Design Patterns

For our application, we have identified a few design patterns that would suit our components. The use of design patterns will help the components of our application be reusable, loosely coupled, and of higher overall quality.

Abstract Factory

For our crawler component, the Abstract Factory creational design pattern could prove tremendously useful. Given the fact that we will implement two crawlers, as mentioned in the requirements analysis, (a CLI one and a GUI one), or even different specific crawlers for each data-source, Abstract Factory is a good fit. Given the fact that the aforementioned crawlers can be tailored to specific uses, but at their core, they are still a web-crawler, they could implement a common interface and each class that inherits said common interface should have its implementation suited to its purpose. For example, we could have a Crawler interface, which would have two subclasses, CLICrawler and GUICrawler. Furthermore, the use of Abstract Factory will prove very helpful if, down the line, for any reason, like finding a new source of data, we decide to extend functionality.

Adapter

For our Normalizer component, whose task is taking the data used and standardizing it into an universal format, one structural design pattern that could prove useful is Adapter. Let's say that the universal format that will be used is a .txt file, but our crawlers, after scraping the web for information, download either wealth, interest, or income declarations. These will, most commonly, be in a .pdf format. The way the Adapter design pattern proposes communication between these two data types is by using an interface, which is compatible with one of the existing object. This will most likely be used for the .pdf documents that will be sourced. Then, our adapter will provide the conversion to .txt by allowing two objects of different data types to "communicate" with each other, courtesy of the aforementioned interface.

Chain of Command

Given our application flow, with each component passing data to one another, sequentially, the Chain of Command behavioral design pattern could prove to be fitting. This design pattern proposes that each component handles its own validation, which, in our case, could mean the following: after data is sourced by the crawlers, the normalizer has to check if documents are in the required format. This should not be done by the crawler, even though it regards data which it has sourced.