

A decorative graphic on the left side of the slide consisting of two overlapping parallelograms. The front one is blue and the back one is a light green. They are positioned diagonally, with the blue one partially covering the green one.

MVC and other Design Patterns

CS 2112 Staff



A motivating example

Suppose you are building a video game like Minecraft

The game should work on different platforms (i.e. iOS, Android, Mac OS, Windows)

All of these platforms use different GUIs to interface with the native OS

Need to be customized for each OS

Input on Desktop and mobile is different (keyboard vs. touch)

We want to keep the core game logic the same across all platforms, but need different input methods and different GUIs for the various platforms

Solution: Model-View-Controller design pattern

Separate program into three parts: Model, View, and Controller



Model

- Stores **state** of the program
- Does not see or modify the View or Controller
- Methods mainly consist of getters and setters
- For minecraft: model represents the blocks in the world, which could be represented as a command line text version



View

- Displays the GUI version of the model
- For minecraft: visualizes the blocks of the world
- Only **gets** information from the model, never *directly* changes the model
- Does not store the state of the program, only the state of UI elements.



Controller

- Handles user input
- Based on that input, updates the model and/or the view
- For desktop: input is keyboard based
- For mobile: input is touch based



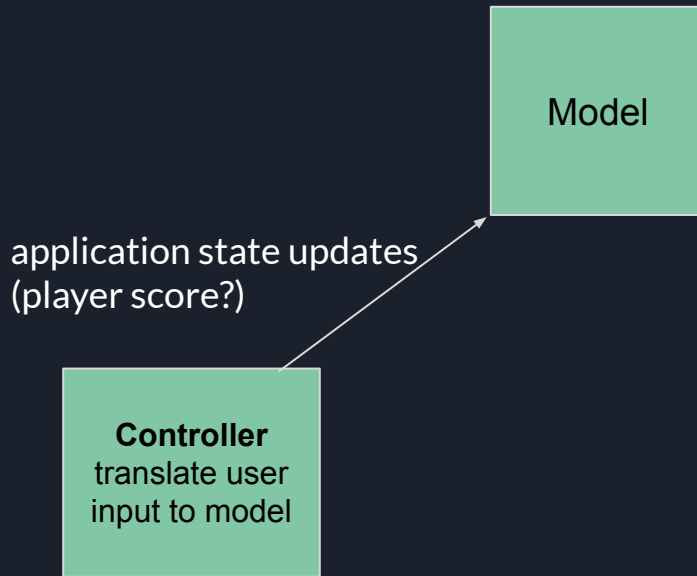
Example: Among Us

Model
application state

Things to store in the model:

- Who is the imposter?
- What's the world position of player X?
- Which tasks have been completed?

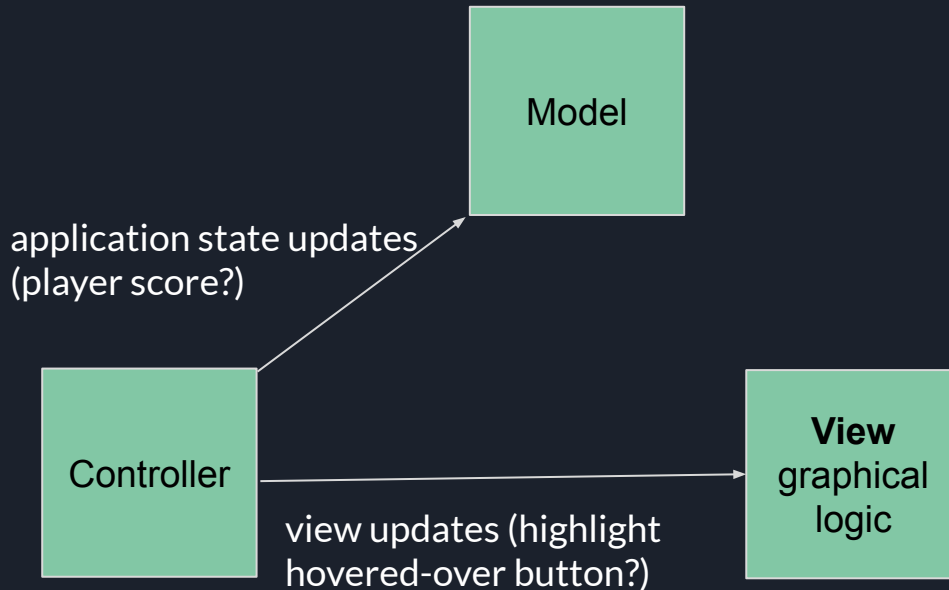
Example: Among Us



How should the Controller interact with the Model?

- Process “joystick tilted left” event, update player X’s location in the model to move left
- Process button press on “emergency meeting” button, update Model to encode that there’s an “emergency meeting” occurring in the game

Example: Among Us



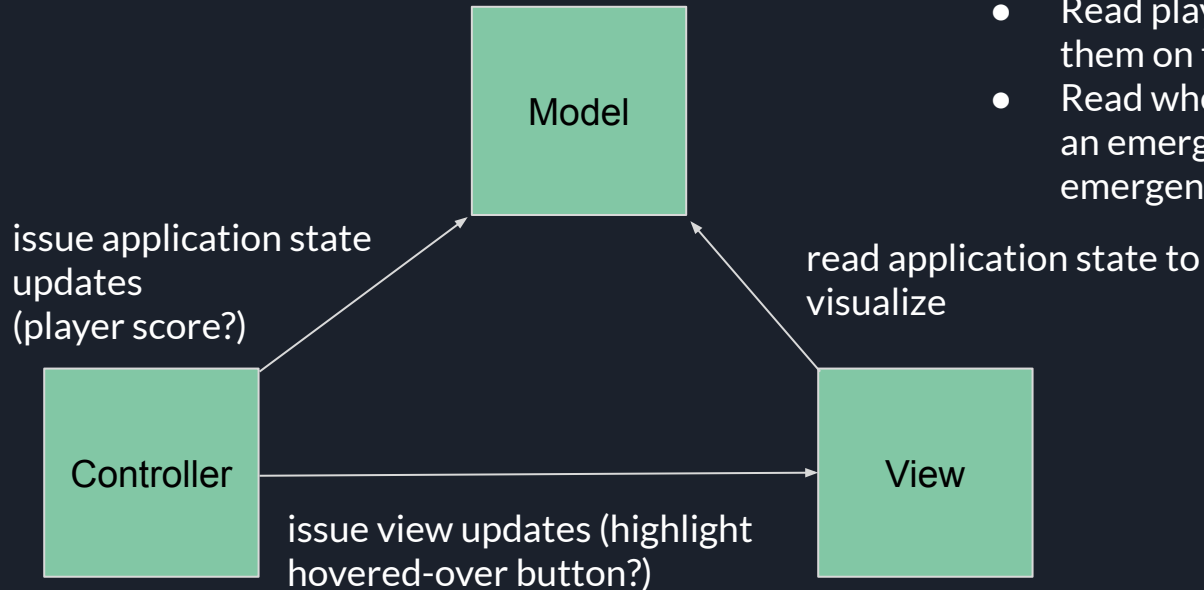
How should the Controller interact with the View?

- Process “tapped Settings button” event, tell View to show Settings menu

What should we store in the View?


- Position of buttons
- Screen positions of players (calculated from logical “world” positions)

Example: Among Us




How should the View interact with the Model?

- Read player positions, draw them on the screen
- Read whether or not there's an emergency meeting, show emergency meeting screen




Question: In Among Us, which part of MVC would store the players still alive?

- A. Model
- B. View
- C. Controller



Question: In Among Us, which part of MVC would store the display resolution of the game?

- A. Model
- B. View
- C. Controller



Question: In Among Us, which part of the MVC would note that the user who is the imposter wants to kill a player?

- A. Model
- B. View
- C. Controller



The Bottom Line

- Your model classes should not even know the controller and view exist
- Consider using interfaces or the Observer pattern to communicate
- Avoid cyclic dependencies
- Avoid having objects modify each other's internals directly. I.e. have mostly read-only interfaces
- One tip: have different people in your group work on model and view / controller for more natural separation of concerns



Minecraft: A Real-World Case Study

Minecraft Java Edition was originally single-player, and was very tightly coupled as a result

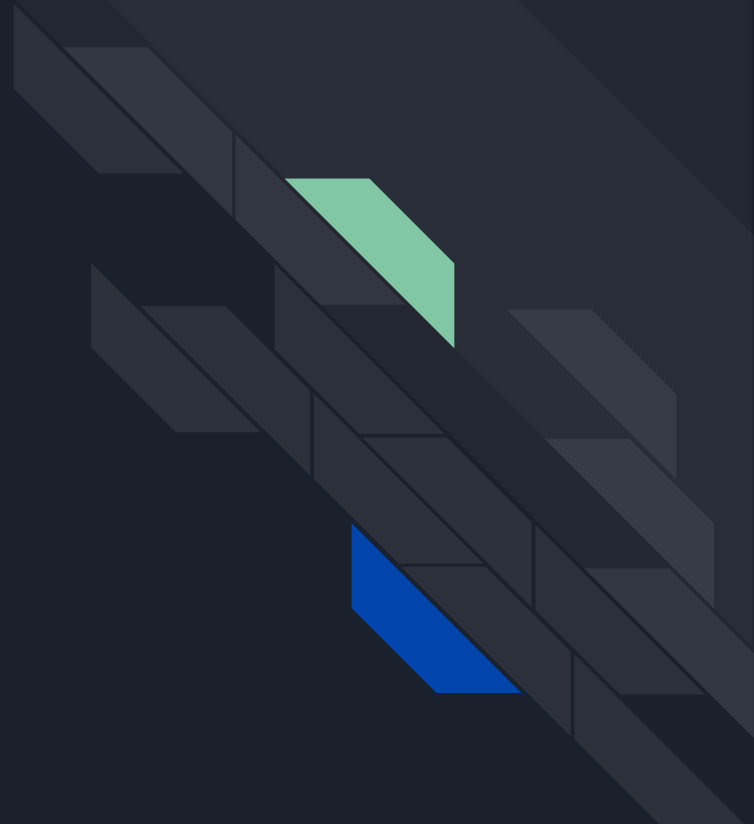
Minecraft then introduced multiplayer, which ended up using completely different code for logic

Required twice the work to maintain, and mods written for single-player didn't work on multiplayer and vice versa

Ultimately required a massive refactor of the code (update 1.3, if you're curious)

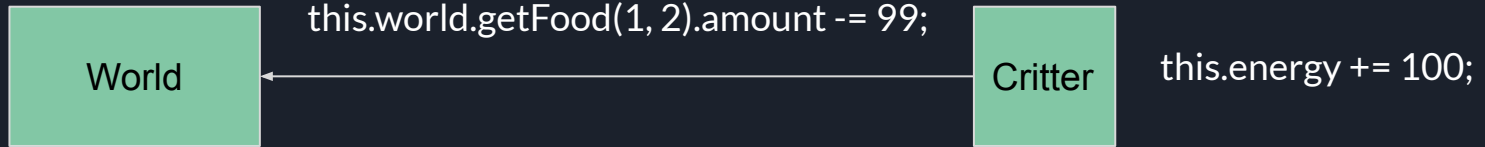
Modern Minecraft uses the same model code shared by both single and multiplayer

Tips for A5

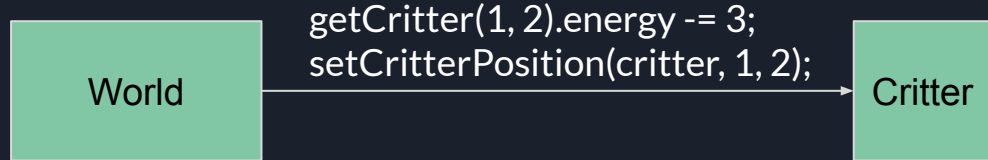


How to Introduce Cyclic Dependencies

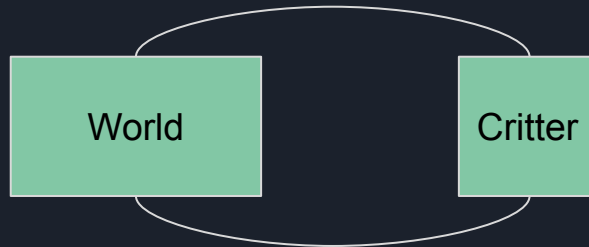
Example: Critter will completely eat the food in front of it.



What about moving critter forward? Easy:



Now we have cyclic dependencies!





Why cyclic dependencies are bad

Tight coupling between classes.

⇒ Hard to test critter in isolation!

⇒ Hard to reason about correctness of critter in isolation!

It's hard to disentangle cyclic dependencies once they have been introduced!



Avoiding Cyclic Dependencies using Observer

```
interface CritterObserver {  
    void onEatFood(Critter c, int n);  
}
```

```
class Critter {  
    CritterObserver o;  
  
    void eat(int amount) {  
        energy += amount;  
        o.onEatFood(this, amount);  
    }  
}
```

```
class World implements CritterObserver {  
    public void onEatFood(Critter c, int n) {  
        // decreases food amount or remove food  
    }  
}
```

```
class DummyWorldForTesting  
    implements CritterObserver {  
    public void onEatFood(Critter c, int n) {  
        // do nothing!  
        // The test only cares about critter  
        // energy being correctly changed!  
    }  
}
```

Although critter still implicitly holds a reference of the world, the implementation of world is decoupled from the critter class.



Avoiding Cyclic Dependencies using Controller

```
class Controller {  
    World w;  
  
    void step() {  
        for (Critter c : w.getCritterQueue()) {  
            // ...  
            // eat case:  
            EatAction.run(w, c);  
        }  
    }  
  
    private void run(World w, Critter c) {  
        w.removeFood(...);  
        c.increaseEnergy(...);  
    }  
}
```

With controllers, we can completely break cyclic dependency between World and Critter.

Now World and Critter only has a simple compositional relationship!



Testability & Modularity

- It's important to **separate each piece of functionality** so it can be easily tested
 - Also makes it easier to divide up work!
- Try writing method signatures and interfaces **first**
- Make sure your methods directly pertain to the functionality of the class they're in
 - Don't write controller code in your models!



Testing early by coding against interfaces

You are in a group and you are the one who implements the critter program interpreter.

The interpreter needs not only the program, but also some world information.

Does that mean you can't test it before your world is complete? NO!

```
interface ReadonlyWorldForInterpreter { int nearby(Critter c, int d); }

class Interpreter {
    int interpreterNearby(ReadonlyWorldForInterpreter w, Expression e);
}

class WorldForTestingNearby implements ReadonlyWorldForInterpreter {
    int nearby(Critter c, int d) { return 65536; }
}
```



Write more tests!



Read-only interfaces

- Recall rep exposure: we don't want to give the user access to our internal representations
 - They could break class invariants, delete things, etc
- How do we restrict the user's access to the internals while still giving them a full representation of the object?
 - Provide a read only interface (like in the source code)



Modifying the world

If the world is read only, how do we modify it?

- Only the **user** needs to see a read-only world
 - Recall the written problems from A3.
- The controller needs a modifiable model in MVC

Antipattern: God Object

```
class CritterWorld
```

```
-----  
void step();  
Stuff getContent(int r, int c);
```

```
Location getCritterLocation(Critter c);  
void moveCritter(Critter c, boolean forward);  
Location getForward(Location l);
```

```
void attack(Critter c1, Critter c2);  
void killCritter(Critter c);  
int getDamage(Critter c1, Critter c2);
```

```
void turn(Critter c, boolean left);
```

You happily started with this small CritterWorld class.

You want to implement forward and backward, so let's add a few methods.

You want to implement attack, so let's add more methods.

Turn looks easy, let's add that.

Another 500+ lines of code for other easy actions...

```
void mate();  
boolean canMate(Critter c1, Critter c2);  
.....
```

Finally, we can get to mate.

You end up with a 1000+ LOC CritterWorld class, potentially with a lot of copy-pasta. 🤔



Managing Complexity through Abstraction

How do we avoid this?

- Separate out logic about the game board and game rules
- Use an intermediate object to perform actions
 - Is performing critter actions really a responsibility of the world?
 - Also helps resolve cyclic dependencies
- Smaller classes are easier to read, debug, and test
- Breaking your problems into more pieces is good!



Over-Engineering

Don't create an entire class for something a data structure could represent in one line!

Do create a new class if the data structure requires many complex operations to represent your data correctly.

[Hello World Enterprise Edition](#)