Model View Controller

Table of contents

Advanced inheritance (recap) 1
MVC 2
Exercise: a minimal todo list manager 4
Summary 8

Lecture 11A, 14 October 2025. See csse1001[1] for course logistics and [mvc] for the full reference on the Model-View-Controller pattern.

Advanced inheritance (recap)

MRO complexity for 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:

- 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 on the inheritance structure.
- If two base classes define the same attribute/method, whichever appears **earlier** in the MRO wins.

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

- 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 on `MyClass`'s MRO* (i.e. the one after the current class, 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.

There is an asymmetry buried in that third bullet: the Controller is the only component that talks to both of the others. The View is never described as reading from the Model, and the Model is never described as talking to anyone.

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

> Model knows nothing about the outside world.

Unpacked against the Model row of the table, that means no printing, no argument parsing and no reading input — pure data and logic. [mvc] draws the dependency shape it implies.

MVC flow

The flow is a six-step cycle:

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.

The two things easiest to lose from that list are that step 3 hands data *back to the Controller* — not to the View — and that the View is chosen by the Controller *after* the Model has already run.

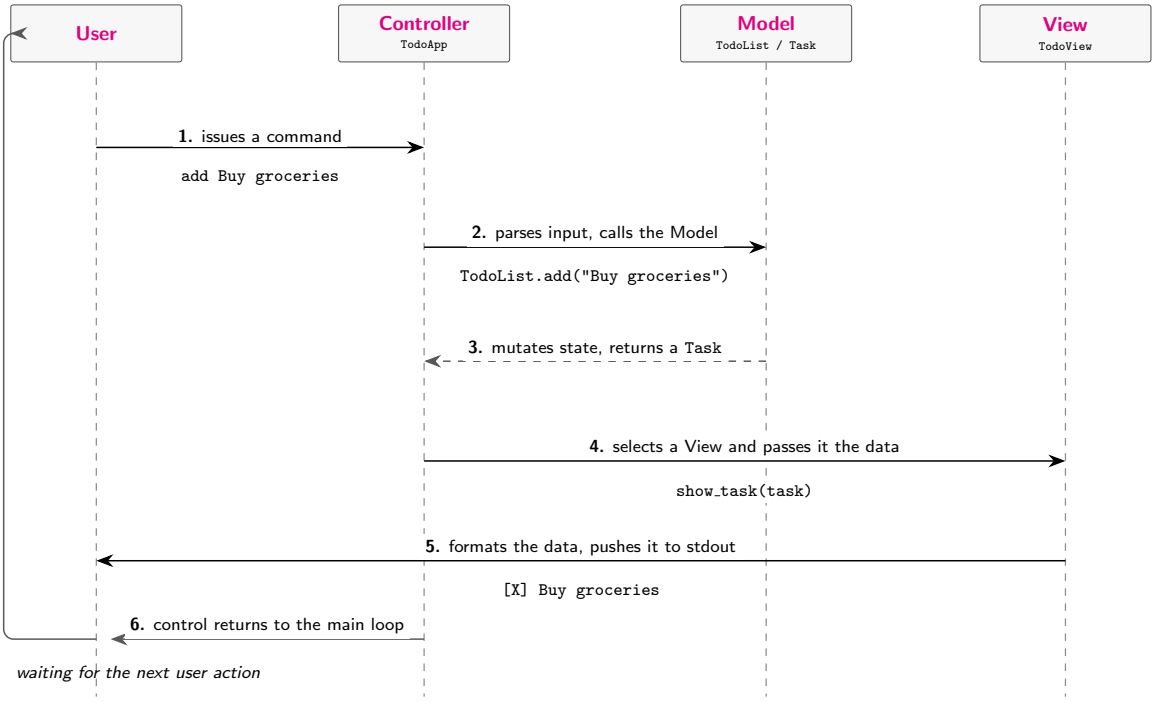

Figure 1: The six-step MVC flow with the todo-list example's names filled in. Reading down is reading forward in time. Step 3 is dashed because it is a return, not a call: the Model hands its domain data back to the Controller, which only then picks a View. The monospace lines beneath arrows 1, 2, 4 and 5 give the concrete values for the command `add Buy groceries`: the call to `TodoList.add`, which returns a `Task`, and then `show_task`. Steps 4 and 5 show the pattern, not a trace of the program: `[X] Buy groceries` is what `show_task` would print if the Controller called it, which for an `add` command the code never does.

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 MVC design pattern.

The classes are assigned to layers as follows:

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

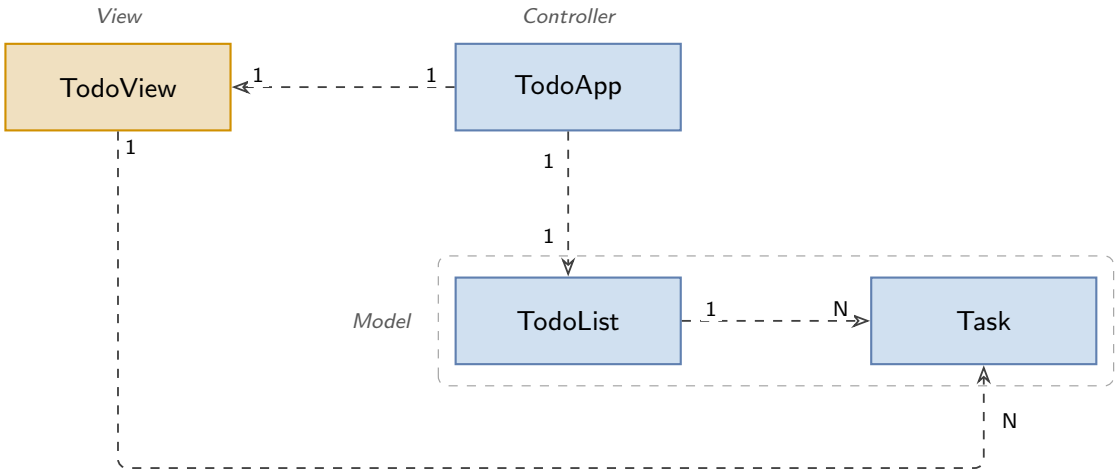

Figure 2: The four exercise classes and the one-way dependencies between them. Each class is a single-compartment box. A dashed line with a small open arrowhead means a class uses another class, with a bare multiplicity at each end. The Controller and Model classes are filled blue and the View class orange. The dashed boundary groups the two Model classes, `TodoList` and `Task`, and the three italic names label the layers. Every dependency shown is read off the constructor and method bodies of the listings below.

The direction of every dashed arrow matters. `TodoApp` knows about `TodoList` and `TodoView`; `TodoList` knows about `Task`; and `TodoView` knows about `Task` too, because `show_task(self, task: Task)` names the type and reads two of its attributes. Nothing points back the other way — no arrow from `TodoView` to `TodoList`, and none from either Model class to the Controller or the View. That one-way shape is the whole point of the *Anatomy of MVC* table, and it is drawn out in [mvc].

The four listings below are one program, split by layer.

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)
```

Two details worth noticing. `add` both mutates state *and* returns the domain object it created — that return value is step 3 of the flow. And `all` returns `list(self._tasks)`, a fresh copy rather than the internal list, so a View that iterates it cannot accidentally mutate the Model (see python-lists[3] on aliasing, and python-representation-invariants[4] on why handing out the internal list would be a leak).

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)
```

`TodoView` is pure presentation: it holds no state at all, and neither method returns anything. `show_list` delegates per-row formatting to `show_task` so the row format is defined in exactly one place.

[3] courses/csse1001/python-lists.pdf
[4] courses/csse1001/python-representation-invariants.pdf

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")
```

`TodoApp` is handed its Model and its View in `__init__` rather than constructing them — that's what makes the modularity claim from *Why MVC?* actually true here; you change the argument at the call site and nothing inside `TodoApp` changes.

`split(maxsplit=1)` is what lets a title contain spaces: `add Buy groceries` splits into exactly `["add", "Buy groceries"]`, never three parts.

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
```

`print("\n My Tasks\n--------")` would actually emit a blank line before `My Tasks`, and the transcript above does not show one.

Tracing `add Buy groceries` through the flow (Figure 1):

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** (e.g. `show_task`) 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.

Steps 4 and 5 there describe the *pattern*, not this program: for an `add` command the Controller calls no View at all — it prints `Task added!` from inside `handle` and returns, which is also the first of the three rule-bends below. `show_task` runs only by way of `show_list`, on a `list` command.

Every `[X]` in the transcript 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.

Three places the exercise bends its own rules

Read against the *Anatomy of MVC* table, the listings break its rules in three places. They don't change the answer to the exercise. They're worth noticing because the pattern is easier to remember once you've seen where a small example fails to honour it.

- **The Controller prints.** `handle` calls `print("Task added!")` and `print("Commands: ...")` directly, and the run loop's `input("$")` prompt is written in the Controller layer too. By the table's own third row a Controller "MUST NOT … format output". Doing it properly would mean a `TodoView.show_message` and routing both strings through it.
- **The View reaches into the Model's private attributes.** `show_task` reads `task._done` and `task._title`, both single-underscore "private by convention" names (python-classes-and-objects[5]). The View is behaving correctly in spirit — it only *reads* — but a `Task` with public accessors would keep the boundary honest.
- **`quit` is handled outside `handle`.** The help string advertises `quit`, but `handle` has no branch for it; the run loop intercepts `quit`/`exit` before `handle` ever sees them. Typing `quit` therefore works, while asking `handle` about it prints the help text.

[5] courses/csse1001/python-classes-and-objects.pdf

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.

> **The diagrams are not course notation**
>
> There is no established shape, layout, arrow direction, or colour for an MVC diagram in this course (see csse1001-diagram-notation[6]). Figure 2 uses the course's class-diagram conventions, apart from the dashed Model boundary and the three layer names. The layout of Figure 1 is not a course convention, and neither is the dependency figure in [mvc]. Don't reproduce them in an exam answer as though they were the course's notation.

[6] courses/csse1001/csse1001-diagram-notation.pdf