Contents

Revision History

Date	Version	Description	Author(s)
26.09.18	0.1	First discussion and writing	Stalder, Khandozhko, Jurilj
27.09.18	0.2	Writing in Chapter 1 and 2	Stalder, Khandozhko, Jurilj
03.10.18	0.3	Discussion and partially Requirements	Stalder, Khandozhko, Jurilj
07.10.18	0.4	Discussion and writing in all parts	Stalder, Khandozhko, Jurilj
09.10.18	0.5	Writing and fixes in all parts	Stalder, Khandozhko, Jurilj
11.10.18	0.6	Fixes in all parts	Stalder, Khandozhko, Jurilj
12.10.18	0.7	Fixes in all parts	Stalder, Khandozhko, Jurilj
13.10.18	0.8	Formatting and fixes	Stalder, Khandozhko, Jurilj
14.10.18	1.0	Final changes and formatting	Stalder, Khandozhko, Jurilj

1 Introduction

1.1 Purpose

This document represents the Software Requirements Specification (SRS) for the Eiffel library *AutoTasks*. It is designed and written for the stakeholders, which include library developers (namely David Stalder, Yuliya Khandozhko, Dominik Jurilj), the Course Instructor, Course and Teaching Assistants, Tutors, other students, and any other potential users of the library. Its purpose is to ensure the product performs as intended by describing both functional and non-functional requirements, constraints and properties without restricting them by giving implementation details.

1.2 Scope

The *AutoTasks* project aims to create an extensible library of classes to optimize the ordering of objects or tasks of daily life. The main focus lies on tasks (further called elements), which have certain constraints to their arrangement, e.g. if one element is dependant on the termination of another element, the former must not be executed before the latter. The basic version of the *AutoTasks* library shall allow any user to add tasks and constraints, and display a graphical representation as a graph. Additionally, the program shall be able to produce a topological sort if the case allows it, and lastly, it shall recognize cycles, document them, if existent, and produce a topological sort of the non-cyclical part.

A more advanced version of the library might include graphical representation of cycles, user interface for easier addition/removal of elements or constraints, or any other additions to help lessen the complexity of usage.

To ensure the proper functioning of the library at least 5 examples have to be given, in addition to the library itself. The first example shall work with the topological sort page of Rosetta Code. For the second example a make-file with dependency information between elements shall be used and return the correct order as a result. The last three examples shall test whether the program runs properly for higher number of elements and constraints.

1.3 Definitions, Acronyms and Abbreviations

Term	**Explanation**
Element	A unit (task or object), which has to be placed in the right order.
Constraint	Prevents that a particular element can be placed before or after another one and is of the form <element 1, element 2>. In other words: “Element 1 must come before element 2”.
Graph	A graphical representation of a specific *AutoTasks* instance.
Topological sort	Algorithm that structures a given partial order into a compatible total order.
Group of elements	All elements of the current instance.
AutoTasks	The Eiffel library that contains all functions of this project.
ID (for functional requirements)	A unique sequence of letters and numbers used to unambiguously identify a requirement.
Priority (for functional requirements)	From 1 as highest to 4 as lowest; the functions of higher priority have to be implemented before the lower ones.
User	The potential person, who gives input in the form of elements and constraints.
GUI	Acronym for “Graphical User Interface”.

1.4 References

IEEE 830-1998 (IEEE Standards Organization, https://standards.ieee.org/findstds/standard/830-1998.html)

Eiffel Programming Language (https://www.eiffel.org/)

Lecture Slides “Software Construction” (Bertrand Meyer, Universität Zürich, 2018)

Examples for SRS provided by the course.

Bertrand Meyer: “Touch of Class: Learning to Program Well with Object and Contract”, Springer-Verlag, 2009

Topological Sort (Rosetta Code, https://rosettacode.org/wiki/Topological_sort, 09.10.2018)

1.5 Overview

The remaining chapters are dedicated to explaining what *AutoTasks* contains:

Chapter 2 delivers the foundation of understanding of what the library should do and comprise, what it might do with possible extensions, and what the main constraints for the (basic) library are.

Chapter 3 has more precise information, such as the functional requirements with descriptions and priority values, performance requirements, and information about maintainability and design constraints.

Chapter 4 contains supporting information, namely a short explanation about topological sorting and two example program runs.

2 Overall Description

2.1 Product Perspective

AutoTasks is an extensible library developed in the Eiffel programming language. As such the library depends on a program for the implementation and execution of its elements. The program could be an Eiffel compiler or an IDE. In this case EiffelStudio IDE is the preferred choice.

To simplify the usage of the library (i.e. for user without knowledge of Eiffel) a GUI could be a useful addition to the system.

Product Functions

The following functions are to be used for *AutoTasks*' basic library:

- Creating an element.
- Creating a constraint between two elements.
- Removing an element.
- Removing a constraint.
- Displaying a graphical representation of the given elements and constraints.
- Perform a topological sort, if applicable.
- In case of a cycle, document the cycle and perform a topological sort of the non-cyclical part.
- In case of a cycle, display it graphically.
- GUI where the user can create and remove an element or a constraint.

Functions for examples:

- Functions for looping elements and constraints into example input files for testing the performance with varying amounts of input.
- A function for each example.

For a more precise and detailed description of the functions, see section 3.2.

2.2 User Characteristics

AutoTasks can be used as a means to structure the day or optimize the order of tasks and objects by anyone with access to the library. Since the project is on the educational side, the users are mostly restricted to the Course Instructor, Course and Teaching Assistants, Tutors, potentially other students, and the developers.

2.3 Constraints

- The library is written in Eiffel Programming language and can be used in an Eiffel environment only.
- A list of predefined elements and constraints, from which the user can choose, is not possible. By defining them beforehand, the user could easily implement e.g. a simple calendar or a todo list (by giving each task an amount of time it takes to fulfill).
- The created group of elements and constraints cannot be saved for later use. The elements and constraints only exist when the program is running.

2.4 Assumptions and dependencies

The library should work on most newer generation computers and the user is expected to use a newer version of EiffelStudio to prevent system errors. The user should understand the basics of the Eiffel programming language to ensure a correct usage.

3 Specific Requirements

This chapter focuses on requirements, constraints and quality aspects of *AutoTasks* and the system.

3.1 External Interfaces

The interface used for the product is the Eiffel Programming environment. The environment used in this project is EiffelStudio.

3.2 Functional Requirements

3.2.1 Requirement Explanation

ID	A unique identification sequence to clearly distinguish the requirements.
Title	Name of the requirement function.
Description	A more detailed explanation of the requirement function. In case of ambiguity, if the description does not mention the implementation method, the developer may choose how to address it (e.g. choosing which form of topological sort to use).
Priority	Defines the importance and the order in which the requirements should be implemented. The priority goes from 1 as the highest to 4 as the lowest. The requirements of higher priority should be implemented prior to the ones of lower priority.

Following are the functions to be used in the library. The priorities range from 1 to 4, where 1 is highest priority and an obligatory requirement, and 4 is the lowest priority, only to be implemented after successfully finishing lower priority ones.

Once the topological sort has been conducted, no further elements/constraints can either be added (cf. 3.2.2.2/ 3.2.2.3) or removed (cf. 3.2.2.4/ 3.2.2.5).

To ensure the proper functioning of the library at least 5 examples have to be given, in addition to the library itself. For the first example the inputs of the topological sort page of Rosetta Code shall be applied to the program and the received results shall be compared to the results given on Rosetta Code. The second example consists of reading a simple make-file. It shall contain the information on dependence between elements as input for the program. The result shall be a line of elements, whose ordering is based on the input. A cyclic structure should also be possible to process in such manner.

The last three examples should have a list of ordering restrictions as input. Each line of the list shall contain a number of elements, the order of which shall be maintained for the final element sorting. The first example shall have about 10 constraints, the second about 1’000 constraints and the third about 100’000 constraints. Each example shall contain about 4, 200 and 2000 elements per line respectively. For the program to be successful the execution time shall roughly be equal the expected execution time. The expected execution time is roughly t for the first example, $100*t$ for the second one and $10’000*t$ for the third one.

3.2.2 Functions for the Main Program

ID	3.2.2.1
Title	application
Description	Contains the main program with functions and the access commands for different libraries including *AutoTasks.*
Priority	1

ID	3.2.2.2
Title	add_element
Description	Creates and adds a new element to a group of elements. Once the topological sort (cf. 3.2.2.7) has been conducted, no further element can be added.
Priority	1

ID	3.2.2.3
Title	add_constraint
Description	Adds a new constraint between two different elements, provided the two elements already exist in the group of elements, where the first element has to occur before the second. Once the topological sort (cf. 3.2.2.7) has been conducted, no further constraint can be added.
Priority	1

ID	3.2.2.4
Title	remove_element
Description	Removes the respective element from the group of elements, provided it has been inserted before (cf. 3.2.2.2). If there has been a constraint (cf. 3.2.2.3) with the element to be removed, the respective constraint will also be be deleted by calling the function remove_constraint (cf. 3.2.2.5). Once the topological sort (cf. 3.2.2.7) has been conducted, no further element can be removed.
Priority	1

ID	3.2.2.5
Title	remove_constraint
Description	Removes the respective constraint from the group of constraints, provided it has been inserted before (cf. 3.2.2.3). Once the topological sort (cf. 3.2.2.7) has been conducted, no further constraint can be removed.
Priority	1

ID	3.2.2.6
Title	show_graph
Description	Usable after the topological sort (cf. 3.2.2.7). The output of this function is a graph, which shows all elements (cf. 3.2.2.2), ordered in regard to the constraints (cf. 3.2.2.3). If a cycle has been discovered (cf. 3.2.2.8), the graph will be shown without the cycle, i.e. if there is a graph with a cycle in the middle only the part before and/or after the cycle (whatever exists), without the elements belonging to the cycle, is shown.
Priority	1

ID	3.2.2.7
Title	topological_sort
Description	Runs a topological sort through the elements (cf. 3.2.2.2) to create the right order for the elements in consideration of the constraints (cf. 3.2.2.3). The sort should run on a linear time O(N+M), where N is the number of elements and M is the number of constraints. The topological sort is only conducted on the non-cyclical part (cf. 3.2.2.8).
Priority	1

ID	3.2.2.8
Title	check_cycle
Description	Runs a check through the elements (cf. 3.2.2.2) to find any possible cycles regarding the constraints (cf. 3.2.2.3). If there is a cycle, it shall be documented and removed. Has to be used before the topological sort (cf. 3.2.2.7).
Priority	1

ID	3.2.2.9
Title	show_cycle
Description	If there is a cycle (cf. 3.2.2.8), show a graphical representation of it.
Priority	3

ID	3.2.2.10
Title	take_input
Description	A user interface that asks the user for the input of elements (cf. 3.2.2.2) and/or constraints (cf. 3.2.2.3) or asks them for the removal of elements (cf. 3.2.2.4) and/or constraints (cf. 3.2.2.5)
Priority	3

ID	3.2.2.11
Title	graphical_ui
Description	A graphical user interface to facilitate the usage of *AutoTasks*. The UI should include boxes for filling in elements/constraints (cf. 3.2.2.2/ 3.2.2.3) and removing (cf. 3.2.2.4/ 3.2.2.5) them.
Priority	4

ID	3.2.2.12
Title	ex_1_rosetta_code
Description	Example covering the topological sort example given on the page of Rosetta code (cf. Topological sort in chapter 1.4 and 4.1).
Priority	2

ID	3.2.2.13
Title	ex_2_make_file
Description	Example that accepts a simple make file as input. The input is a sequence of lines each of the following form: elem0: elem1 elem2 elem3 … elemN where each element (elemX) represents a simple word and where the ":" means dependence of the former element of the latter elements (elem0 depends on elem1...elemN). One can assume that the input always satisfies this format.
Priority	2

ID	3.2.2.14
Title	ex_3_10_cons_4_els
Description	Example that accepts an input with 4 different elements and 10 constraints. The input is created randomly (cf. 3.2.2.17/ 3.2.2.18). As a result the execution time shall be calculated and returned.
Priority	2

ID	3.2.2.15
Title	ex_4_1000_cons_200_els
Description	Example that accepts an input with 200 different elements and 1000 constraints. The input is created randomly (cf. 3.2.2.17/ 3.2.2.18). As a result the execution time shall be calculated and compared to the execution time, calculated in example 3 (cf. 3.2.2.14).
Priority	2

ID	3.2.2.16
Title	ex_5_100000_cons_2000_els
Description	Example that accepts an input with 2000 different elements and 100000 constraints. The input is created randomly (cf. 3.2.2.17/ 3.2.2.18). As a result the execution time shall be calculated and compared to the execution time, calculated in example 3 (cf. 3.2.2.14).
Priority	2

ID	3.2.2.17
Title	element_loop
Description	Automatized (and randomized) creation of new elements (cf. 3.2.2.2) for the example files (cf. 3.2.2.14/ 3.2.2.15/ 3.2.2.16).
Priority	2

ID	3.2.2.18
Title	constraint_loop
Description	Automatized (and randomized) creation of new constraints (cf. 3.2.2.3) for the example files (cf. 3.2.2.14/ 3.2.2.15/ 3.2.2.16). The output shall be a number of lines with elements. The order of elements (cf. 3.2.2.2) in a line shall be also pertained in the overall ordering.
Priority	2

3.3 Performance Requirements

We assume the use of a newer generation computer (2013+), where the program shall never take more than 5 seconds in normal usage. The examples described in 3.2 might take longer, since it is not a realistic real life situation, especially the examples with looped inputs. Those examples (examples 3, 4, and 5) shall respectively have an execution time of *t*, 100**t*, and 100'000**t*. Lastly we have the running time of the topological sort, which should not exceed running time O(N+M), where N is the number of elements and M is the number of constraints (cf. 3.2.2.7). No endless-loops, stopping the program from terminating, should be encountered.

3.4 Maintainability

3.4.1 Elements and Constraints

The way the elements and constraints are defined (element as one single task, and constraint, that includes only two elements) makes them rather simple to use with every possible function. However, that is also what makes them non-extendable regarding the functions that use them. For example, if one were to use three or more elements in a constraint, the other functions will have troubles to adapt to the changes, if not implemented with an exceptional design. Another way of extending those differentiated elements and constraints would be creating new functions which are compatible with other types.

3.4.2 Graphs

As mentioned in 3.4.1, the elements and constraints are well defined, which allows it to add additional graph types and possibilities to showcase the content of specific *AutoTasks* implementations.

3.4.3 User Interface

The basic library of *AutoTasks* provides only simple text input possibilities for users. A possible graphical user interface (lower priority) could provide additional input functionality and/or new ways to

interact with the library. The GUI functions should be easily extensible by adding more detail to the existing functions, by implementing new ones or even by creating multiple GUI. One of the problems of implementing multiple interfaces might be changing the selection of the UI by the program itself (i.e. which UI shall be used right now?).

3.4.4 Other Functions

By upholding the points mentioned in the previous subchapters, namely holding true to the form of the elements and constraints, there is no clear limit of what can be added to the library.

3.5 Design Constraints

For this program the following design constraints will be considered:

- Example 5 will contain 100’000 constraints and 2000 elements, therefore the maximum number of constraints that can be entered will be 150’000 and the maximum number of elements will be 2500.
- With a large number of elements and constraints, the graph will become too complex, therefore the graph will only be outlined if the number of elements is small enough to be properly displayed.
- The elements and the constraints are functionally well-defined and therefore once implemented it will be nearly impossible to change them (e.g. one constraint can only contain two elements).
- The program cannot be run by an executable file, it must always be opened with Eiffel Studio.
- There can only appear one GUI at once, i.e. there cannot be two or more GUI’s opened at once.
- Should an error occur in the second example, it will not be fixed, but it will be considered as the correct input.
- The program will not perform the process of saving elements and constraints permanently.

4 Supporting Information

4.1 Topological Sorts

There are multiple methods to implement a topological sorting algorithm. They accomplish the same thing in slightly different ways. The choice of method used in this project lies with the developers. Two well-known algorithms include Kahn’s Algorithm and Depth-first search.

For more information on Kahn’s algorithm:

Kahn, Arthur B. (1962), "Topological sorting of large networks", *Communications of the ACM*, **5** (11): 558–562, doi:10.1145/368996.369025.

For more information about the depth-first algorithm:

Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001), "Section 22.4: Topological sort", *Introduction to Algorithms* (2nd ed.), MIT Press and McGraw-Hill, pp. 549–552, ISBN 0-262-03293-7.

4.2 Examples

What follows is an example of a possible run of the program:

- While in *application* (cf. 3.2.2.1), three calls are made to *add_element* (cf. 3.2.2.2), entering different elements through the user interface.

- Using *add_constraint* (cf. 3.2.2.3) twice, and do a *topological_sort* (cf. 3.2.2.7). The three aforementioned elements should (or not) have a certain order of execution.
- We use *check_cycle* (cf. 3.2.2.8) to be certain our constraints will not create a cycle. In case of a cycle appearing, we would be notified.
- Satisfied with having no cycle, we make the program show us a visualization using *show_graph* (cf. 3.2.2.6), since the graph will surely be small enough to be displayed with this tiny sample size.
- The program concludes correctly.

A more complex example (for details of the more important functions, look at the first example in this chapter):

- We add 20 elements.
- We create 30 constraints. We notice that there will be redundant constraints.
- We add another 5 elements and 5 constraints. All of the constraints target at least one of the 5 new elements each.
- After using the *topological_sort,* we use *check_cycle* and notice that there is a cycle in the middle of the order, since one was documented.
- *show_cycle* (cf. 3.2.2.9) visualizes the part with a cycle.
- We still want to look at the remaining elements, so we use *show_graph* to design a graph of the non-cyclic parts (before and after the cycle). Since the number of elements and constraints is small enough, we get a visual representation of the order (without the cycle).
- The program concludes correctly.