Implementation Notes

As a requirement for the project, the programming language *Eiffel* was used for the implementation of *AutoTasks*. Eiffel is an object-oriented language, which uses the 'Design by Contract' principle. Meaning the contracts, in form of preconditions *require*, postconditions *ensure,* and invariant, make sure that if the input is correct before execution, it must ensure that the output is given out and is conform to the preconditions. Also like in other object-oriented languages, the program code is divided into classes, which for their part contain functions, or *routines*, that are used for execution of different program functions.

AutoTasks is intended as an *Eiffel* library, which implements the topological sort algorithm and makes it easier to use externally. The basic function of the topological sort algorithm is taking tasks or elements, that must be ordered and sort them into the optimal order, regarding the given constraints. *AutoTasks* works similar to the description of the algorithm. The elements and constraints can be entered or removed from the outside, in form of a console prompt. From the console the topological sort can be started as well. When the topological sort is started, the elements and constraints are processed, the elements are ordered according to the constraints, cycles are cut out, a topological sort is produced of the non-cyclical part and the resulting order of elements is given out in form of a visual graph. From the console mentioned above five examples can be accessed as well.

For *AutoTasks* the *Chain of Responsibility Design Pattern* is used. There are three big groups in which the classes can be divided. The first one is responsible for taking and processing the input and the start of the execution of different modes, the second group deals with the build-up of the program elements and building blocks, and the third group consists of the actual topological sort algorithm class. The groups are interconnected by routine calls to the other groups: the input group to the building group, and the building group to the topological sort class.

The *Input_Handler* class is responsible for the console input interface, the reading out, processing and transfer of inputs to another class. When executed, the routine in the class invokes a console, where the user chooses between modes (add element, add constraint, remove element, remove constraint, execute topological sort, execute example 1, example 2, example 3, example 4, example 5 and finish execution) and enters the inputs. The addition and removing happens in the three local string arrayed lists (for the element input, for the first subelement of constraint and for the second subelement of constraint). When the topological sort mode is invoked, the three string arrayed lists get transferred through a routine call to the *Element_Library* class.

The *Element_Library* class has multiple duties. The main function of the class is to turn the given input lists into a form that the *Topological_Sort* class can use. By using the routines in class, the given string arrayed lists are transformed into a linked list of elements of the self- defined class *Element* and a linked list of constraints of the self-defined class *Constraint.* Further *Element_Library* acts as a link between the classes *Input_Handler* and *Topological_Sort* by receiving the inputs from the former and invoking a routine from the latter, with the transformed inputs as attributes. The *Element_Library* is also responsible for the invocation of the five examples. When needed, the routines for examples create the example specific input for topological sort and invoke the topological sort routine with the specific input as attribute.

The *Topological_Sort* class contains the routines needed for the execution of the actual topological sort algorithm. The class gets the inputs, in form of a linked list of elements and a linked list of constraints, from a routine being called inside the *Element_Library* class. Both linked lists get processed and transferred to another type of list, which can be used by the algorithm. Then all elements without a predecessor are found. Depending on the constraints the successors for these elements are found and the elements with found successors are transferred into the sorted list and a

cycle is reported, if it exists. A routine call to a routine in the *Graph* class commits the sorted list, and the cycle information to the *Graph* class routines.

The graphs of the ordered elements and possible cycles are created by the routines in the *Graph* class. The routines get a string list of elements, that has to be turned into a graph, as attribute and returns a .dot-file with the code for *Graphviz* as output. *Graphviz* is a third-party graphical software that creates the actual graphical representation of the sorted elements.

There were many difficulties, that were encountered, while implementing this project. The functionality of some classes overlapped and it was not very clear in point of what function belongs to which class. Much time went into making different parts, like the topological sort, the looping of constraints for the random number generator and the connections between classes, work. Some parts had to be abandoned for time reasons, like the input of the constraint in a special form or additional functions. The program is still quite error prone: some examples are not working and wrong inputs lead to error messages. The design pattern had to be changed from *Iterator* to *Chain of Responsibility,* because the structure ‘linked_list’ in *Eiffel* combines all features of the *Iterator* pattern and was more useful as a tool rather than our actual pattern. There are many improvements, which could be made on the program. The first two goals would be to make the program function completely and make the console more user-friendly.