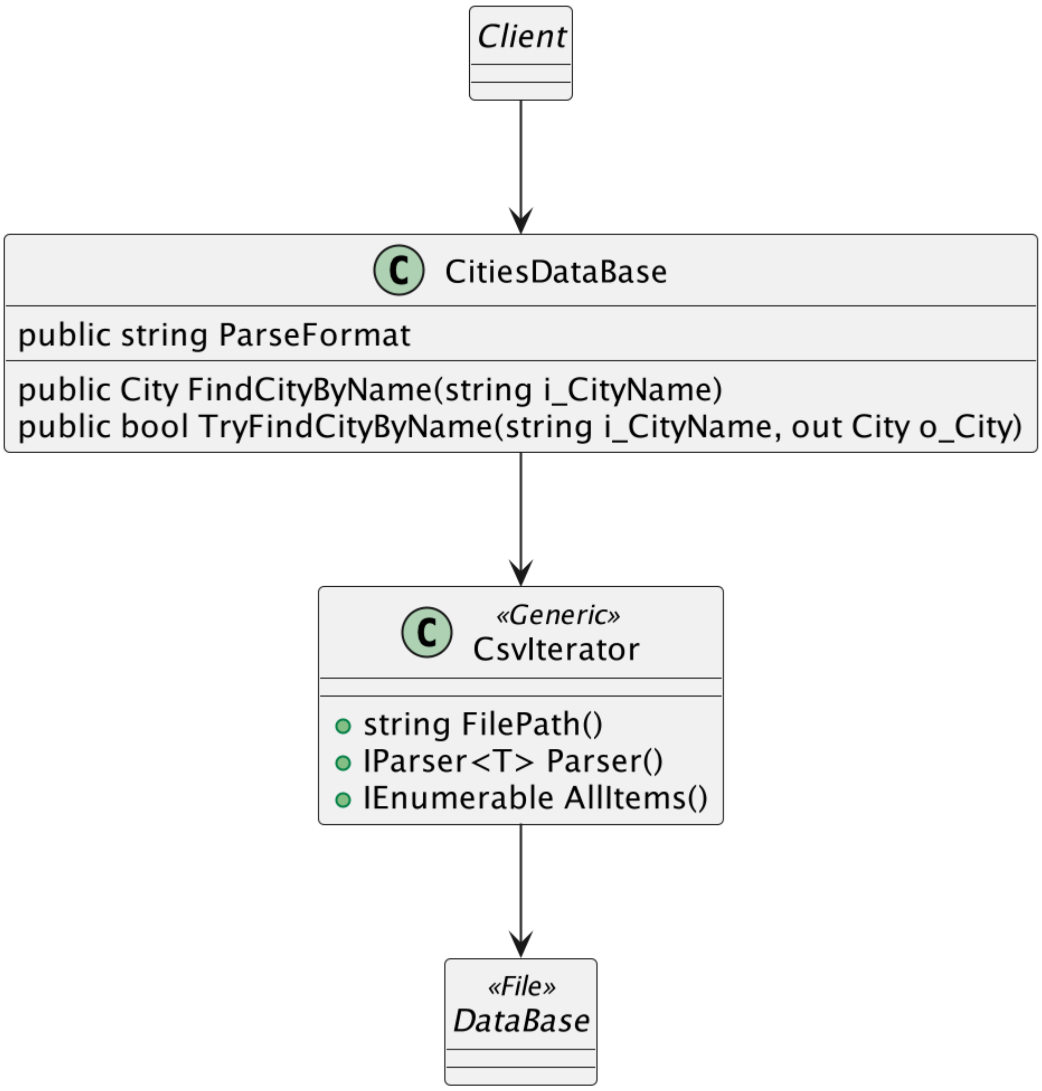
Selected design patterns:

1. Iterator – The iterator design pattern is used in order to promote reusability, extensibility, maintainability and modularity of iterating trough a specific collection. It does so by defining a scanning algorithm rather than supplying the actual data structure.

   In the current project a simple txt file (written in csv format) was used as the cities' data base. As such it is highly recommended to supply an iterator in order to scan the data-base (especially since the only use of it is for searching a specific city).This the scanning logic is aggregated in one component and the city class is decoupled from it. Also, performance is dramatically improved since searching in a large is no longer requiring the creation of all cities. This structure institutes an infrastructure for the second pattern – strategy.



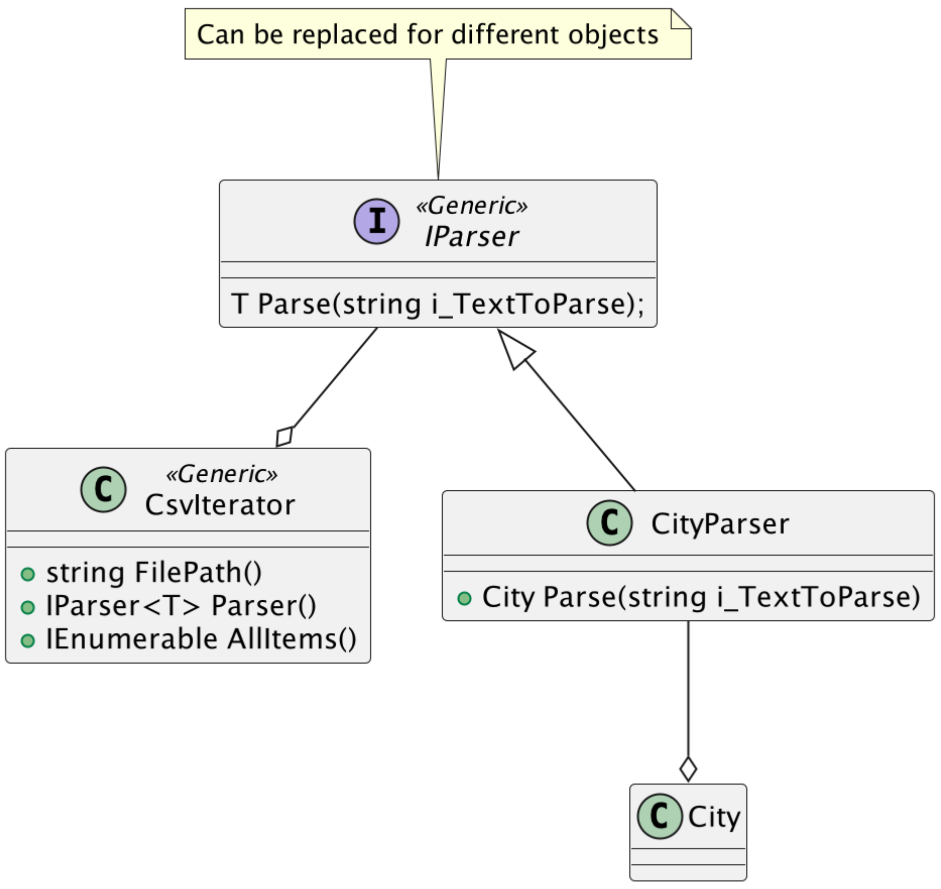
Client
C CitiesDataBase
public string ParseFormat
public City FindCityByName(string i_CityName)
public bool TryFindCityByName(string i_CityName, out City o_City)
«Generic»
C CsvIterator
string FilePath()
IParser<T> Parser()
IEnumerable AllItems()
«File»
DataBase

2. Strategy – The strategy design pattern is used in order to modify a specific selected part in an overall none-modifiable component. As such it allow to reuse a specific logic in different context. The modifiable part is stored in a different class or in a different function pointer (if supported by the language) and can be swapped upon need or even in runtime.

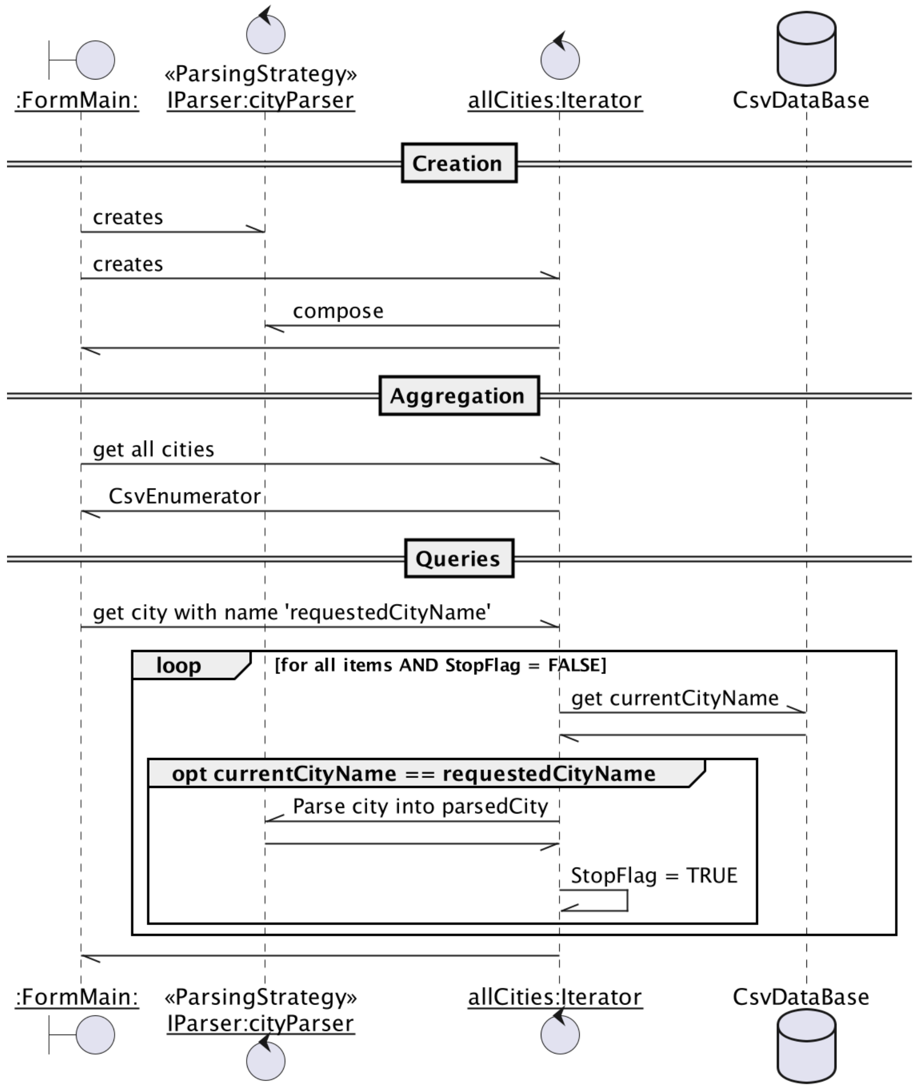
   In the app it was implemented in order to regulate parsing format. As mentioned in the iterator design pattern, the app used a csv format. Thus the parsing logic was implemented to suit those needs. Nonetheless changing the format to a JSON or XML is certainly a feasible situation as the system grow.

   Therefore the parsing mechanism is aggregated in a separate class – CsvParser.

   Nevertheless the entity who uses the CsvParser knows it as an interface IParser rather than a class. This makes the format switching t in the future highly convenient.

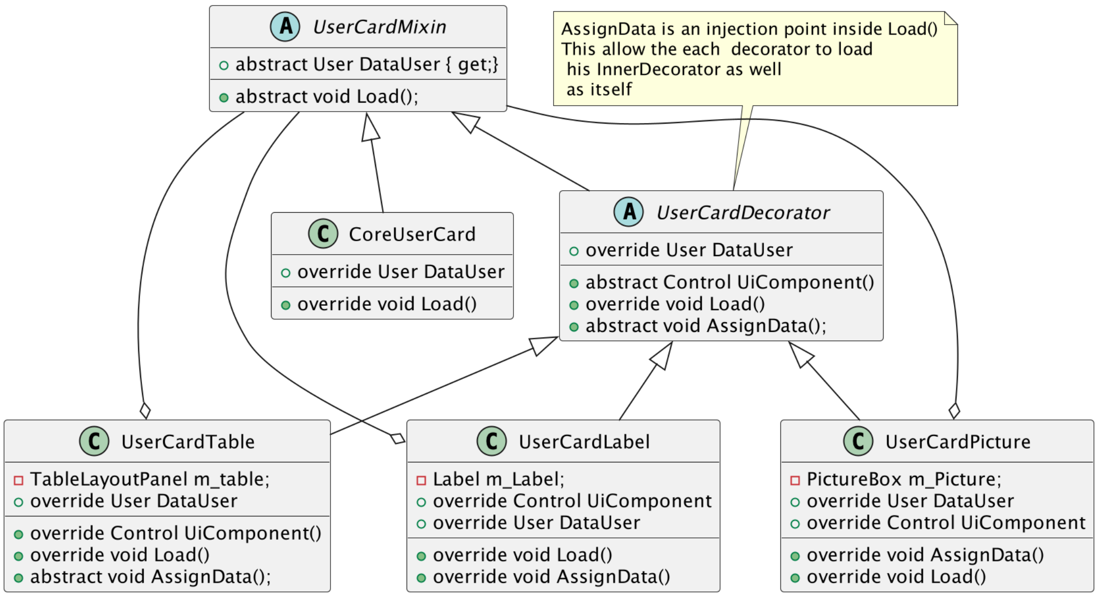


A Sequence diagram that demonstrate the integration between pattern 1 and pattern 2:


:FormMain:
«ParsingStrategy» IParser:cityParser
allCities:Iterator
CsvDataBase
Creation
creates
creates
compose
Aggregation
get all cities
CsvEnumerator
Queries
get city with name 'requestedCityName'
loop
[for all items AND StopFlag = FALSE]
get currentCityName
opt currentCityName == requestedCityName
Parse city into parsedCity
StopFlag = TRUE

3. Decorator – the decorator design pattern is used in order to create different combinations of multiple functionalities in the same polymorphic family .This pattern uses a 'Core Decorator' which is essentially the most basic implementation - no functionality at all or basic functionality. The core decorator can be 'decorated' with other decorators and have different uses upon choice.

   In the current project the pattern used for presenting the user's 'card' i.e. his or her basic information, similarly to an ID card. In this way adding or removing more details in the future is way more flexible.



UserCardMixin
abstract User DataUser { get;}
abstract void Load();
AssignData is an injection point inside Load()
This allow the each decorator to load
his InnerDecorator as well
as itself
CoreUserCard
override User DataUser
override void Load()
UserCardDecorator
override User DataUser
abstract Control UiComponent()
override void Load()
abstract void AssignData();
UserCardTable
TableLayoutPanel m_table;
override User DataUser
override Control UiComponent()
override void Load()
abstract void AssignData();
UserCardLabel
Label m_Label;
override Control UiComponent
override User DataUser
override void Load()
override void AssignData()
UserCardPicture
PictureBox m_Picture;
override User DataUser
override Control UiComponent
override void AssignData()
override void Load()