The Shop is a frame displaying relevant information and

buttons for all the buyable items in the game (see the get shop items method in SokobanModel).

The Shop should contain a title at the top and a frame for each buyable item (each potion). Each

item’s frame should contain the following widgets, packed left to right:

• A label containing the name of the item and the cost to buy that item.

• A button for buying the item at the listed price. The callback for these buttons must be

created in the controller (see ExtraFancySokoban) and passed to the Shop when calling

create buyable item (see below).

See Figure 4 for an example of the shop interface.

The methods that you must implement in this class are:

• init (self, master: tk.Frame) -> None: Sets up the shop to act like a tk.Frame

and to have a title label at the top in bold font. Note that you are not required to create

the item frames and internal widgets here.

• create buyable item( self, item: str, amount: int, callback: Callable[[], None]

) -> None: Create a new item in this shop. That is, this method creates a new frame within

the shop frame and then creates a label and button within that child frame. The button

should be bound to the provided callback.

Note: Handling callbacks is an advanced task. These callbacks will be created within the controller

class, as this is the only place where you have access to the required modelling information. Start

this task by trying to render display correctly, without the callbacks. Then integrate these views

into the game before working on the callbacks.

6.4 FancySokobanView

The FancySokobanView class provides a wrapper around the smaller GUI components you have

just built, and provides methods through which the controller can update these components.

The methods that you must implement in this class are:

• init (self, master: tk.Tk, dimensions: tuple[int, int], size: tuple[int,

int]) -> None: Sets up a new FancySokobanView instance. This includes creating the title

banner, setting the title on the window, and instantiating and packing the three widgets

described earlier in this task sheet.

• display game( self, maze: Grid, entities: Entities, player position: Position

) -> None: Clears and redraws the game view.

• display stats(self, moves: int, strength: int, money: int) -> None: Clears

and redraws the stats view.

create shop items( self, shop items: dict[str, int], button callback:

Callable[[str], None] | None = None ) -> None: Creates all the buyable items in the

shop. shop items maps item id’s (result of calling get type on the item entity) to price.

For each of these items, the callback given to create buyable item in Shop should be a

function which requires no positional arguments and calls button callback with the item

id as an argument. Note: if you create your callback within a loop using a lambda function,

you may need to include a keyword argument with a default value of the specific item’s id

in order to prevent Python from using the last item for all buttons.

6.5 ExtraFancySokoban

ExtraFancySokoban is the controller class for the overall game. The controller is responsible for

creating and maintaining instances of the model and view classes, event handling, and facilitating

communication between the model and view classes. Figure 1 provides an example of how the

ExtraFancySokoban game should look. Certain events should cause behaviour as per Assignment

2 (note that this includes the ability to undo a move). You should not reimplement this behaviour,

but rather use attempt move method for the model. The methods that you must implement in

this class are:

• init (self, root: tk.Tk, maze file: str) -> None: Sets up the ExtraFancySokoban

instance. This includes creating instances of SokobanModel and SokobanView, creating the

shop items, binding keypress events to the relevant handler, and then redrawing the display

to show the initial game state. When creating the shop items, you will need to create a

function to pass to the create shop items method. This method should:

1. Take an item id as a parameter

2. Tells the model to attempt to buy that item

3. Tells the entire view to redraw• redraw(self) -> None: Redraws the game view and stats view based on the current model

state.

• handle keypress(self, event: tk.Event) -> None: An event handler to be called when

a keypress event occurs. Should tell the model to attempt the move as per the key pressed,

and then redraw the view. If the game has been won or lost after the move, this method

should cause a messagebox to display informing the user of the outcome and asking if they

would like to play again (see Fig. 5). If the user selects yes, the game should be reset

(i.e. reset the model and then redraw the view). If the user selects no, the program should

terminate gracefully. this is my code:import tkinter as tk

from tkinter import messagebox, filedialog

from typing import Callable, Union

from model import SokobanModel, Tile, Entity

from a2\_support import \*

from a3\_support import \*

# Write your classes and functions here

class FancyGameView(AbstractGrid):

def \_\_init\_\_(self, master: tk.Frame | tk.Tk,\

dimensions: tuple [int,int], size:tuple[int,int],\

\*\*kwargs) -> None:

"""Sets up the FancyGameView to be an AbstractGrid with the appropriate dimensions and size,

and creates an instance attribute of an empty dictionary to be used as an image cache."""

super().\_\_init\_\_(master,dimensions,cell\_size=size,\*\*kwargs)

self.\_image\_cache = {}

def display(self,maze:Grid,entities:Entities,player\_position:Position):

"""Clears the game view, then creates (on the FancyGameView instance itself) the images for

the tiles and entities."""

self.delete(tk.ALL) # Clear the game view

# Loop through the maze and display tiles and entities

for row in range(len(maze)): # Loop through the rows

for col in range(len(maze[row])):

tile = maze[row][col] #the location of the tile

image = get\_image(tile,self.get\_cell\_size(),self.\_image\_cache) # Get the image for the tile

self.create\_image(self.get\_midpoint(row,col),image=image) # Create the image on the canvas

"display entities"

"if an entity is at a specific location, assume there's a Floor tile underneath it"

for position, entity in entities.items():

# # If an entity is at a specific location, assume there's a FLOOR tile underneath

image = get\_image(entity,self.get\_cell\_size(),self.\_image\_cache)

self.create\_image(self.get\_midpoint(position),image=image)

#Render the entity image on top of the title image

entity\_image = get\_image(entity,self.get\_cell\_size(),self.\_image\_cache)

self.create\_image(self.get\_midpoint([position]),image=entity\_image)

class FancyStatsView(AbstractGrid):

def \_\_init\_\_(self, master: tk.Tk | tk.Frame) -> None:

# # Set up the grid with 3 rows and 3 columns

super().\_\_init\_\_(master, dimensions=(3, 3), size=(MAZE\_SIZE+ SHOP\_WIDTH,STATS\_HEIGHT))

#display the title in th top row and second column

self.create\_text(self.get\_midpoint(0,1),text="Plazyer Stats",font=TITLE\_FONT,anchor=tk.CENTER)

"display the second line of stats for initial view"

self.create\_text(self.get\_midpoint(1,0),text="Moves remaining: ",font=FONT)

self.create\_text(self.get\_midpoint(1,1),text="Strength: ",font=FONT)

self.create\_text(self.get\_midpoint(1,2),text="Money: ",font=FONT)

def draw\_stats(self, move\_remaining: int, strength: int, money: int) -> None:

self.clear() #clear out the previous stats

"Redraw the titles and 'static' stats"

self.create\_text(self.get\_midpoint(0,1),text="Player Stats",font=TITLE\_FONT,anchor=tk.CENTER)

self.create\_text(self.get\_midpoint(1,0),text="Moves remaining: ",font=FONT)

self.create\_text(self.get\_midpoint(1,1),text="Strength: ",font=FONT)

self.create\_text(self.get\_midpoint(1,2),text="Money: ",font=FONT)

"display the values in third line of stats"

self.create\_text(self.get\_midpoint(2,0),text=str(move\_remaining),font=FONT)

self.create\_text(self.get\_midpoint(2,1),text=str(strength),font=FONT)

self.create\_text(self.get\_midpoint(2,2),text=str(money),font=FONT)

class Shop(tk.Frame):

def \_\_init\_\_(self, master:tk.Frame)->None:

super().\_\_init\_\_(master)

"initial the title of the shop"

self.title\_label =tk.Label(self,text="Shop",font=TITLE\_FONT)

self.title\_label.pack(pady=10) #add some padding around the title

def create\_buyable\_item(self, item: str, amount: int, callback: Callable[[],None])-> None:

"create a new frame for the item"

item\_frame = tk.Frame(self)

item\_frame.pack(pady=5)

"create a label for the item"

item\_label = tk.Label(item\_frame,text=f"{item} (${amount})")

item\_label.pack(side=tk.LEFT)

"create a button to buy the item"

buy\_button = tk.Button(item\_frame,text="Buy",command=callback)

buy\_button.pack(side=tk.LEFT)

class FancySokobanView(tk.Frame):

def \_\_init\_\_(self,master:tk.Tk, dimensions: tuple[int,int], size: tuple[int,int])->None:

super().\_\_init\_\_(master)

master.title("Fancy Sokoban") #set the title of the window

"create the banner in master"

self.banner = tk.Label(self, text = "Fancy Sokoban", font = TITLE\_FONT)

self.banner.pack()

"instantiating and packing the three widgets"

self.game\_view = FancyGameView(self,dimensions,size)

self.game\_view.pack()

self.stats\_view = FancyStatsView(self)

self.stats\_view.pack()

self.shop = Shop(self)

self.shop.pack()

def display\_game(self, maze: Grid, entities: Entities, player\_position: Position) -> None:

"""

Redraws the game view with the given maze, entities, and player position.

Args:

maze (Grid): The maze to display.

entities (Entities): The entities to display.

player\_position (Position): The position of the player in the maze.

"""

self.game\_view.clear()

self.game\_view.display(maze, entities, player\_position)

def display\_stats(self, moves\_remaining: int, strength: int, money: int)->None:

self.stats\_view.clear()

self.stats\_view.draw\_stats(moves\_remaining,strength,money)

" connect the buy\_item\_callback to the shop"

def create\_shop\_items( self, shop\_items: dict[str,int],button\_callback:Callable[[str], None]| None = None)->None:

"create a buyable item in the shop"

for item\_id, price in shop\_items.items():

# Use a closure to capture the current item\_id

def make\_callback(item\_id):

return lambda: button\_callback(item\_id)

callback = make\_callback(item\_id)

self.shop.create\_buyable\_item(item\_id, price, callback)