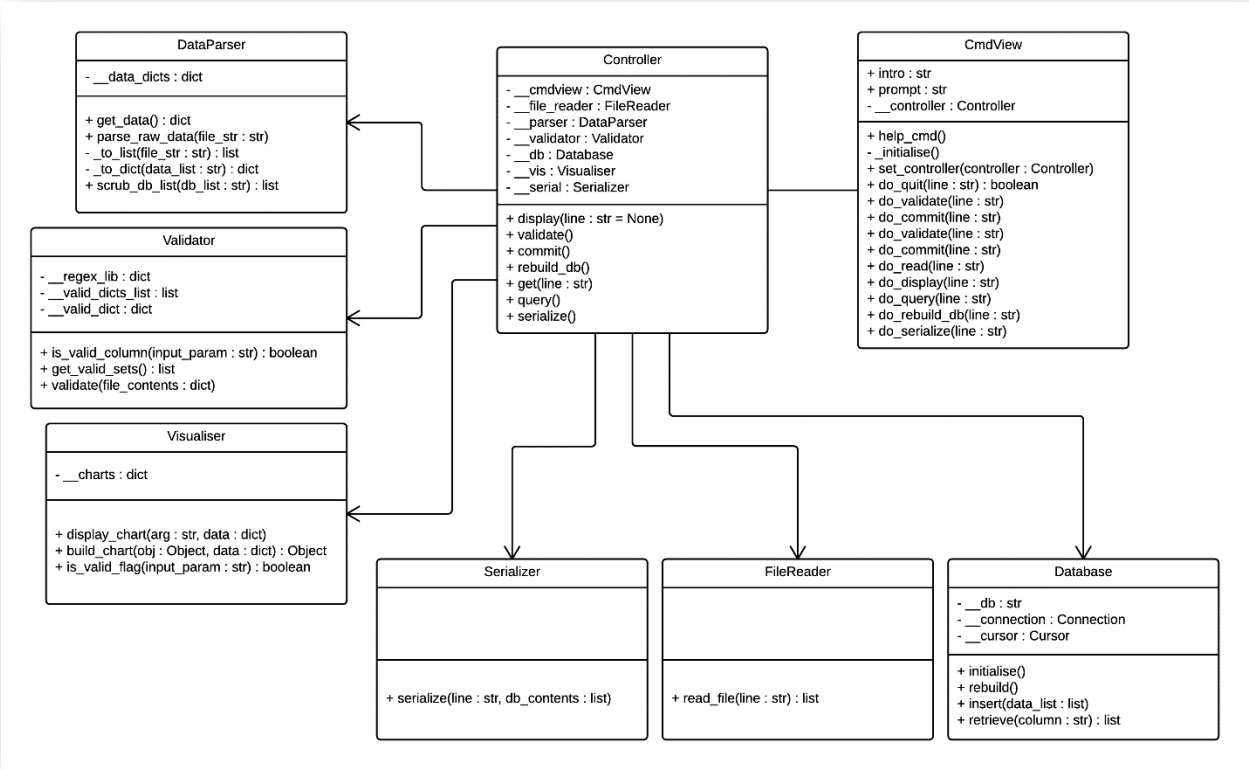
Contents

Pattern 1: **Builder**

Class Diagram: **Before**



Code Location

Assignment 2 *(Before applying the pattern)*:	**Assignment 3** *(After the pattern)*:
Files:	

`file_reader.py`	`file_reader.py`
	`file_read_director.py`
	`file_builder.py`
	`data_file.py`
	`txt_builder.py`
	`csv_builder.py`

Classes (Lines):	

`FileReader (30 – 39)`	`FileReader (9 – 13, 38 – 46)`
	`FileReadDirector (whole file)`
	`FileBuilder (whole file)`
	`DataFile (whole file)`
	`TxtBuilder (whole file)`
	`CsvBuilder (whole file)`

Justification

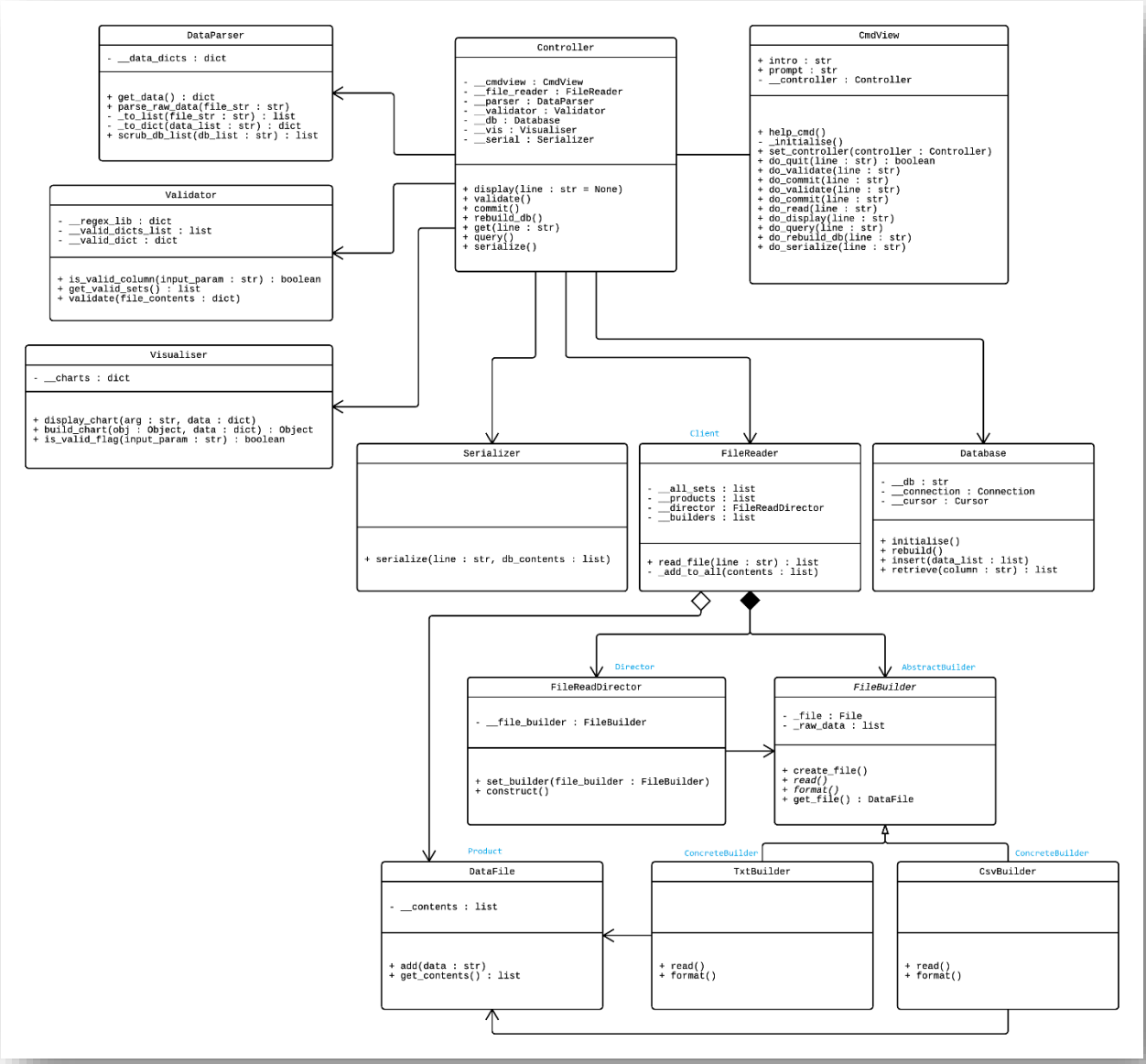
Currently, the FileReader class is hard-coded to only read .txt file types (`file_reader.py(before), line:31`). Trying to extend this to read other file types, or even use other read methods, would be difficult because it would mean a significant re-write of the whole class.

I decided this needs refactoring because reading more file types, and more reading methods, would be the most likely functionality to be extended for this application.

I decided to use Builder to refactor the FileReader. Using Builder allowed me to easily extend the functionality by adding an additional sub-class of the Builder base class with customised implementation for reading and building .csv files, and allowed me to clean up the client code.

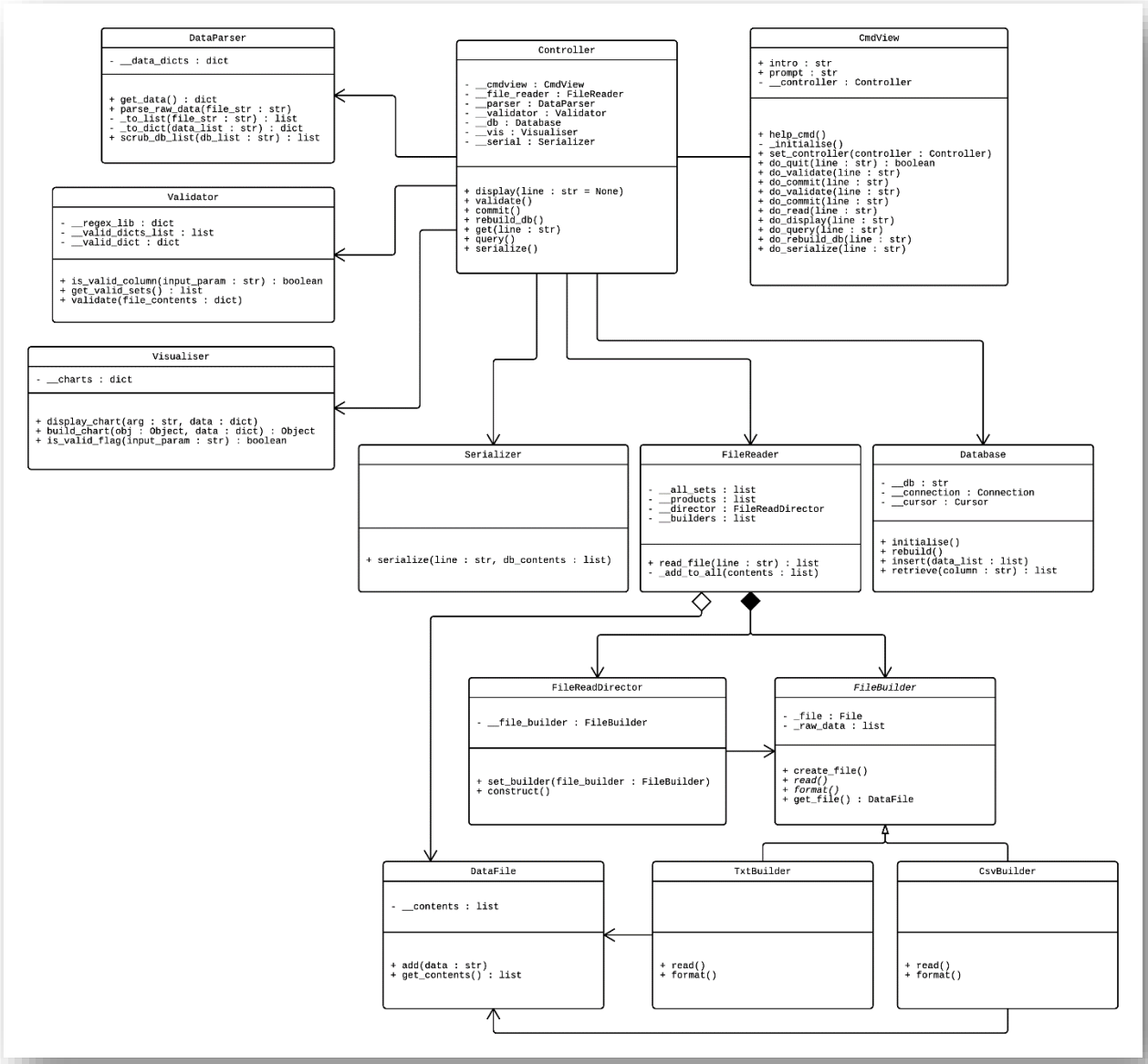
If I were to introduce more file types or read methods then all I would need to do is create another subclass, and add it to the new list of concrete builders in my FileReader (`file_reader.py(after), line:12-13, 38-41`).

Class Diagram: **After**



Pattern 2: **Flyweight**

Class Diagram: **Before**



Code Location

Assignment 2 *(Before applying the pattern)*:	**Assignment 3** *(After the pattern)*:
Files:	

`visualiser.py`	`visualiser.py`
	`chart_flyweight_factory.py`
	`chart_flyweight.py`

Classes (Lines):	

`Visualiser (7 – 30)`	`Visualiser (7 – 13)`
	`ChartFlyweightFactory (whole file)`
	`ChartFlyweight (whole file)`

Justification

The main reason I used Flyweight to refactor my Visualiser class was because it was improved when it was refactored in assignment 2, but it was still not easy to extend, and the logic to build the different parts of the chart were all tied into one method (`visualiser.py, line:17-24`).

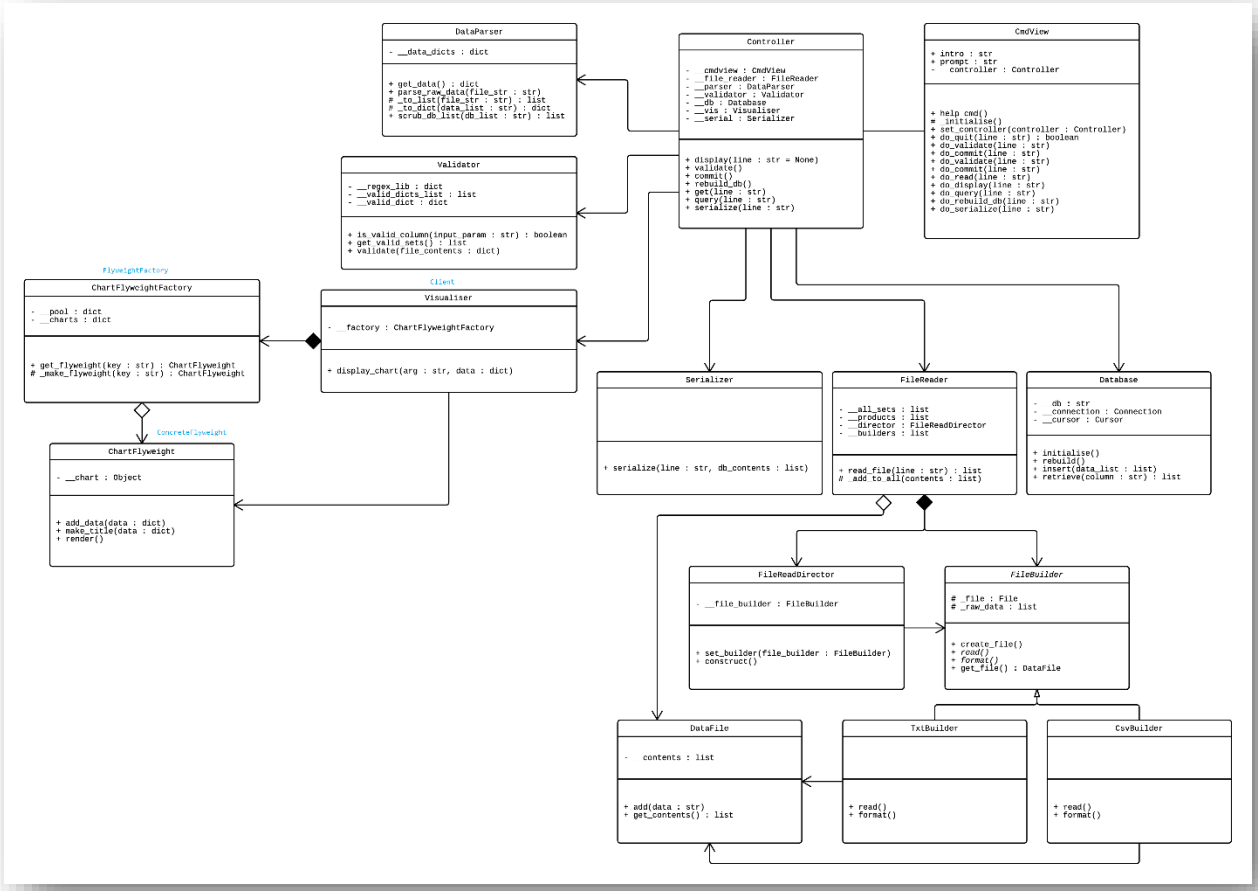
A second reason was that pygal objects were being instantiated when the application ran (`visualiser.py, line:7-10`), even though they may never be used, and then every time a chart was called.

By using Flyweight I could enable this part of the application to be more easily extended in the future (with more chart types), and also it would be a more efficient use of pygal objects by only storing one instance of a chart type when it is needed, and reusing it with different data as needed by the application (`chart_flyweight_factory.py, line:8-11`) and (`chart_flyweight.py, line:3`).

There is a drawback to my implementation. Currently, my pygal charts are built by passing in the same data regardless of chart types. This is a problem from my initial design, because some data might be well suited to be displayed in a bar chart, but that same data might not make sense as a pie chart. This flaw was introduced mainly due to time constraints (I designed with a bar chart in mind, and only added the other 3 chart types as an afterthought), and to change this would mean a significant amount of refactoring other parts of my application to allow for more flexible control over the data being displayed.

The side effect of this flaw, is that I implemented Flyweight using a single concrete Flyweight class that takes in its intrinsic state through its constructor. This allows me to create different Flyweight objects using the same class, because the implementation (how the chart is built) is not likely to change, only the extrinsic state (the data passed in) is different. But considering this flaw, it would not be hard to change my Flyweight implementation should the rest of my application be improved and fixed. In fact, now that I have implemented Flyweight, this area should be *easier* to modify (with an inheritance hierarchy for the Flyweights) to be in-line with any application-wide fixes, should the need arise.

Class Diagram: **After**



Self-Marking Evaluation

Part 2			
	Pattern 1	**Pattern 2**	
Design Pattern Applied	Builder	Flyweight	
Class Diagram Before	1.5	2	/2
Location of Code	2	2	/2
Name of Design Pattern	2	2	/2
Justification	2	1.5	/2
Class Diagram After	2	1.8	/2
Apply Design Pattern	10	9	/10
Sub-Totals	19.5	18.3	/20
	Total	37.8	/40