The Model-View-Controller Paradigm

Welcome back to CS 2100!

Prof. Rasika Bhalerao

Separating concerns between the Model, View, and Controller

Some programs are large and complicated. We like to organize our code by separating it into "categories."

The Model

- Classes related to functionality
- Not care about what functionality is needed when
- Not depend on the controller calling its methods in a certain order
- Not care when / how results are shown to user

The View

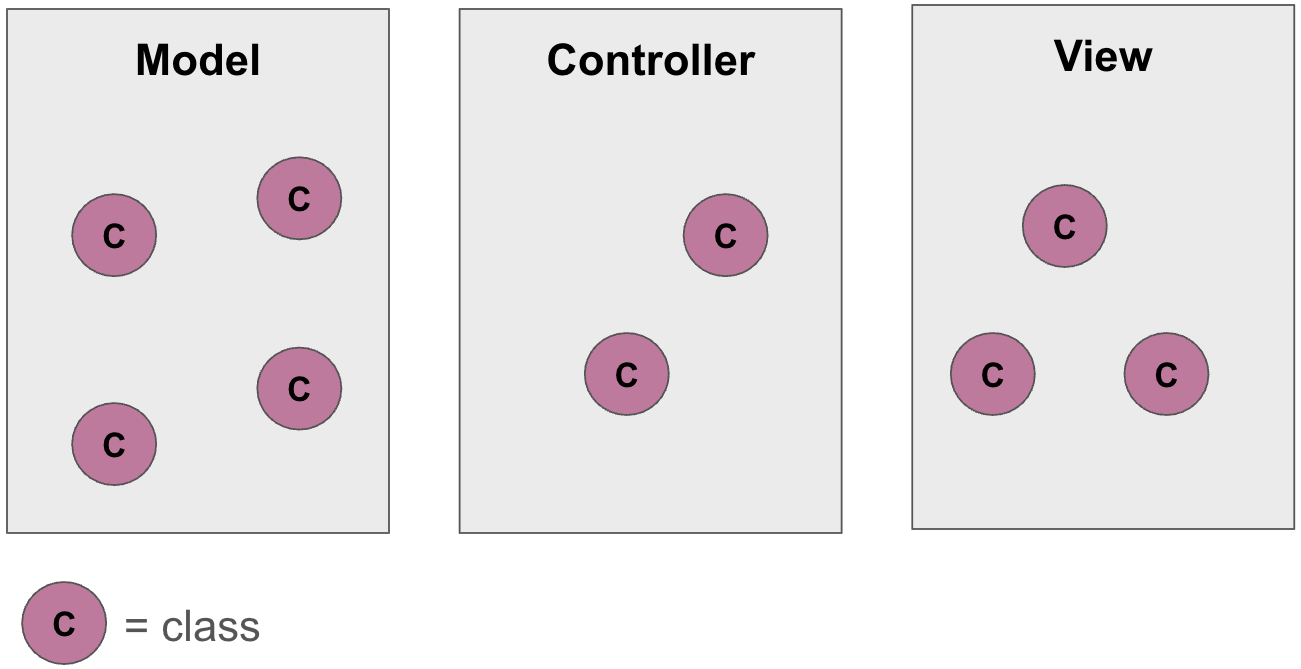
- Displays results to user
- Not care how results were calculated
- Not directly respond to user actions

The Controller

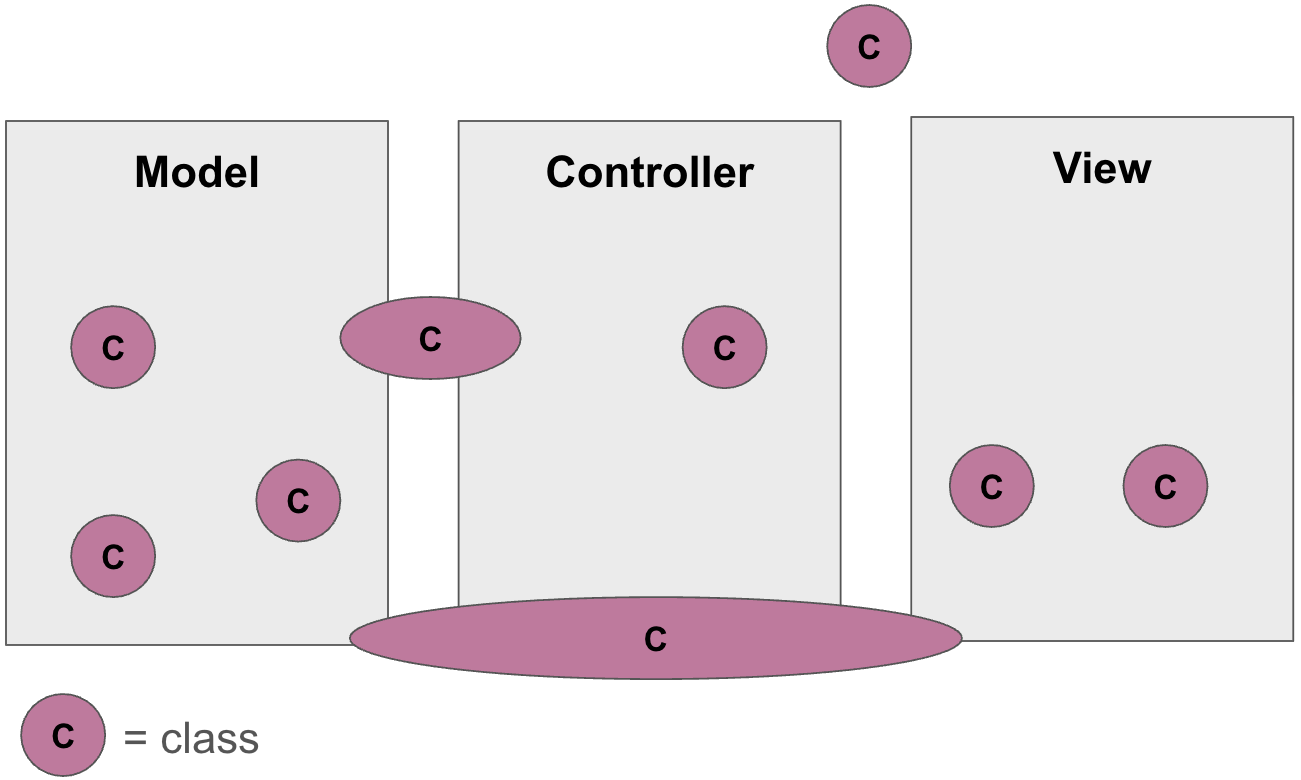
- Takes user inputs
- Tells the model what to do, tells view what info to display
- Not care how the model implemented the functionality
- Not care how results are displayed
- Only one that can talk to the model or the view

Separation of concerns

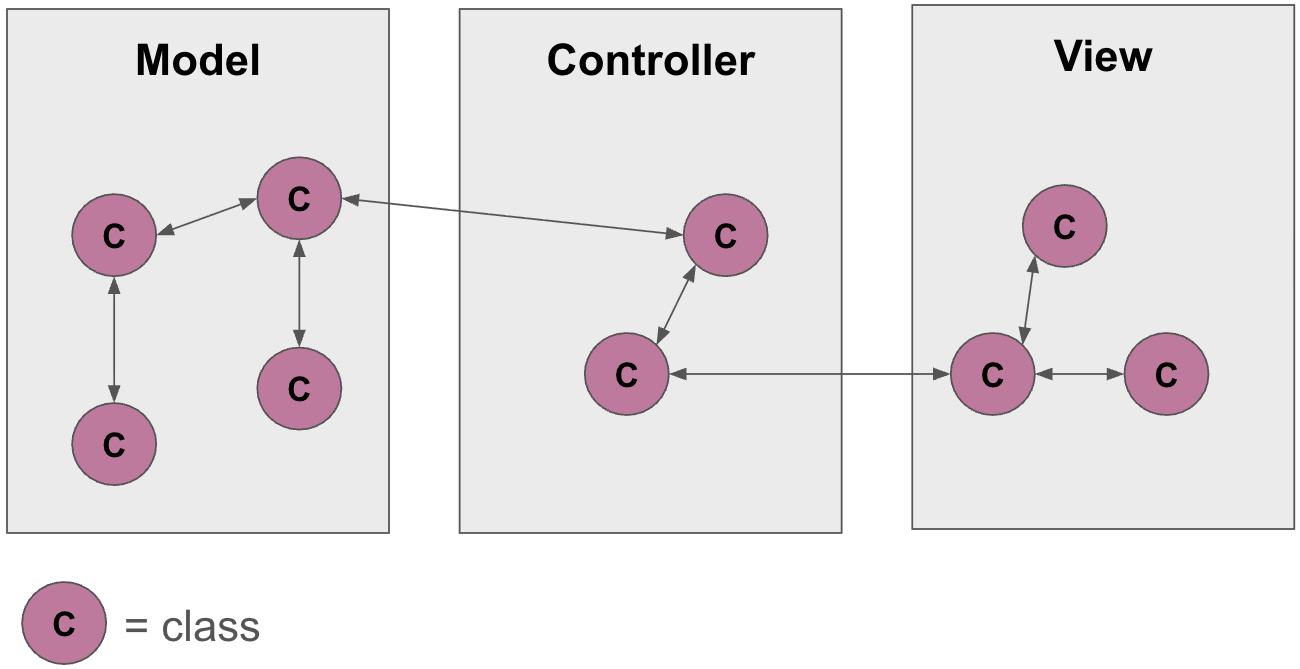
Good separation of concerns:



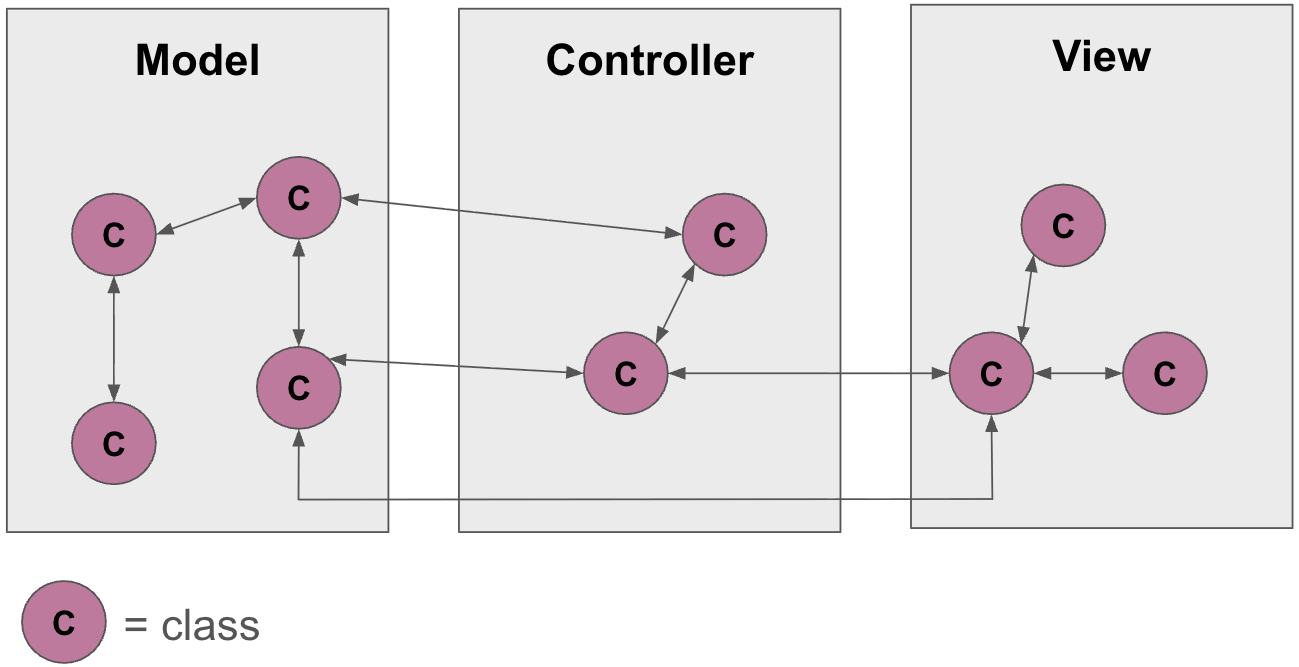
Bad separation of concerns:



Good separation of concerns (communication):



Bad separation of concerns (communication):



Examples

IDE (VSCode)

- Model: compile code, run code, decide syntax highlighting, interact with file system, all thinking stuff...
- View: Everything the user sees: text editor view, showing the output of code, the GUI
- Controller: tell model/view what to do (tell the model that the user clicked on Run so now please run the code), take user input (read the keys/clicks)

Pawtograder (or Canvas)

- Model: storing people, storing assignments, enforcing grades/due dates
- View: what the webpage looks like
- Controller: waits for someone to click the button, and when they do, tell the model do to the thing the button says, or tell the view to display what the user asked for

  Note: Pawtograder does not actually follow MVC, but it could if we wanted it to. Pawtograder's is closer to MVVM.

Entrypoint / main driver class

Okay there is *one* class outside of the Model, View, or Controller: the Main or Driver class. This is all it does:

- Instantiate the Model
- Instantiate the View
- Instantiate the Controller, passing the Model and View as args to its constructor
- Hand over control to the Controller (using something like `controller.go()`)

Separating extra concerns out of the Model

Let's write the Model for a game of Tic Tac Toe.

Avoid:

- Playing when it's not your turn
- Playing a "cell" that is outside the grid
- Playing a cell that's already taken
- Playing after the opponent has already won

These things should not be allowed by the Model (it should prevent or raise errors in these cases).

Let's write an interface.

Methods in tic tac toe model

- `def place(row: int, col: int) -> None`
  - Start with separate methods for `play_X()` and `play_O()`
  - But that design depends on the Controller to know which method to call (to prevent playing out of turn)
  - So instead, the Model will keep track of whose turn it is, and we just have the single `place()` method
  - Comments: `ValueError` if invalid position, cell is taken, or game is over
- `def get_next_player() -> Player`
  - To tell the Model whose turn it is (since we combined `play_X()` and `play_O()`)
  - `ValueError` if game is over
  - Returns enum

Methods in tic tac toe model

```
from enum import Enum

class Player(Enum):
  X = 1
  O = 0
```

- `def is_game_over() -> bool`
- `def get_winner() -> Optional[Player]`
  - What if draw?
  - Cannot "nobody" to enum because of `get_next_player()`
  - `None` makes sense because it means there is no winner
  - Also returns `None` if game is not over
- `def get_board() -> list[list[Optional[Player]]]`

Poll: Which of these can we prevent just by designing the Model interface well?

1. Playing out of turn
2. Playing a cell that is already taken
3. Playing after the game has ended
4. Playing a cell that doesn't exist

Hint: the rest will need to be prevented by raising errors in the class that implements the interface

Tests:

- `play()`
  - General valid cases
  - Edge cases
  - Moving to invalid cell
  - Moving to occupied cell
- `get_next_player()`
  - At least one for each player
  - `ValueError` after game is over
- `is_game_over()`
  - At least one returns false
  - At least one for boundary condition for false (last move)
  - At least one returns true
- `get_winner()`
  - At least one for X, at least one for O, at least one for draw
  - At least one for after game is over

Poll:

1. What is your main takeaway from today?

2. What would you like to revisit next time?