June 13: Object Oriented Design

Object Oriented Design (OOD)

- in the previous class we focused on

- OOP syntax
- programming style to write object oriented code

- today we will focus on how to design OO code

- analyze problem domain into a set of objects, classify obj. typ
- write a class to represent each object
- instantiate classes to create individual objects
- write the ‘main’ program, specifying communication/interaction between objects

OOD: Analysis

- critical step: divide problem domain into right set of objects

- in playing cards, this was fairly obvious, and there was little interaction/communication between objects

- typically this is not the case

- in non-trivial problems equally plausible decomposition of problem into a different set of objects is possible!

- easiest case: problem domain consists of physical objects (packages, boxes, trucks, warehouses)

- hardest cases:

- no physical objects, or objects do not leave physical traces

- too many physical objects (climate modelling & molecules), requiring some degree of **abstraction**

OOD: Classes

- once types of objects identified, we can start writing classes

- we need to identify objects' <u>attributes:</u>

- What **qualities** does this object have? weight, color, size, owner, position

- What **information** can it **store**?

- we need to identify objects' <u>methods</u>:

- What can it **do**? what kind of requests can it respond to? Can it move, cancel, record, trigger?

- What questions can it **answer** about itself? size, position, colour?

- attributes correspond to adjectives & nouns

- methods correspond to verbs

OOD: Instantiation & main program

- instantiation

- creating objects the program requires, consisting of calls to the object constructors

```
d = Deck()
 ...
roxx = Hand()
chris = Hand()
```

- writing the main program

- the last step to complete the program

- specifies the pattern of interaction between objects in the system

Extended Example: Hunt the Wumpus!

- in playing card example,

- objects were fairly obvious
- not much communication/interaction between objects
- this is not typical!

- in this extended example we will consider

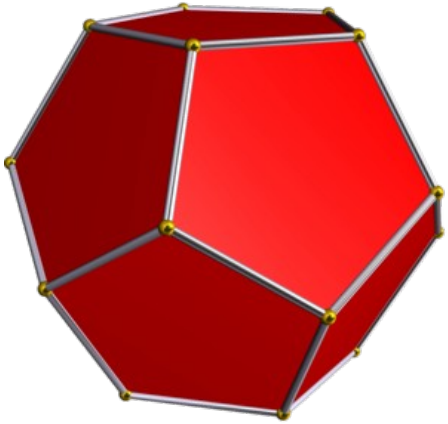
- problem with more objects
- less obvious analysis than playing card
- with considerably more interaction/communication b/w objcts

- our task is the implement the game:
Hunt the Wumpus!

About “Hunt the Wumpus”

- a text based survival horror type adventure game
- developed in 1973 by Gregory Yob
- player moves through a series of connected caves (arranged in dodecahadron)
- and hunts a monster named Wumpus
- while avoiding fatal bottomless pits & super bats
- player fires a ‘crooked arrow’ to kill Wumpus
- originally written in BASIC

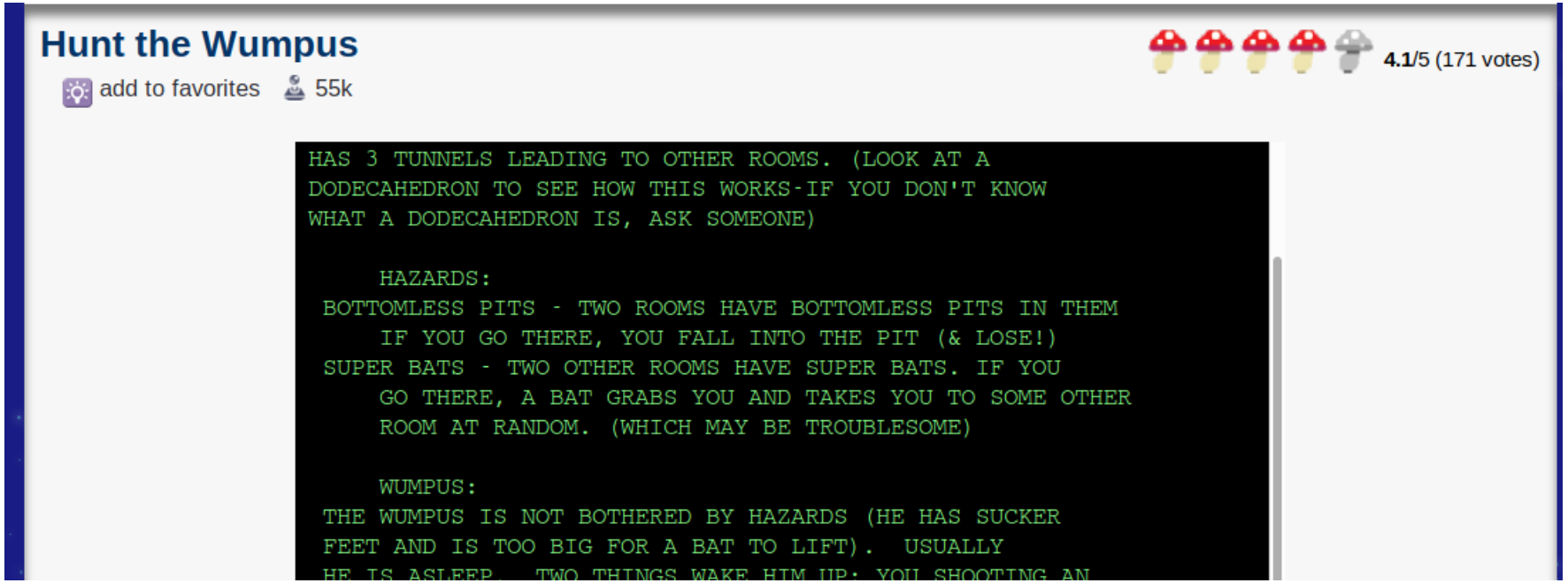
```
You are in room 3.
Tunnels lead to 2, 4, 12.
Shoot or Move (S-M)? M
Where to? 12

You are in room 12.
   I smell a Wumpus.
Tunnels lead to 3, 11, 13.
Shoot or Move (S-M)? S
No. of Rooms (1-5)? 1
Room # 13
AHA! You got the wumpus!
HEE HEE HEE - The Wumpus'll get you next time!!
Same setup (Y-N)? Y

You are in room 2.
   I feel a draft.
Tunnels lead to 1, 3, 10.
Shoot or Move (S-M)? M
Where to? 3

You are in room 3.
Tunnels lead to 2, 4, 12.
Shoot or Move (S-M)? M
```



Hunt the Wumpus [Demo]

Check out:

https://jayisgames.com/games/hunt-the-wumpus/



OO Analysis

- we have a pretty good informal specification of the problem:

- description of the game and
- its rules are displayed on start

- remembering that objects correspond to nouns,

- read over description and highlight objects

OO Analysis

```
Welcome to "Hunt the Wumpus"

The  wumpus  lives in a  cave  of 20  rooms . Each room has 3  tunnels  to
other rooms. (Look at a dodecahedron to see how this works. If you
don't know what a dodecahedron is, ask someone.)

Hazards:
 Bottomless  pits  - Two rooms have bottomless pits in them. If  you  go
 there, you fall into the pit (& lose)!
 Super  bats  - Two other rooms have super bats. If you go there, a
 bat grabs you and takes you to some other room at random (which
 may be troublesome).

Wumpus:
 The wumpus is not bothered by hazards. (He has sucker feet and is
 Too big for a bat to lift.) Usually he is asleep. Two things
 wake him up: Your shooting an  arrow , or your entering his room.
 If the wumpus wakes, he moves (p=.75) one room or stays still
 (p=.25). After that, if he is where you are, he eats you up and
 you lose!
```

OO Analysis

```
You:
 Each turn you may move or shoot a crooked arrow.
 Moving: You can move one room (Through one tunnel).
 Arrows: You have 5 arrows. You lose when you run out. Each
 arrow can go from 1 to 3 rooms. You aim by telling the
 computer the rooms to which you want the arrow to go. If the
 arrow can't go that way (if no tunnel) it moves at random to
 the next room.
 If the arrow hits the wumpus, you win.
 If the arrow hits you, you lose.

Warnings:
 When you are one room away from a wumpus or hazard, the computer
 says:
 Wumpus: "I smell a wumpus!"
 Bat : "Bats nearby!"
 Pit : "I feel a draft!"

You are in room 12.
There are tunnels to rooms 11, 13, and 19.
I feel a draft!
Shoot, move, or quit (s/m/q)? m
To which room (11, 13, or 19)? 13
AEEEIIIIII! You fell into a pit!
```

OO Analysis

```
Welcome to "Hunt the Wumpus"

The wumpus lives in a cave of 20 rooms. Each room has 3 tunnels to
other rooms. (Look at a dodecahedron to see how this works. If you
don't know what a dodecahedron is, ask someone.)

Hazards:
 Bottomless pits - Two rooms have bottomless pits in them. If you go
 there, you fall into the pit (& lose)!
 Super bats - Two other rooms have super bats. If you go there, a
 bat grabs you and takes you to some other room at random (which
 may be troublesome).

Wumpus:
 The wumpus is not bothered by hazards. (He has sucker feet and is
 Too big for a bat to lift.) Usually he is asleep. Two things
 wake him up: Your shooting an arrow, or your entering his room.
 If the wumpus wakes, he moves (p=.75) one room or stays still
```

OO Analysis

The types of objects we have then are:

- a wumpus;
- a cave system of
- rooms connected by
- tunnels;
- pits;
- you, i.e. the player,
- bats; and
- arrows.

From the rules, we can tell how objects will behave:

- Wumpus mostly sleeps, wakes when disturbed, and reacts in a random way
- Bats grab players and fly them somewhere else
- You, the player, move from room to room, and shoot arrows etc.

Object attributes and methods

Keeping the objects we've identified in mind we can re-read the game description looking for verbs and adjectives to reveal the objects' attributes and methods.

Player
- Attributes: At any instant the player has a **location** in the cave system and a **number of arrows** left.
- Methods: The player is able to **move** from room to room and to **shoot** arrows.

Bat
- Attributes: Like the player(s) each bat has a **location** in the cave system.
- Methods: A Bat's only action is to **snatch** the player and carry him or her away.

Pit
- Attributes: Like Players and Bats Pits have a **location**.
- Methods: Pits can **swallow** players who stumble into them.

Wumpus
- Attributes: **Location**
- Methods: All the wumpus does is **wake up** and either run off or eat a player.

Arrows
- Attributes: A **path** they will try and follow.
- Methods: An Arrow **flies** from room to room along the specified path.

Cave system objects

The cave system is the backbone of this system of objects:
- all the simple objects are situated in and/or move through it.

It is the most complex of the objects because it is a compound object:
- the cave system consists of multiple rooms connected by multiple tunnels.
- it will have to be a data container type of some sort.

Attributes: 20 **caves**; the **pattern of connections** between caves.

Methods: Should be able to tell us **where you can go from here** (i.e. what rooms are connected to this one); if a room **has a bat**; if a room **has a pit**; if the **player is in a room**.

The cave system contains 20 Room objects which as objects have their own attributes and methods:

Attributes: 3 **tunnels**, possibly **a bat**, possibly **a pit**, possibly **a wumpus**.

Methods: Should be able to tell us if it **has a bat**, **has a pit**, or **has a wumpus**.

Main routine Pseudocode

Let's use these class, attribute and method names we've identified so far and try to write the main routine of our program.

Doing so should reveal any missing objects, or missing attributes or methods of objects.

First, we might want to write pseudocode as plainly as we can in English.

```
While the game isn't over
    Display the game state
    Display the menu of possible actions
    Get the user's choice of action
    If the action is move
        Display the choices (connected rooms)
        Get the user's choice
        Move the player
        If the room has a bat
            Have the bat snatch the player
        Otherwise if the room has a pit
            Have the pit swallow the player
        Otherwise if the room has a Wumpus
            Wake the Wumpus up
    Otherwise if the action is shoot
        Get the path for the arrow to follow from the user
        Tell the arrow to fly that path
    Otherwise if the action is quit
        Quit the game
```

Main routine Pseudocode

```
 1 while not game_over:
 2     print cave_system
 3     print player.location
 4     action = input('Shoot, move or quit (s/m/q)? ')
 5     if action == 'm':
 6         print 'Choose from: ', cave_system.rooms[player.location].tunnels
 7         room_choice = int(input('Your choice? '))
 8         player.move(room_choice)
 9         new_room = cave_system.rooms[player.location] # Note simplifying alias.
10          if new_room.has_bat():
11              new_room.bat.snatch()
12          elif new_room.has_pit():
13              new_room.pit.swallow()
14              game_over = True
15          elif new_room.has_wumpus():
16              game_over = room.wumpus.wake_up()
17     elif action == 's':
18         player.shoot()
19     elif action == 'q':
20         game_over = True
```

class Cave_System

- In writing classes, one usually goes from simplest to most complicated
- in this case we will start the Cave_System, the most complicated one, as all other objects refer to it
- Remember our initial specs for this class:
  - **Attributes:** 20 caves; the pattern of connections between caves.
- Looking at our pseudocode, it seems we picked a good “container” for this info:

```
6     print('Choose from: ', cave_system.rooms[player.location].tunnels )
...
9     new_room = cave_system.rooms[player.location] # Note simplifying alias.
```

- This shows: , **cave_system** has **rooms** attribute, which is a list
- **rooms** is a list of **Room** objects

How can represent pattern of connections between Rooms?

- One simple way is for each **Room** to know to what other **Room**s it is connected
- it is reasonable for a **Room** to know where it is connected to
- That is in fact what **tunnels** attribute yields (possibly a list of rooms)

class Cave_System

- what about initial specs for methods:

Methods: Should be able to tell us where you can go from here/what rooms are connected to this one; if a room has a bat; if a room has a pit; if the player is in a room.

- if we look at our pseudocode, we see that these questions are asked of Room objects not cave_system

- these methods belong to Rooms

- So, does cave_system has any method?

- Looking at line 2, we realize a __str__ method is needed!

```
10          if new_room.has_bat():
                ...
12          elif new_room.has_pit():
                ...
15          elif new_room.has_wumpus():
                ...
```

```
2     print cave_system
```

class cave_system

In summary, cave_system has only two methods: **__init__** and **__str__**

```
20 class Cave_System:
21
22     def __init__(self, tunnels, system_map):
23         self.rooms = []
24         for i in range(len(tunnels)):
25             self.rooms.append( Room(i, tunnels[i]) )
26
27         self.map = system_map
28
29     def __str__(self):
30         return self.map
31
```

__str__ **has** to return a string, so we define a system map using chars

To the left we see a squashed dodecahedron

```
155 TUNNELS = [[1,7,19], [0,2,14], [1,3,6], [2,4,13], [3,5,11],
156            [4,6,9], [2,5,7], [0,6,8], [7,9,18], [5,8,10],
157            [9,11,17], [4,10,12], [11,13,16], [3,12,14], [1,13,15],
158            [14,16,19], [12,15,17], [10,16,18], [8,17,19], [0,15,18]
159           ]
160
161 MAP ='''
162      19 - - - - - - - - - - - - - - - - - -  15
163     /   \                                  /  \
164    /     \      - - - - 1 - - - -         /    \
165   /       \   /         |         \     /      \
166  /         \ /          |          \   /        \
167 |           0           2           14          |
168 |          /           / \          \           |
169 |         /           /   \          \          |
170 |        /           /     \          \         |
171 |       7 - - - 6              3 - - - 13       |
172 |        \         \          /         /       |
173 |         \         \        /         /        |
174 18 - - - - 8         5 - - - 4         12 - - - - 16
175  \          \       /          \       /         /
176   \          \     /            \     /         /
177    \          9 - - - 10 - - -  11             /
178     \                 |                       /
179      \                |                      /
180       \               |                     /
181         - - - - - -  17 - - - - - - - -
182 '''
```

class Room

- Rooms have the attributes and methods we identified in our first pass **plus** the methods we have shifted from Cave_Systems to Rooms

- Attributes: 3 **tunnels**, possibly a **bat**, possibly **a pit**, possibly **a wumpus**.
- Methods: Should be able to tell us if it **has a bat**, **has a pit**, or **has a wumpus**,

PLUS

- "should be able to tell us **where you can go from here**/what rooms are connected to this one; if a room **has a bat**; if a room **has a pit**; if the **player is in a room**.

- As a result, we got rid of duplicate methods **has_pit**, **has_wumpus**, etc

any other duplicate stuff?
“player is in room?” is duplicate, as can be seen in pseudocode where we do: **player.location**, i.e, we ask the player not the room if s/he is in it.

So remove this duplicated method from **Room** class

class Room

Notes:

- We will identify rooms by their number as shown in MAP previously.
- connections here is a list of rooms connected by tunnels to this one, e.g. [1, 7, 4] meaning this room is connected to room number 1, room number 7, and room number 4.
- By default a room does not have a bat, pit or wumpus (b=None, p=None, w=None). These will be sprinkled around the cave system later.
- The has_... methods that answer questions look to see if the relevant attribute is defined and if it is returns True, otherwise False.

```
32 class Room:
33     '''e.g. r = Room(0,[1,7,4])'''
34     def __init__(self, room_number, connections, \
35                  b = None, p = None, w = None):
36         self.number = room_number
37         self.tunnels = connections
38         self.bat = b
39         self.pit = p
40         self.wumpus = w
41 
42     def has_bat(self):
43         if self.bat:
44             return True
45         else:
46             return False
47 
48     def has_pit(self):
49         if self.pit:
50             return True
51         else:
52             return False
53 
54     def has_wumpus(self):
55         if self.wumpus:
56             return True
57         else:
58             return False
```

class Pit

- recall that **Pit** has one attribute: **location** and one method: **swallow**

```
78 class Pit:
79     def __init__(self, room_number):
80         self.location = room_number
81 
82     def swallow(self):
83         # What a pit does when it swallows a player.
84         print('AIEEEE! You\'re falling to your doom down the pit!')
```

In a modern graphic game a Pit's swallow method would trigger some fabulous animation of the player tumbling downward, but since this is text-based we have to settle for saying what is happening.

class Bat

- Like **Pit**s, **Bat**s have one attribute, **location**, and one method called **snatch**.
- Snatching is a little more involved than swallowing.
- The bat has to move the player to a new randomly chosen room and
- then flap off to another randomly chosen room.

```
29 #with bug
30 class Bat:
31     def __init__(self, roomnumber):
32         self.location = roomnumber
33 
34     def snatch(self, player):
35         print()
36         print('Aaah! A giant bat has snatched you!')
37         print()
38         self.location = random.randint(0, len(cave_system.rooms) - 1) # BUG!
39         player.location = random.randint(0, len(cave_system.rooms) - 1)
40         print("I am ", self, "in room number", self.location)
41 
```

Above, can you determine what the bug is ?

class Wumpus

Like **Bat**s, **Wumpus** has a **location** attribute and a single method called **wake_up**

```
86 class Wumpus:
87     def __init__(self, roomnumber):
88         self.location = roomnumber
89 
90     def wake_up(self):
91         ''' Wake Wumpus up and see what happens.
92         returns True if the Wumpus eats the player,
93         and False otherwise.'''
94         # 1 in 4 chance Wumpus eats the player,
95         if random.randint(1,4) == 4:
96             print('Uh oh the Wumpus got you!')
97             return True
98         # and 3 in 4 chance it runs to an adjacent room.
99         else:
100            print('Wheww! The Wumpus woke up but he ran away.')
101            self.location = random.choice(cave_system.rooms[self.location].tunnels) #
102            return False
```

class Player

- Like the other objects in the cave system **Player**s have **location**s.
- They also have two methods: they can **move** and they can **shoot** arrows.

```
103
104 class Player:
105     def __init__(self, roomnumber=0):
106         self.location = roomnumber
107
108     def move(self, new_location):
109         if new_location in cave_system.rooms[self.location].tunnels:
110             self.location = new_location
111         else:
112             print("You can't walk through walls in this game!")
113
114     def shoot(self):
115         # You will be filling this method in!
116         print("Shooting ...")
117
```

Note that the move method receives an argument saying what room to move to, but it doesn't blindly trust the argument to be valid and instead checks that the destination room is linked to the current room.

You will be filling in the **shoot** method!

Putting the pieces together

Spend some time on analyzing **Wumpus_4.py**

This is the longest program in the course so take some time to read it over. You'll notice that there is:

- some new code (whose details to be explained shortly),
- some bugs (that you'll be correcting as part of the assignment), and
- a few pieces that are missing (that you'll be adding as part of the assignment).

New code: Initializing the bats

- see wumpus_4.py

- Notes:

1. number of bats is not ‘hardcoded’:

instead we defined a global constant/parameter

NUMBATS

The jargon: the number of bats has been parameterized!

2. note random.choice: it choses an item at random from a list

```
room = random.choice(cave_system.rooms)
#vs
room = cave_system.rooms[random.randint(0, len(cave_system.rooms) - 1)]
```

New code (pointing to a missing class): shoot and Arrows

New code: show_cheats()

see wumpus_4.py

New code: Display status and warnings

-see wumpus_4.py

Bug: Fixing snatch

- Assume **Bat** is in Room 12, and goes to Room 8 after a snatch
- Then we do:
  self.location = random.randint(0, len(cave_system.rooms) – 1)

- Although **Bat**'s location is updated
  - Room 12 doesn't know that it doesn't have a **Bat** anymore
  - Room 8 doesn't know that it has a **Bat**

- so, in addition to **Bat**'s location to Room 8, we need to do:
  - set Room 12's bat attribute to None
  - set Room 8's bat attribute to reference this **Bat** object

<u>Another small Bug</u>

When the bat's new room is chosen

- it shouldn't be a room with a Bat already in it,
- nor should it be the room it has just dropped the player in (which will happen approximately 1 in 20 times since we are choosing randomly).
- Fixing this secondary bug will be part of your assignment.

Two more (small) problems

Bug: **Wumpus** movement

The wumpus position is not updated properly after moving for the same reason that **Bat**s' movements were not properly updating. Fixing this will be part of your assignment.

Missing Piece: Initialize Pits

You have probably noticed that although the **Bat**s and **Wumpus** are initialized, the Pits are not. Adding this is another part of your assignment.