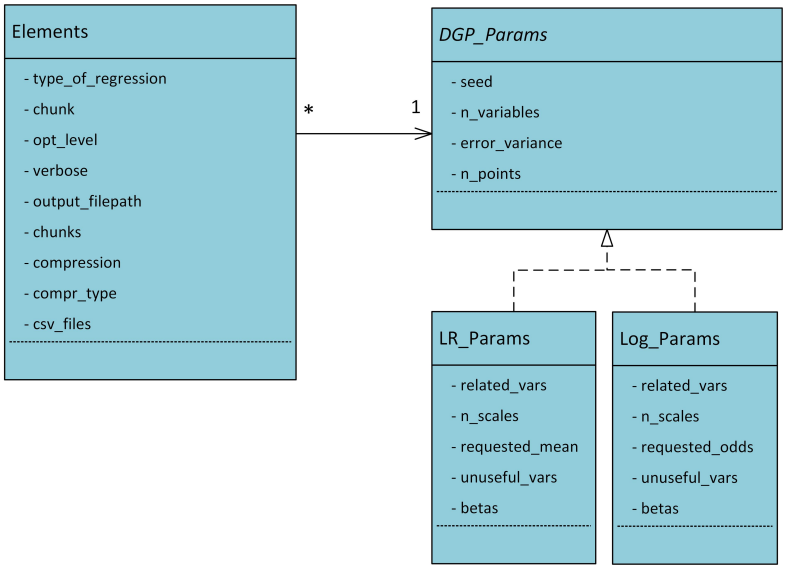
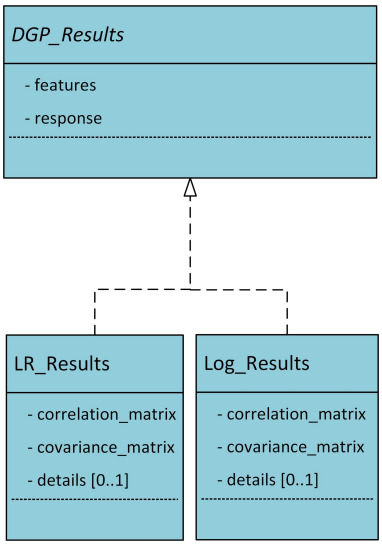
0.1 Domain diagrams



- To represent the different mathematical parameters in the different types of regression we have chosen to construct the abstract class (*DGP_Params*) and have it implemented by the parameters of the generation varieties that we want to achieve.

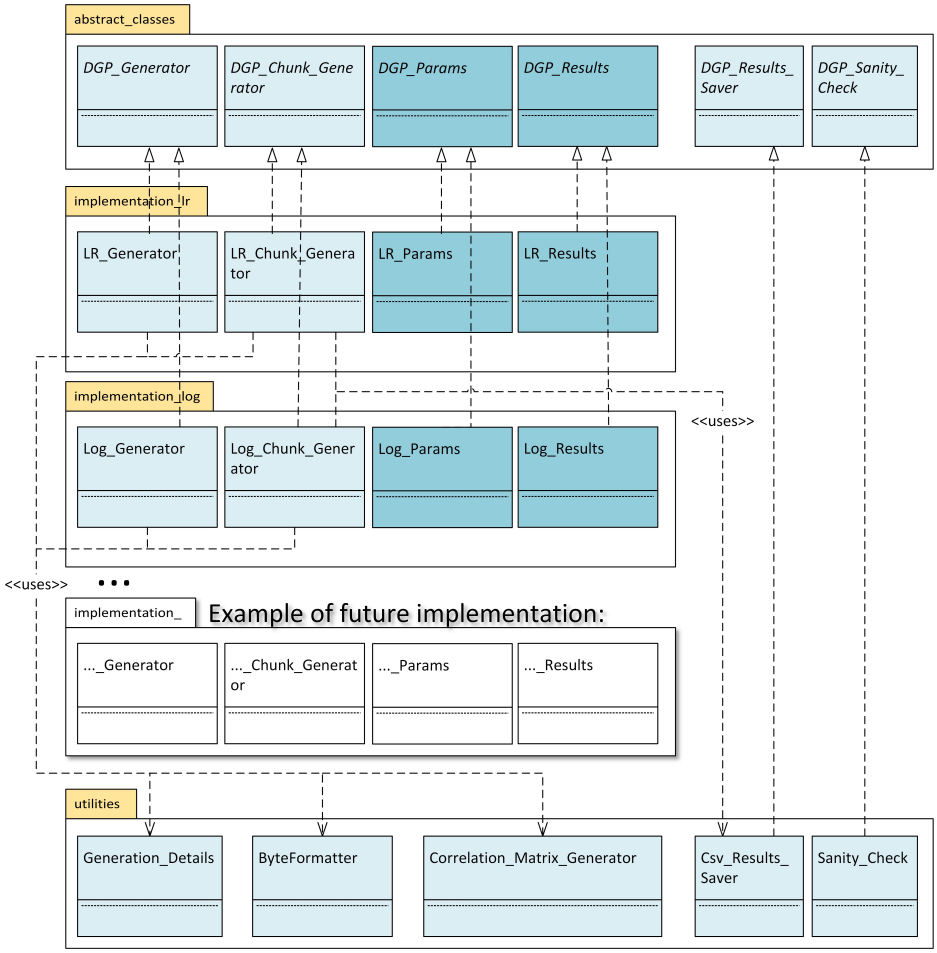
- The relation between the *Elements* and *DGP_Params* classes is "one to many" since each instance of the *Elements* class corresponds to one and only one instance of the *DGP_Params* class, and since each instance of the *DGP_Params* class can belong to an indefinite number of Element instances, even none.

- The abstract classes are distinguished from the others, in this diagram and the following ones, by the title in italics

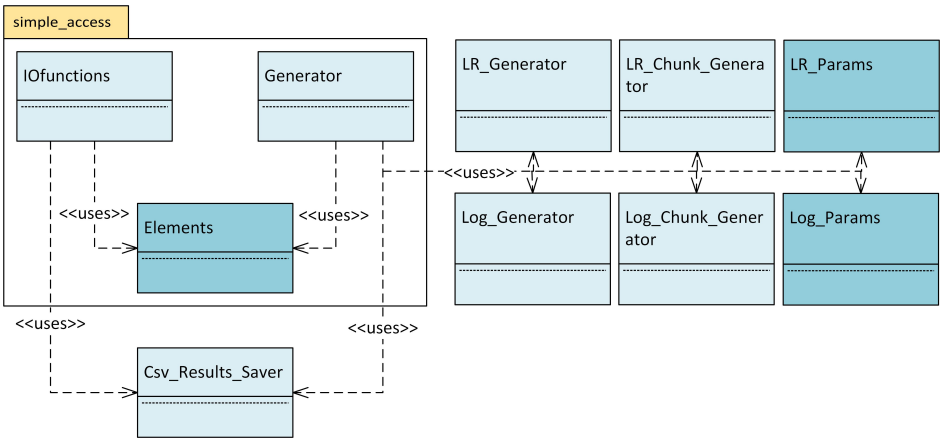


Even if for Linear and Logistic Regression the outputs produced by the generator will be the same (X table, Y series, correlation matrices, covariance matrices and details), it is still good to base the output domain structure on an abstract class, only with the aim of favoring future extensions of the model to types of generations that produce different outputs.

0.2 Packages

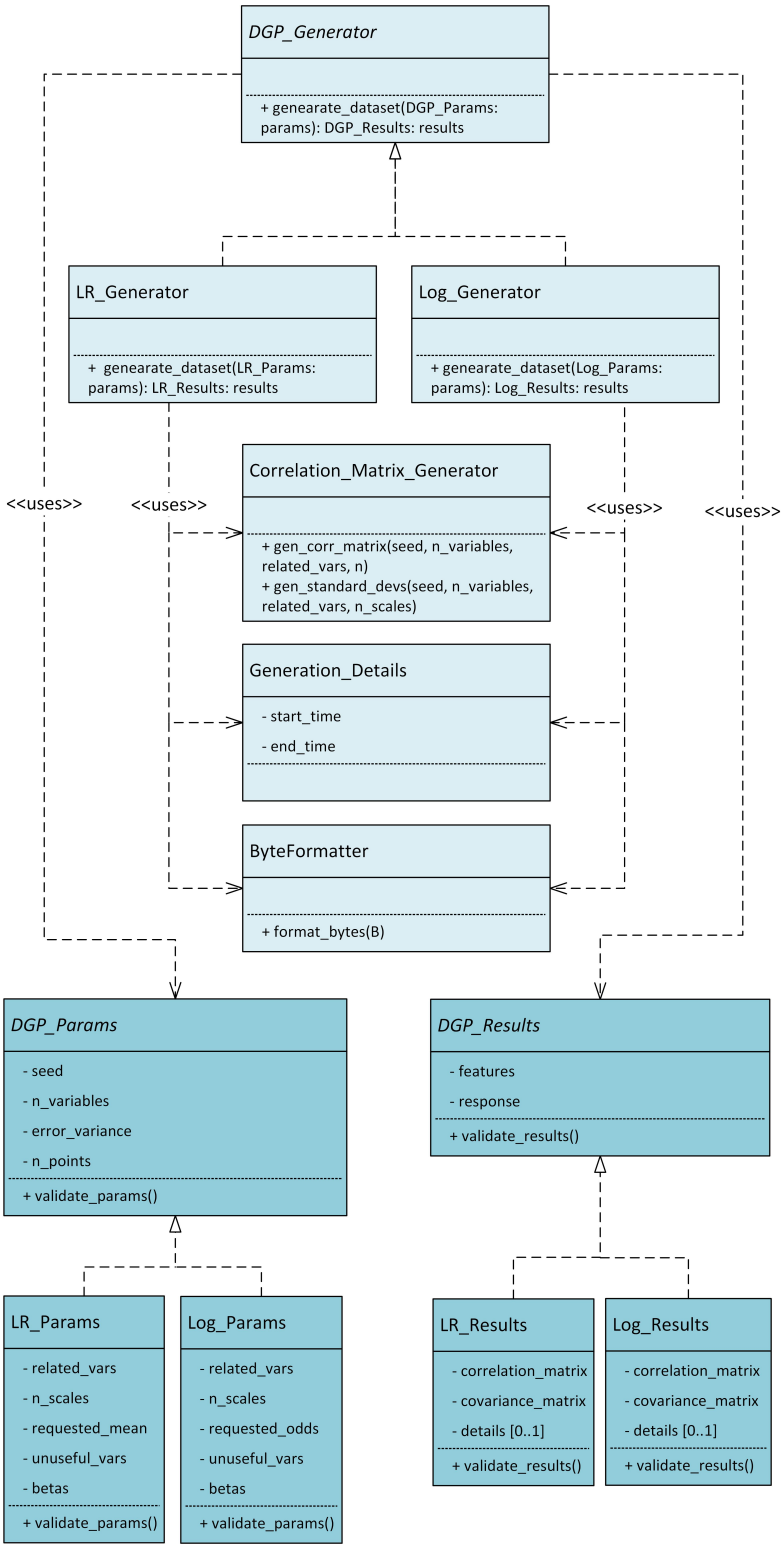


- All the classes that can be useful for the different types of generation will be contained in the *utilities* package. Among these can appear ordinary classes (for example *Correlation_Matrix_Generator*) or implementations of abstract classes (for example *Csv_Results_Saver*).
- The domain classes, in this diagram and in all subsequent ones, are distinguished from the others because they are colored a darker blue.

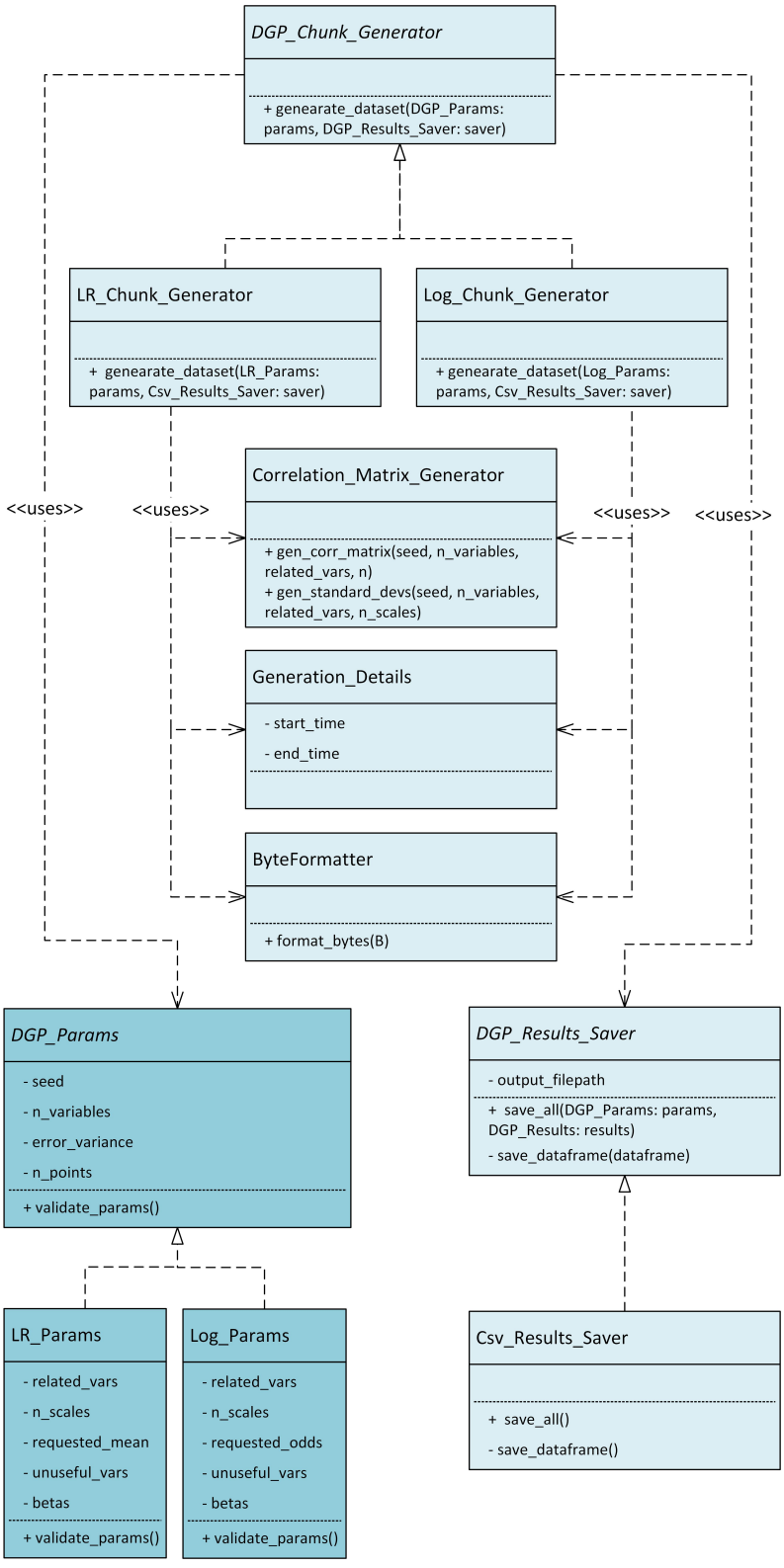


The classes of the *simple_access* package relate, directly or indirectly, to the entire system

0.3 Classes

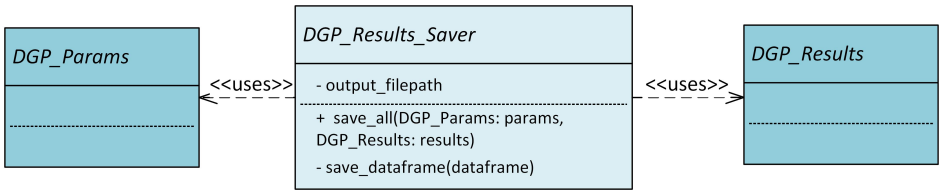

DGP_Generator
+ genearate_dataset(DGP_Params: params): DGP_Results: results
LR_Generator
+ genearate_dataset(LR_Params: params): LR_Results: results
Log_Generator
+ genearate_dataset(Log_Params: params): Log_Results: results
Correlation_Matrix_Generator
+ gen_corr_matrix(seed, n_variables, related_vars, n)
+ gen_standard_devs(seed, n_variables, related_vars, n_scales)
Generation_Details
- start_time
- end_time
ByteFormatter
+ format_bytes(B)
<<uses>>
<<uses>>
<<uses>>
<<uses>>
DGP_Params
- seed
- n_variables
- error_variance
- n_points
+ validate_params()
DGP_Results
- features
- response
+ validate_results()
LR_Params
- related_vars
- n_scales
- requested_mean
- unuseful_vars
- betas
+ validate_params()
Log_Params
- related_vars
- n_scales
- requested_odds
- unuseful_vars
- betas
+ validate_params()
LR_Results
- correlation_matrix
- covariance_matrix
- details [0..1]
+ validate_results()
Log_Results
- correlation_matrix
- covariance_matrix
- details [0..1]
+ validate_results()

- The abstract class *DGP_Generator* represents the core of the software and its implementations will contain all the logic of a specific generation typology.
- The data generations will be carried out on the basis of the user's choices and, above all, starting from an instance of *DGP_Params* which, with its specific implementations, will provide the mathematical parameters
- The generation function will return to an instance of a *DGP_Results* class.
- The two generations delegate certain features to the 3 classes *Correlation_Matrix_Generator*, *Generation_Details* and *ByteFormatter*, however they are found in the *utilities* package, therefore they can also be used by the new types of generation that will develop in the future.
- *Correlation_Matrix_Generator* is capable of generating matrices that randomly mix correlated and uncorrelated variables.


DGP_Chunk_Generator
+ genearate_dataset(DGP_Params: params, DGP_Results_Saver: saver)
LR_Chunk_Generator
+ genearate_dataset(LR_Params: params, Csv_Results_Saver: saver)
Log_Chunk_Generator
+ genearate_dataset(Log_Params: params, Csv_Results_Saver: saver)
Correlation_Matrix_Generator
+ gen_corr_matrix(seed, n_variables, related_vars, n)
+ gen_standard_devs(seed, n_variables, related_vars, n_scales)
Generation_Details
- start_time
- end_time
ByteFormatter
+ format_bytes(B)
<<uses>>
<<uses>>
<<uses>>
<<uses>>
DGP_Params
- seed
- n_variables
- error_variance
- n_points
+ validate_params()
DGP_Results_Saver
- output_filepath
+ save_all(DGP_Params: params, DGP_Results: results)
- save_dataframe(dataframe)
LR_Params
- related_vars
- n_scales
- requested_mean
- unuseful_vars
- betas
+ validate_params()
Log_Params
- related_vars
- n_scales
- requested_odds
- unuseful_vars
- betas
+ validate_params()
Csv_Results_Saver
+ save_all()
- save_dataframe()

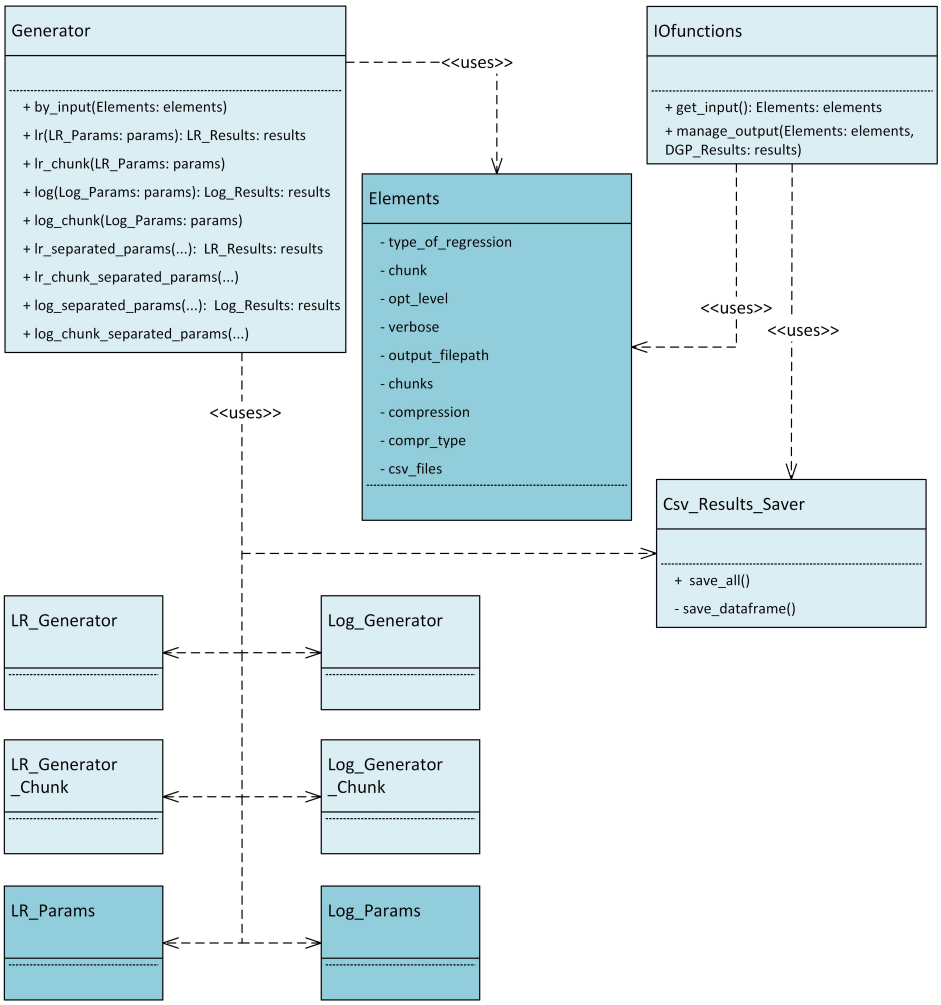
- The consideration of *DGP_Generator* for the generation in portions and *DGP_Chunk_Generator*, this type of generation does not require the use of classes in which to save the results, but requires the 'use of *DGP_Results_Saver*.

- The only class implementing *DGP_Results_Saver* present at the moment is *Csv_Results_Saver*, however the very existence of the abstract class *Csv_Results_Saver* opens the door to the implementation of new methods of saving risultati.



- This diagram analyzes the classes involved in saving results.
- The *Csv_Results_Saver* class is related to the *DGP_Params* class as it has among its functions the printing of beta parameters on .csv
- The *Csv_Results_Saver* class is related to the *DGP_Results* class as it has among its functions the printing on .csv of all the attributes present in *DGP_Results*. Obviously, this functionality is associated only with the standard generation, as the generation in portions does not produce instances of *DGP_Results*

0.3 Classes


Generator
+ by_input(Elements: elements)
+ lr(LR_Params: params): LR_Results: results
+ lr_chunk(LR_Params: params)
+ log(Log_Params: params): Log_Results: results
+ log_chunk(Log_Params: params)
+ lr_separated_params(...): LR_Results: results
+ lr_chunk_separated_params(...)
+ log_separated_params(...): Log_Results: results
+ log_chunk_separated_params(...)
<<uses>>
IOfunctions
+ get_input(): Elements: elements
+ manage_output(Elements: elements, DGP_Results: results)
Elements
- type_of_regression
- chunk
- opt_level
- verbose
- output_filepath
- chunks
- compression
- compr_type
- csv_files
<<uses>>
<<uses>>
<<uses>>
Csv_Results_Saver
+ save_all()
- save_dataframe()
LR_Generator
Log_Generator
LR_Generator _Chunk
Log_Generator _Chunk
LR_Params
Log_Params

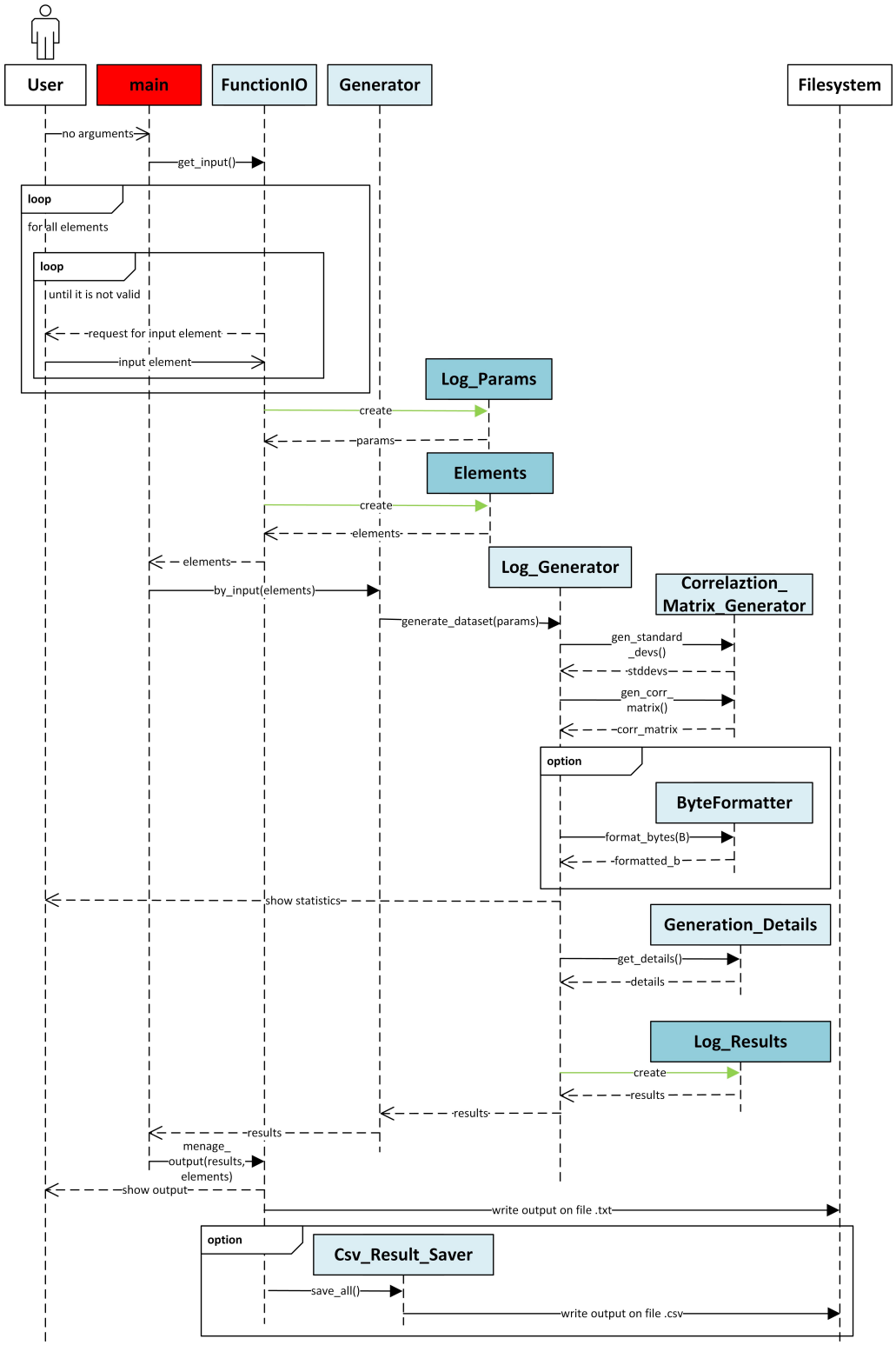
- The *IOfunctions* class implements the functions of input from file and terminal and management of the generation results (presentation on terminal, saving on .txt file and possibly on .csv file)

- The *Generator* class provides numerous and simple generator access functions:

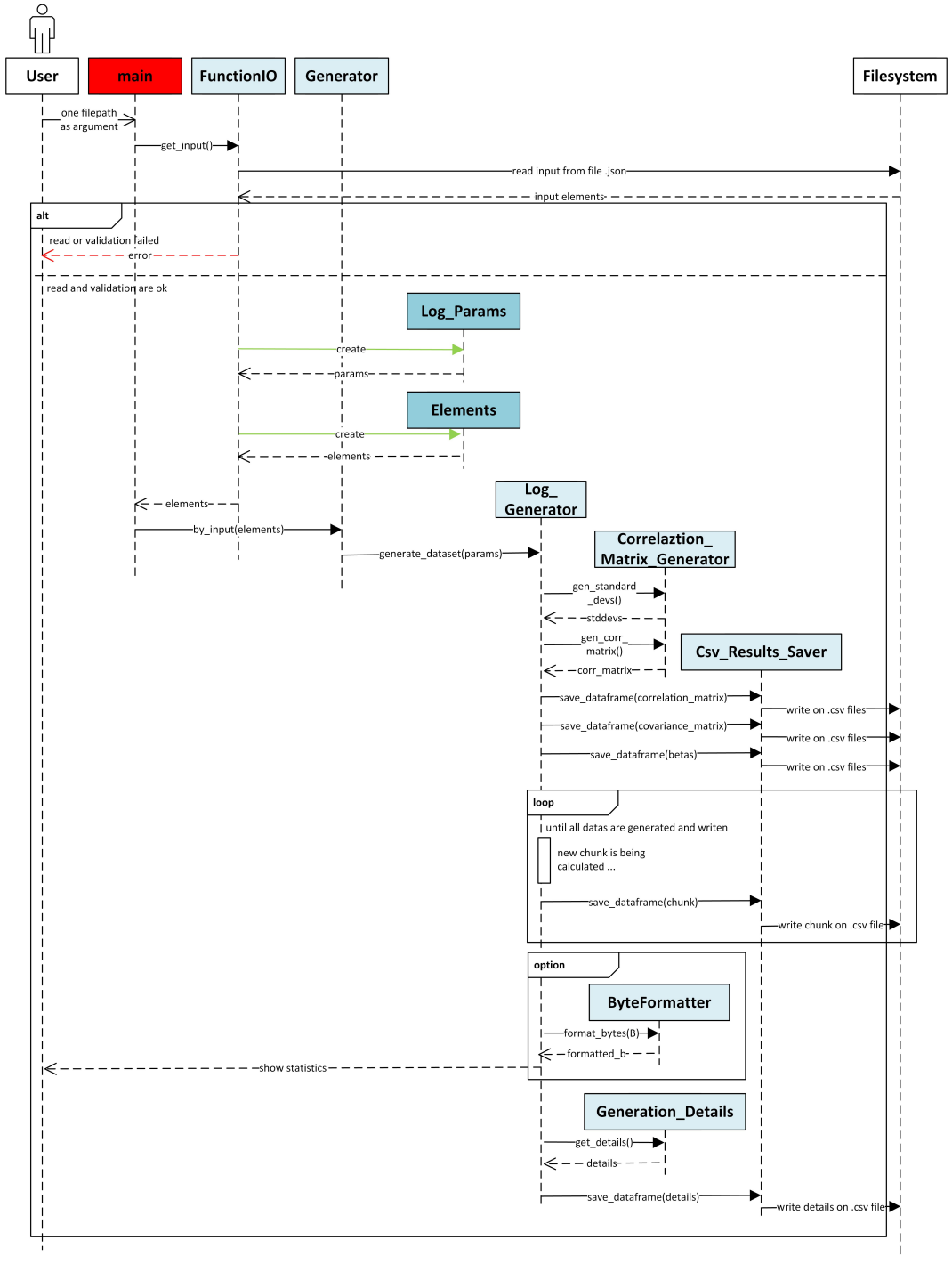
  - the *by_input* function starts generation from the mathematical parameters and choices contained in an instance of *Elements*. Such an instance will typically be obtained from the *get_input* function of *IOfunctions*. item the functions ending with *_separated_params* are the most intuitive of all, in fact they require in input the single mathematical parameters and the generation choices (which can be omitted, as there is default for each). item the remaining functions require as input a class *DGP_Params* already instantiated and possibly the generation choices.

0.4 Sequence

- The first diagram describes the input of terminal input and the generation of data starting from Logistic Regression.
- The second diagram describes the loading of input from .json files and the generation of data by chunk on the basis of Logistic Regression.


User
main
FunctionIO
Generator
Filesystem
no arguments
get_input()
loop
for all elements
loop
until it is not valid
request for input element
input element
Log_Params
create
params
Elements
create
elements
elements
Log_Generator
Correlaztion_
Matrix_Generator
by_input(elements)
generate_dataset(params)
gen_standard
_devs()
stddevs
gen_corr_
matrix()
corr_matrix
option
ByteFormatter
format_bytes(B)
formatted_b
show statistics
Generation_Details
get_details()
details
Log_Results
create
results
results
results
menage_
output(results,
elements)
show output
write output on file .txt
option
Csv_Result_Saver
save_all()
write output on file .csv

0.4 Sequence


User
main
FunctionIO
Generator
Filesystem
one filepath as argument
get_input()
read input from file .json
input elements
alt
read or validation failed
error
read and validation are ok
Log_Params
create
params
Elements
create
elements
elements
by_input(elements)
Log_ Generator
generate_dataset(params)
Correlaztion_ Matrix_Generator
gen_standard _devs()
stddevs
gen_corr_ matrix()
corr_matrix
Csv_Results_Saver
save_dataframe(correlation_matrix)
write on .csv files
save_dataframe(covariance_matrix)
write on .csv files
save_dataframe(betas)
write on .csv files
loop
until all datas are generated and writen
new chunk is being calculated ...
save_dataframe(chunk)
write chunk on .csv file
option
ByteFormatter
format_bytes(B)
formatted_b
show statistics
Generation_Details
get_details()
details
save_dataframe(details)
write details on .csv file

- The diagrams presented can easily be translated from a Linear-based generation situation to a Logistic-based situation, in fact all you have to do is replace the creation of class instances of the domain of a regression type with the opposite regression type. *For example*:
  Log_Params will become *LR_Params*.
  Log_Results will become *LR_Results*.