Model View Controller

Table of contents

See csse1001[1] for course logistics — this note covers Lecture 11A's technical content. See [mvc] for the full reference on the Model-View-Controller pattern.

Advanced inheritance (recap)

A quick recap table from the end of last week, tying MRO complexity to each inheritance model (see python-inheritance[2] for the full detail):

Inheritance type	MRO complexity
Single	Simple linear order
Multilevel	Chain-like resolution
Multiple	C3 linearization
Hybrid (e.g. diamond)	C3 linearization

A few extra facts about MRO worth reinforcing:

- Every class has an MRO, computed the moment the class is *defined* (not when it's first used) — inspect it with `ClassName.mro()` or `ClassName.__mro__`.
- The MRO is built by the **C3 linearization** algorithm and depends entirely on the inheritance structure.

[1] courses/csse1001/csse1001.pdf
[2] courses/csse1001/python-inheritance.pdf

- If two base classes define the same attribute/method, whichever appears **earlier** in the MRO wins.
- Inside a method of an object of type `MyClass`, every `super()` call — whether it's in `MyClass` itself or in one of its parents/grandparents — refers to the *next class after the current one on `MyClass`'s MRO* (not the next class up from wherever the method happens to be defined).

MVC

The **Model-View-Controller (MVC)** pattern separates an application into three interconnected components:

- **Model**: stores the data and business rules of the application; responsible for every state change.
- **View**: some (usually visual) representation of the data, without altering it.
- **Controller**: an interface that receives user input (commands), tells the Model what to do, then selects a View to present the result.

Anatomy of MVC

Layer	Core question	Owns	MUST NOT
Model	What *is* the data?	Data & rules	`print`, parse args, call `input()`
View	How is info *shown*?	Formatting & rendering	change state, validate data
Controller	What happens *next*?	Workflow & commands	store raw data, format output

> The Model knows nothing about the outside world — no printing, no argument parsing, no reading input. It's pure data and logic.

MVC flow

1. **User** issues a command.
2. **Controller** parses input and calls the appropriate **Model** method.
3. **Model** mutates state and returns domain data.
4. **Controller** selects a **View** and passes the data.
5. **View** formats the data and pushes it to stdout.
6. Control returns to the main loop, waiting for the next user action.

Why MVC?

- **Modularity** — swap in new models, views, or controllers without breaking the application.
- **Testability** — each component can be tested in isolation.
- **Separation of concerns** — each component has a distinct responsibility.

Exercise: a minimal todo list manager

Implement a minimal todo list manager with the MVC design pattern:

- `Task` + `TodoList` → **Model** (domain)
- `TodoView` → **View** (presentation only)
- `TodoApp` + run-loop → **Controller** (input parsing & orchestration)

Model

```
class Task:
    def __init__(self, title: str) -> None:
        self._title = title
        self._done = False

class TodoList:
    def __init__(self) -> None:
        self._tasks = []

    def add(self, title: str) -> Task:
        task = Task(title)
        self._tasks.append(task)
        return task

    def all(self) -> list[Task]:
        return list(self._tasks)
```

View

```
class TodoView:
    def show_task(self, task: Task) -> None:
        mark: str = "/" if task._done else "X"
        print(f"[{mark}] {task._title}")
```

```
    def show_list(self, tasks: list[Task]) -> None:
        print("\n My Tasks\n--------")
        for t in tasks:
            self.show_task(t)
```

Controller

```
class TodoApp:
    def __init__(self, todo: TodoList, view: TodoView) -> None:
        self._todo = todo
        self._view = view

    def handle(self, command: str) -> None:
        parts = command.strip().split(maxsplit=1)
        if not parts:
            return
        cmd: str = parts[0]
        if cmd == "add" and len(parts) == 2:
            self._todo.add(parts[1])
            print("Task added!")
        elif cmd == "list":
            self._view.show_list(self._todo.all())
        else:
            print("Commands: add <title>, list, quit")
```

Run loop

```
app = TodoApp(TodoList(), TodoView())
while True:
    user_cmd: str = input("$")
    if user_cmd.strip() in ("quit", "exit"):
        break
    app.handle(user_cmd)
```

```
$add Work on A2
Task added!
$add buy groceries
Task added!
$list

 My Tasks
```

```
--------
[X] Work on A2
[X] buy groceries
```

Tracing through `add Buy groceries` against the MVC flow above:

1. **User** issues `add Buy groceries`.
2. **Controller** (`TodoApp.handle`) parses the input and calls `TodoList.add`.
3. **Model** (`TodoList`) mutates state and returns a `Task`.
4. **Controller** selects a **View** method (`show_task`, or `show_list` for `list`) and passes the data.
5. **View** formats the data and pushes it to stdout.
6. Control returns to the main loop, waiting for the next command.

> Every `[X]` in the output above is correct, if a little misleading at first glance — `X` just means "not done" here (`mark = "/" if task._done else "X"`), and nothing in this exercise ever sets `_done = True`. It's not a bug, just an unused feature stub left for extension.

Worked example: a larger system

`game.py` (a small pygame grid-world) is a good example of these same ideas showing up in a messier, more realistic setting:

- **Model**: `Grid`, `Cell`, `Robot`, and the `Enemy` hierarchy hold all the state and rules — cell rewards, slipperiness, robot/enemy position, and score.
- `Enemy` is itself an **abstract base class** in the unenforced style from python-inheritance[3]: `act()` just raises `NotImplementedError`, and the two concrete subclasses `RandomEnemy` and `ChaserEnemy` each provide their own `act()` — `RandomEnemy` picks a random legal direction, `ChaserEnemy` greedily moves toward whichever direction minimises squared distance to the robot.
- **View + Controller**: `Game` owns *both* of these — `draw()`/`_draw_grid()`/`_draw_hud()`/`_draw_actor()` render the current state (View), while `handle_events()` reads keyboard input and directly drives the Model (`self.robot.move(...)`, `self.enemy.act(...)`, `self.enemy.check_collision(...)`) (Controller).

  > This is a useful contrast with the todo list example above: real GUI/game loops very often merge View and Controller into one class for pragmatic reasons (rendering and input handling are both tied to the same per-frame loop), even though the stricter three-way split is what's taught here. The important invariant that *is* preserved is the one in the "MUST NOT" table: the Model (`Grid`, `Cell`, `Robot`, `Enemy`) never calls `print` or `input()`, and knows nothing about pygame at all.

[3] courses/csse1001/python-inheritance.pdf

One subtlety in `handle_events()`: on every keypress the order is always robot moves, then the enemy acts, then collision is checked — so it's possible for the robot to step onto the enemy's *old* square and the enemy to then step away in that same tick, without a collision ever being registered. Not necessarily a bug, but worth noticing if the game feels like it "should" have caught you.

Summary

- MVC is a pattern, typically used in web/GUI applications.
- Separation of concerns makes code more organised.
- It's easier to test, maintain, and extend code built this way.
- It provides a solid foundation for future enhancements.
- Numerous variants of MVC exist.

Next: further design patterns (Lecture 11B).