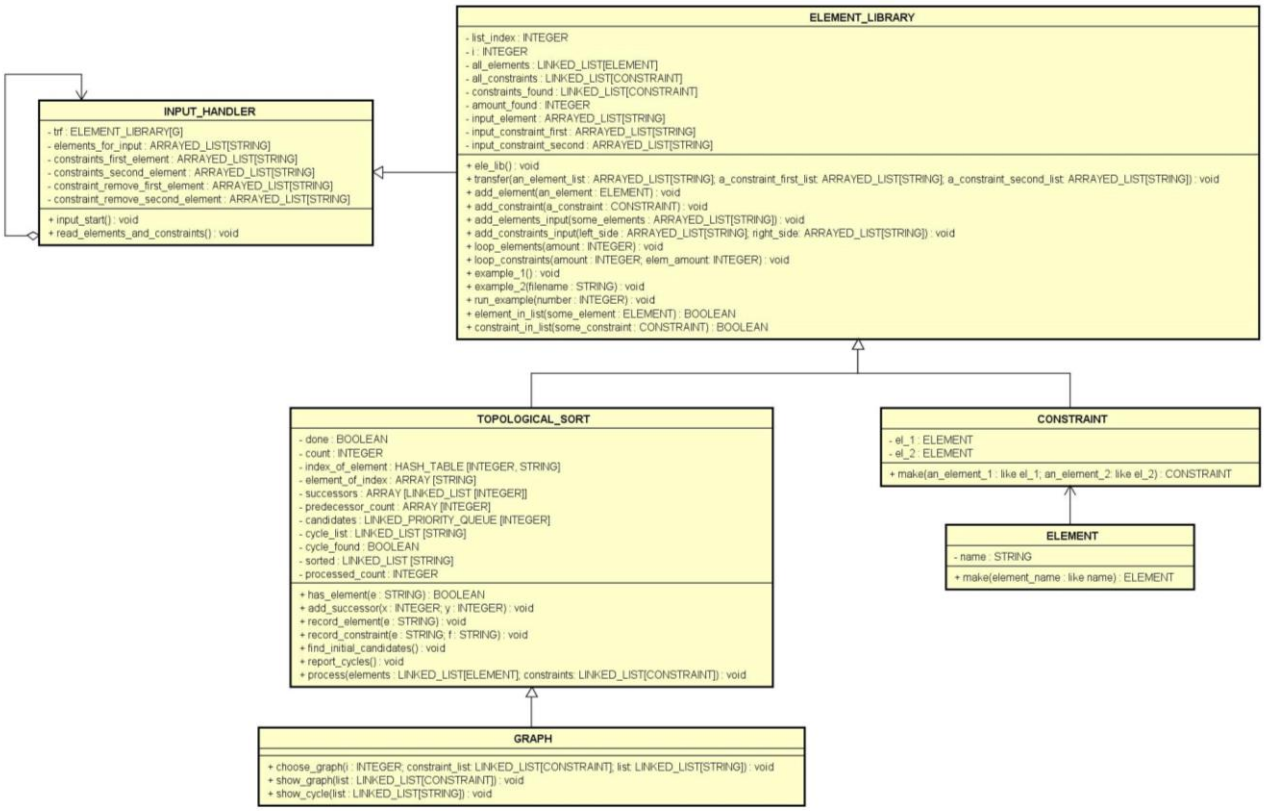
Design Document

Description of our design pattern in use - Chain of Responsibility

For the design of this program we used the *Chain of Responsibility* as our design pattern, because the client request is getting send down the class structure chaining the classes together. We have a single processing pipeline starting at the *Input_Handler* class as our client, which is sending the request and/or inputs down to the *Element_Library* class, that decides, if it can fulfil the request itself or has to send the request further down to *Topological_Sort* and *Constraint* classes. For a request to make a graphical representation of the order of sorted elements the *Topological_Sort* class has to pass down the request to the *Graph* class. A more detailed use of classes in the *Chain of Responsibility* design pattern is as followed:

- The *Input_Handler* class acts as a client. The *read_elements_and_constraints* routine asks for user inputs and, depending of the user request, sends the user requests and inputs down to the *Element_Library* class by using its *transfer* routine. The request for one of the five examples is made by calling the *run_example* routine of the *Element_Library* class.
- The *Element_Library* takes the requests and inputs from the *Input_Handler* class and, depending of the request, makes a request itself by calling one of the routines. If the request is for a topological sort, the routines *add_element*, *add_constraint*, *add_elements_input* and *add_constraints_input* turn the user input into a form suitable for the routines of the *Topological_Sort* class to use and the *transfer* routine sends a request to the *Topological_Sort* class by calling the *process* routine of the *Topological_Sort* class. The request for the execution of one of the examples is received from the *Element_Library* class through the access of the *run_example* routine. Depending of user request, *run_example* calls *example_1*, *example_2,* or one of the three examples inside the *run_example* routine, which themselves transfer the request to the *Topological_Sort* class by calling the *process* routine.
- The *Topological_Sort* class receives the user request with inputs from different routines of class *Element_Library* calling the *process* routine of the *Topological_Sort* class. The *process* routine applies the topological sort algorithm on the given input by executing various routines inside the *Topological_Sort* class and then sends a request to the *Graph* class using the routine *choose_graph,* belonging to the *Graph* class.
- When a graphical representation of the results of the topological sort is needed, the *Graph* class receives the request and runs the routines *show_graph* or *show_cycle*, depending on the request given. The results of the both routines are the main outputs of the program.
- The *Constraint* class receives a request every time an object of type Constraint has to be created. Mostly the routines *add_constraints_input*, *loop_constraints*, *example_1*, *example_2*, and *run_example* of the *Element_Library* class create constraints. To create a constraint objects of class *Element* are needed, so the class *Constraint* makes an indirect request to the class *Element.*
- The *Element* class receives a request every time an object of type *Constraint* has to be created. Mostly the routines *add_elements_input*, *loop_elements*, *example_1*, *example_2,* and *run_example* of the *Element_Library* class create objects of class *Element*.


INPUT_HANDLER
- trf : ELEMENT_LIBRARY[G]
- elements_for_input : ARRAYED_LIST[STRING]
- constraints_first_element : ARRAYED_LIST[STRING]
- constraints_second_element : ARRAYED_LIST[STRING]
- constraint_remove_first_element : ARRAYED_LIST[STRING]
- constraint_remove_second_element : ARRAYED_LIST[STRING]
+ input_start() : void
+ read_elements_and_constraints() : void
ELEMENT_LIBRARY
- list_index : INTEGER
- i : INTEGER
- all_elements : LINKED_LIST[ELEMENT]
- all_constraints : LINKED_LIST[CONSTRAINT]
- constraints_found : LINKED_LIST[CONSTRAINT]
- amount_found : INTEGER
- input_element : ARRAYED_LIST[STRING]
- input_constraint_first : ARRAYED_LIST[STRING]
- input_constraint_second : ARRAYED_LIST[STRING]
+ ele_lib() : void
+ transfer(an_element_list : ARRAYED_LIST[STRING]; a_constraint_first_list: ARRAYED_LIST[STRING]; a_constraint_second_list: ARRAYED_LIST[STRING]) : void
+ add_element(an_element : ELEMENT) : void
+ add_constraint(a_constraint : CONSTRAINT) : void
+ add_elements_input(some_elements : ARRAYED_LIST[STRING]) : void
+ add_constraints_input(left_side : ARRAYED_LIST[STRING]; right_side: ARRAYED_LIST[STRING]) : void
+ loop_elements(amount : INTEGER) : void
+ loop_constraints(amount : INTEGER; elem_amount: INTEGER) : void
+ example_1() : void
+ example_2(filename : STRING) : void
+ run_example(number : INTEGER) : void
+ element_in_list(some_element : ELEMENT) : BOOLEAN
+ constraint_in_list(some_constraint : CONSTRAINT) : BOOLEAN
TOPOLOGICAL_SORT
- done : BOOLEAN
- count : INTEGER
- index_of_element : HASH_TABLE [INTEGER, STRING]
- element_of_index : ARRAY [STRING]
- successors : ARRAY [LINKED_LIST [INTEGER]]
- predecessor_count : ARRAY [INTEGER]
- candidates : LINKED_PRIORITY_QUEUE [INTEGER]
- cycle_list : LINKED_LIST [STRING]
- cycle_found : BOOLEAN
- sorted : LINKED_LIST [STRING]
- processed_count : INTEGER
+ has_element(e : STRING) : BOOLEAN
+ add_successor(x : INTEGER; y : INTEGER) : void
+ record_element(e : STRING) : void
+ record_constraint(e : STRING; f : STRING) : void
+ find_initial_candidates() : void
+ report_cycles() : void
+ process(elements : LINKED_LIST[ELEMENT]; constraints: LINKED_LIST[CONSTRAINT]) : void
CONSTRAINT
- el_1 : ELEMENT
- el_2 : ELEMENT
+ make(an_element_1 : like el_1; an_element_2: like el_2) : CONSTRAINT
ELEMENT
- name : STRING
+ make(element_name : like name) : ELEMENT
GRAPH
+ choose_graph(i : INTEGER; constraint_list: LINKED_LIST[CONSTRAINT]; list: LINKED_LIST[STRING]) : void
+ show_graph(list : LINKED_LIST[CONSTRAINT]) : void
+ show_cycle(list : LINKED_LIST[STRING]) : void