TABLE OF CONTENTS

REFACTORING PLAN

1. SWITCH STATEMENT – `CONTROLLER().DISPLAY()`

This was the worst smell because it covers both the `Controller` and `Visualiser` classes, and it means I have to re-write most of the `Visualiser` class. I will be using polymorphism and implementing an inheritance hierarchy, which might end up being a lot more work than the other smells I have found.

Location:

Package:	`Interpreter`
File:	`controller.py`
Class:	`Controller`
Method:	`display()`
Lines:	`26 - 33`

Reasons:

- The code block in the `Controller.display()` method that is a **switch statement** that compares an input passed into the method and calls a different `display_chart()` method of the `Visualizer` class depending on what conditional branch was executed.
- This area of the program breaks the Open/Close principle, meaning that it is not easily extensible. To extend the functionality to display more chart types I would have to add another branch to the if/else in `Controller.display(),` and add a whole new method in `Visualizer` for the new chart type, but it would contain more of the same duplicate code.

Strategies/ approaches:

1. **Extract Method:** Firstly, I will extract the switch code block into a private method inside Controller.
2. **Move method:** Next, I will move the method into the Visualizer class.
3. **Replace Conditional with Polymorphism:** Lastly, I will replace the switch statement with an inheritance hierarchy. This will reduce the duplicate code, as well as following the Open/Close principle, making it easy to extend in the future.

2. FEATURE ENVY – `CONTROLLER().DISPLAY()`

This was chosen as the second worst smell because it covers two classes (`Controller` and `Validator`) again, and I will have to re-write and create multiple methods.

Location:

Package:	`Interpreter`
File:	`controller.py`
Class:	`Controller`
Method:	`display()`
Lines:	`17, 18`

Reasons:

- This is the third bad smell I chose because it affects multiple modules, controller.py and validator.py.
- It is bad because the controller needs to check if some of its data is valid. It does this by accessing the data stored in the validator and performing the comparison in the controller.
- This is feature envy because all of the data is being brought into the controller and being used there.

Strategies/ approaches:

1. **Extract method**: First I will extract the logic that compares the input data with the list of valid data columns, and turn it into its own method inside Controller.
2. **Extract method**: Second, I will do the same with the logic that compares the input parameter with the list of valid flags, turning it into its own method inside the Controller.
3. **Move method**: Then, I will move the first method into the Validator class and re-write the initial logic into the new function call.
4. **Move method**: And I will do the same for this second method. If everything goes well this shouldn’t introduce any new bugs!

3. LARGE CLASS – `CONTROLLER()`

This bad smell was chosen as the third worst because it means creating a whole new class and moving the functionality into it. I have a plan to do it but I am unsure if complications might arise, and it means re-writing some calling code all through the application.

Location:

Package:	`Interpreter`
File:	`controller.py`
Class:	`Controller`
Method:	`serialize ()`
Lines:	76

Reasons:

- The controller class is used to manage the flow of data between the view (`CmdView`) and the various models (`Validator`, `DataParser`, `Visualiser`, `DatabaseView`), but near the end of the project it started to contain functionality that would be better suited in its own class, for example, the serialization functionality.
- I chose this as the next important bad smell to fix because it was cluttering up the `Controller` and making it confusing to understand what it should be doing. It will also make it easier to clean up the serialize functionality when it is in its own class.

Strategies/ approaches:

1. **Extract Class:** Mainly, I will move the `serialize` method into its own class. I will do this by first creating a new module called `Serialize.py`, and define a new class `Serializer`.

   I will then move the `serialize()` method from the `Controller` and into the new `Serializer` class, and replace the old function with a call to the new class method.

4. SPECULATIVE GENERALITY – `VIEW`

This was chosen as the last smell because it affects multiple modules (`fileview.py`, `databaseview.py`, `cmdview.py`) and is adding unnecessary complexity to my application through a useless inheritance hierarchy, and is obfuscating some of the functionality of these classes because of the bad naming convention I chose for the `View` interface.

Location:

Package:	`Interpreter`
File:	`fileview.py, cmdview.py, databaseview.py`
Class:	`FileView, CmdView, DatabaseView`
Method:	`get(), set()`
Lines:	`fileview.py -> line: 44, line: 10`
	`cmdview.py -> line: 92, line: 89`
	`databaseview.py -> line: 38, line: 62`

Reasons:

- I attempted to create an interface called `View` in order to generalize the reading and writing of data to and from different domains with `get()` and `set()` methods.. I then implemented this interface with `CmdView`, `FileView`, and `DatabaseView`.
- The problem is that I never ended up implementing all of the functionality. `DatabaseView` implemented both `get()` and `set()`, but `FileView` and `CmdView` only implemented `get()`, while `set()` remained un-implemented.
- The empty `set()` methods are cluttering up the respective classes.
- Using such basic names as get and set makes the functionality of the modules vague. Removing the interface would mean these methods could be renamed to better describe their behaviour, making the code easier to understand in these three modules.

Strategies/ approaches:

2. **Collapse Hierarchy:** I will first remove the implementing classes dependency on the interface by removing the empty `set()` methods, and making them no longer inherit from `View`. Once this is done I can safely delete the `View.py` ABC module.
3. **Rename Method:** Next, I will rename the methods that were used by the interface, and name them to describe their behaviour more clearly.

   Sub-Steps: *[old-name] > [new-name]*

   1. `FileView.get() > FileView.read_file()`
   2. `DatabaseView.set() > DatabaseView.insert()`

3. `DatabaseView.get() > DatabaseView.retrieve()`
4. `CmdView.do_get() > CmdView.do_read()`

4. **Rename Class:** I will also rename the classes because they no longer need to indicate an interface implementation. This will hopefully make their purpose easier to understand.

   Sub-Steps: *[old-name] > [new-name]*

   1. `FileView > FileReader`
   2. `DatabaseView > Database`

   I won't rename `CmdView` because I feel the name accurately reflects the module's purpose as a cmd-line interface (View in MVC). However, renaming `FileView` and `DatabaseView` makes sense considering they are no longer implementing the `View` interface.