# Design Patterns

#### Design Pattern 1: Template Method

Location (before):

FOLDER\_NAME = mih0760-umlify

FILE\_NAME = umlify\_component\_viewer.py

CLASS\_NAME = UmlifyComponentViewer

METHOD\_NAMES = generate\_pie\_chart(), generate\_pie\_charts(), generate\_bar\_chart(), \_get\_attribute\_types(), \_remove\_by\_key()

BETWEEN\_LINES = 67-69, 71-98, 100-143, 177-180

Location (after):

FOLDER\_NAME = generators

FILE\_NAME = chart\_generator.py, pie\_chart\_generator.py, bar\_chart\_generator.py

CLASS\_NAME = ChartGenerator, PieChartGenerator, BarChartGenerator

Notes:

* The two initial classes that generated the charts and diagrams, UmlifyComponentViewer and ComponentViewer, were refactored before implementing this design pattern. The reason was that they combined both methods of generating charts and diagrams when they should be in separate classes. Hence, the change of class names and extraction of methods into the appropriate classes.
* The VisualGenerator class is the main generator which holds the objects of UMLDiagramGenerator and ChartGenerator. This class is connected to the CommandLine class.

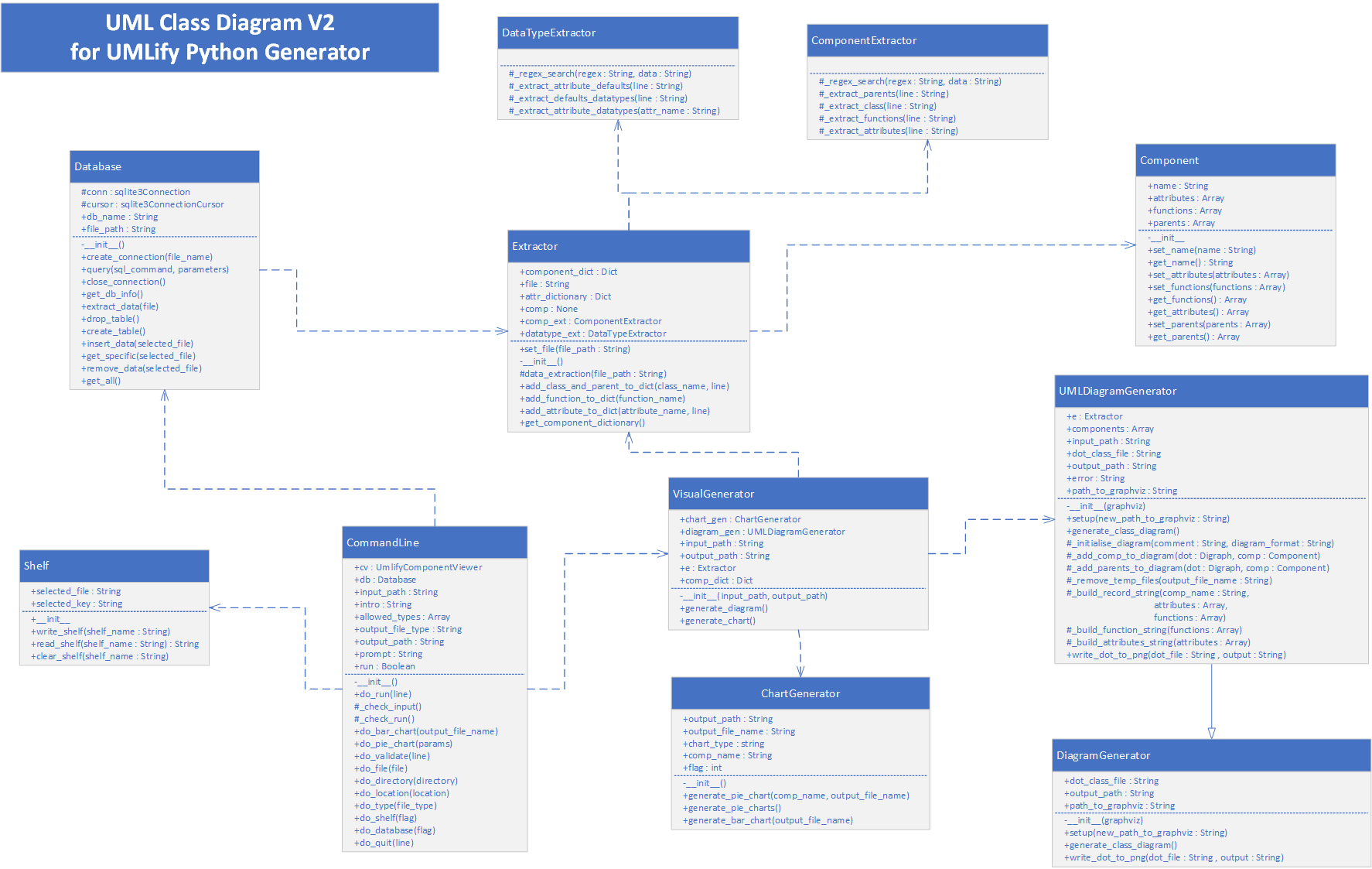
Reasons:

1. The steps of generating a pie chart and bar chart are both similar in terms of how they:
   1. Require an output file name to be set
   2. Setting up the labels, titles, and other details for that chart
   3. Setting the chart type
   4. And using pyplot.savefig() at the end to produce the charts into a specified format
2. The steps of setting labels and titles and its chart type would be different internally so these steps can be overridden from the parent class.
3. Both methods of generating charts uses the same program (pyplot) to create outputs.
4. By using template method, it would result in less duplicated code especially the method that sets the output file name and when running the pyplot method of building the chart.
5. This would allow sub-classes to re-define part of the logic (in Reason 1) for their own situation (e.g. producing a bar chart requires an axis whereas pie charts don’t).
6. This would allow more different types of charts to be produced just by creating a new subclass which would have the inherit the steps (algorithm) of its Parent class but will have varying details based on its type which can be modified by overriding the certain methods.

Class Diagram BEFORE Template Method:

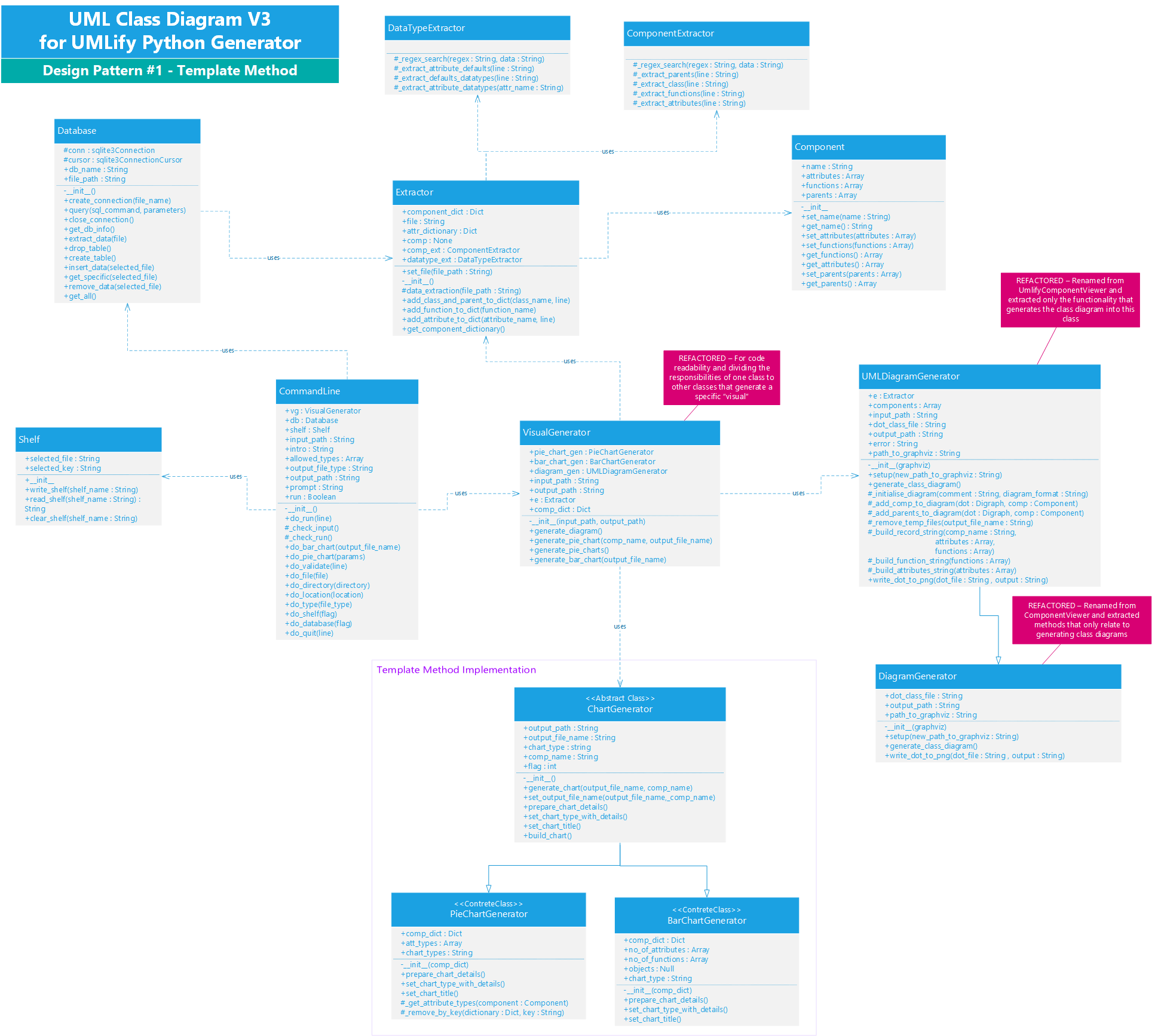
**Source:** before-pattern1-diagram.png

Red box shows where the design pattern would be implemented. Notes were made about the refactoring process before applying template method in the “after implementation” diagram.



Class Diagram AFTER Template Method:

**Source:** after-pattern1-diagram.png



#### Design Pattern 2: Observer

Location (Before):

FOLDER\_NAME = mih0760-umlify (master)

FILE\_NAME = commandline.py

CLASS\_NAME = CommandLine

BETWEEN\_LINES = 12-498

Location (After):

FOLDER\_NAME = mih0760-umlify (master)

FILE\_NAME = interpreter.py, controller.py, main.py

CLASS\_NAME = Interpreter, InterpreterController, Subject, Observer

Reasons:

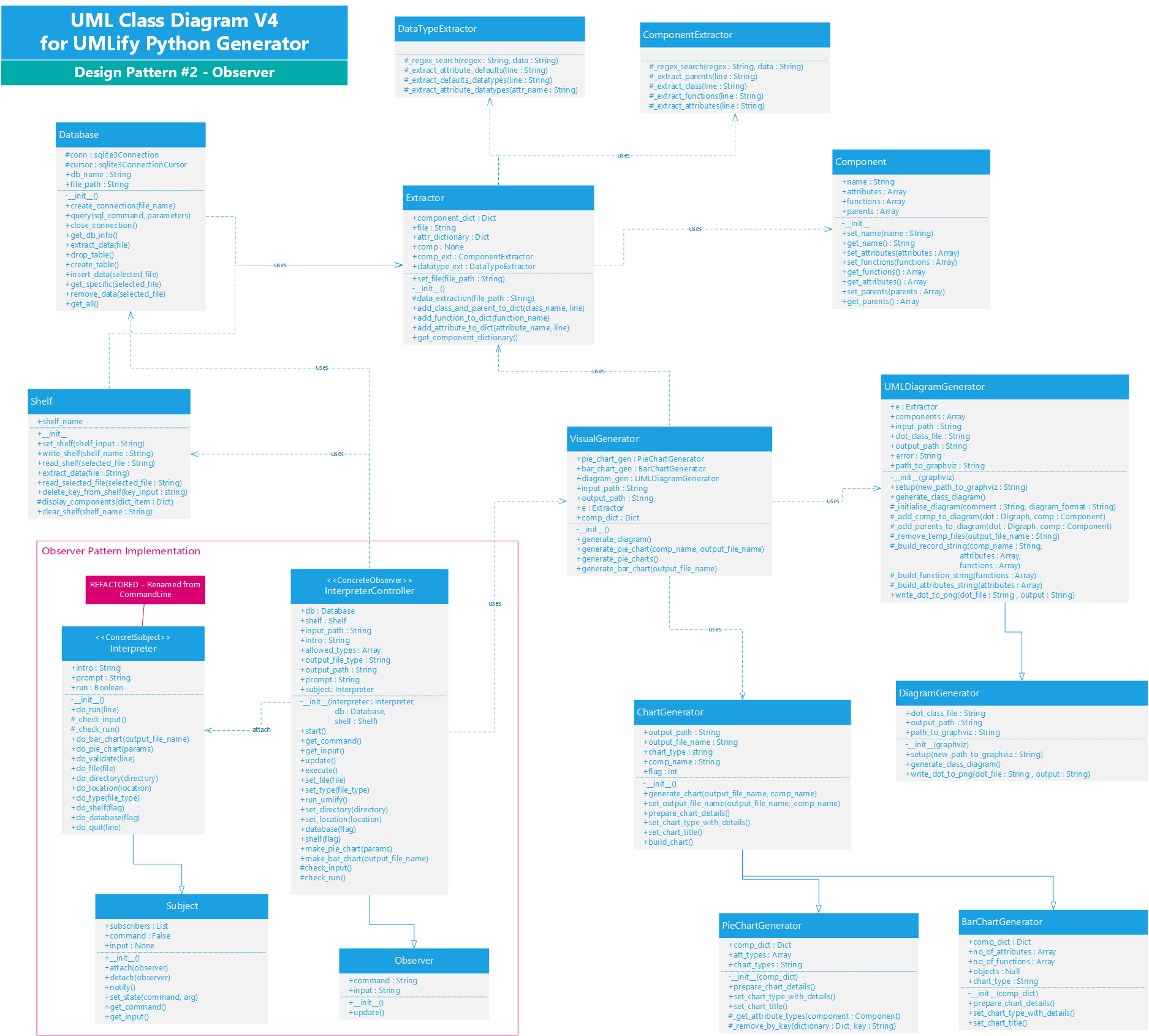
1. The CommandLine class acted as both handling user input and using that input in a business logic. So in MVC terms, it was both a model and a view (and a controller) which would result in more complexities later on and inflexibility when a new functionality (command) is required.
2. Observer pattern enables an MVC pattern to be constructed in the design. This reduces the number of methods within CommandLine class and delegate the responsibilities of getting user input from the console (view) and using those inputs in the business logic (model).
3. Delegating the responsibility for notifying the view and controller of new changes to the model prevents an unnecessary system overload which happens due to constantly asking the model whether any changes has been made.
4. This pattern would be suitable as it allows multiple views to be used due to using MVC, so instead of just using the console to interact with the user, a GUI application could be applied without having to modify the model to enable a GUI view. This change of view can just be done by the client.
5. All this is resulting in decoupling the model from the view and reducing the number of dependencies.
6. The Interpreter(model) would use the observer pattern to keep the InterpreterController (controller and view) updated on which commands the user has selected and what input it received, then sends it to the InterpreterController to be used.

NOTE: The InterpreterController acts as both the view and controller. The client is the main.py.

Class Diagram BEFORE Observer:

**Source:** before-pattern-2-diagram

Class Diagram AFTER Observer:

**Source:** after-pattern-2-diagram.png