Lecture 9: Design Patterns

Model View Controller

MVC Design Pattern

MVC is considered an architectural pattern rather than a design pattern.

The difference between an architectural and a design pattern is that the former has a broader scope than the latter

Model-View-Controller

MVC is one of the most widely used patterns in architecture today

You will see it everywhere

It’s worth paying attention to this lecture even if your own personal design pattern is to sleep through them ;)

MVC: Model

The model is the core component. It represents knowledge. It contains and manages the (business) logic, data, state, and rules of an application.

MVC: View

The view is a visual representation of the model. Examples of views are a computer GUI, the text output of a computer terminal, a smartphone's application GUI, a PDF document, a pie chart, a bar chart, and so forth. The view only displays the data; it doesn't handle it.

MVC: Controller

The controller is the link/glue between the model and view. All communication between the model and the view happens through a controller.

MVC: Example

A typical use of an application that uses MVC, after the initial screen is rendered to the user is as follows:

1. The user triggers a view by clicking (typing, touching, and so on) a button

2. The view informs the controller of the user's action

3. The controller processes user input and interacts with the model

4. The model performs all the necessary validation and state changes and informs the controller about what should be done

5. The controller instructs the view to update and display the output appropriately, following the instructions that are given by the model

MVC: Why the Controller?

You might be wondering, why the controller part is necessary? Can't we just skip it?

We could, but then we would lose a big benefit that MVC provides: the ability to use more than one view (even at the same time, if that's what we want) without modifying the model. To achieve decoupling between the model and its representation, every view typically needs its own controller. If the model communicated directly with a specific view, we wouldn't be able to use multiple views (or at least, not in a clean and modular way).

Model-View-Controller

MVC is a very generic and useful design pattern. In fact, all popular web frameworks (Django, Rails, and Symfony or Yii) and application frameworks (iPhone SDK, Android, and QT) make use of MVC or a variation of it—model-view-adapter (MVA), model-view- presenter (MVP), and so forth. However, even if we don't use any of these frameworks, it makes sense to implement the pattern on our own because of the benefits it provides, which are as follows:

MVC Benefits

The separation between the view and model allows graphics designers to focus on the UI part and programmers to focus on development, without interfering with each other.

Because of the loose coupling between the view and model, each part can be modified/extended without affecting the other. For example, adding a new view is trivial. Just implement a new controller for it.

Maintaining each part is easier because the responsibilities are clear.

MVC: how to implement

When implementing MVC from scratch, be sure that you create smart models, thin controllers, and dumb views.

MVC: Smart Models

A model is considered smart because it does the following:

Contains all the validation/business rules/logic

Handles the state of the application

Has access to application data (database, cloud, and so on) Does not depend on the UI

MVC: Thin Controllers

A controller is considered thin because it does the following:

Updates the model when the user interacts with the view

Updates the view when the model changes

Processes the data before delivering it to the model/view, if necessary Does not display the data

Does not access the application data directly

Does not contain validation/business rules/logic

MVC: Dumb Views

A view is considered dumb because it does the following:

Displays the data

Allows the user to interact with it

Does only minimal processing, usually provided by a template language (for example, using simple variables and loop controls)

Does not store any data

Does not access the application data directly Does not contain validation/business rules/logic

MVC: Checking your instincts

If you are implementing MVC from scratch and want to find out if you did it right, you can try answering some key questions:

If your application has a GUI, is it skinnable? How easily can you change the skin/look and feel of it? Can you give the user the ability to change the skin of your application during runtime? If this is not simple, it means that something is going wrong with your MVC implementation.

If your application has no GUI (for instance, if it's a terminal application), how hard is it to add GUI support? Or, if adding a GUI is irrelevant, is it easy to add views to display the results in a chart (pie chart, bar chart, and so on) or a document (PDF, spreadsheet, and so on)? If these changes are not trivial (a matter of creating a new controller with a view attached to it, without modifying the model), MVC is not implemented properly.

MVC Example

Idea: Quote Printer.

The user enters a number and sees the quote related to that number. The quotes are stored in a quotes tuple. This is the data that normally exists in a database, file, and so on, and only the model has direct access to it.

MVC Example

```
quotes = (
'As I said before, I never repeat myself.',
    'Black holes really suck...',
    'Facts are stubborn things.'
     ...
)
```

Example: Model

The model is minimalistic; it only has a get_quote() method that returns the quote (string) of the quotes tuple based on its index n. Note that n can be less than or equal to zero, due to the way indexing works in Python.

```
class QuoteModel:
    def get_quote(self, n):
        try:
            value = quotes[n]
        except IndexError as err:
            value = 'Not found!'
return value
```

Example: View

The view has three methods: show(), which is used to print a quote (or the message Not found!) on the screen, error(), which is used to print an error message on the screen, and select_quote(), which reads the user's selection.

Example: View

```
class QuoteTerminalView:
    def show(self, quote):
        print(f'And the quote is: "{quote}"')
    def error(self, msg):
        print(f'Error: {msg}')
    def select_quote(self):
        return input('Which quote number would you like to see? ')
```

Example: Controller

The controller does the coordination. The __init__() method initializes the model and view. The run() method validates the quoted index given by the user, gets the quote from the model, and passes it back to the view to be displayed as shown in the following code:

Example: Controller

```
class QuoteTerminalController:
    def __init__(self):
        self.model = QuoteModel()
        self.view = QuoteTerminalView()
    def run(self):
        valid_input = False
        while not valid_input:
            try:
                n = self.view.select_quote()
                n = int(n)
                valid_input = True
            except ValueError as err:
self.view.error(f"Incorrect index '{n}'")
        quote = self.model.get_quote(n)
self.view.show(quote)
```

Example: Main

Last but not least, the main() function initializes and fires the controller as shown in the

following code:

```
def main():
    controller = QuoteTerminalController()
    while True:
controller.run()
```