University of Zurich UZH

Department of Informatics

Software Construction (HS18)

Instructor:	Prof. Bertrand Meyer
Course Assistants:	Carol Alexandru, Gerald Schermann
Teaching Assistant:	Anna Jancso
Tutors:	Louis Bienz, Philip Hofmann, Sarah Zurmühle, Marc Zwimpfer

In this term, you will carry out a project with the prime goal of putting to practice the main principles and techniques that you learn in the Software Construction course.

Project Description (version 1.1)

Overview

In our daily lives, we perform various tasks, be it at home or at work or elsewhere. Many tasks can be carried out independently of other tasks. For example, it usually does not matter whether we first clean the kitchen or the bathroom. Other tasks, however, must occur in a certain order. For example, you probably do not want to first put on your shoes and then your socks.

The aim of this project is to create an Eiffel library called *AutoTasks* which allows you to specify tasks and ordering constraints between them, and to produce *task orderings* compatible with the constraints. Apart from the library, your delivered software must also include five example programs (with sample input data) using the library to define correct task orderings.

To create these task orderings, you will have to implement the **topological sort**. There are two different underlying algorithms for the topological sort, Kahn's algorithm and depth-first search. You can use whichever you prefer, but only Kahn's algorithm will be studied in the lectures.

Library

The target domain contains "elements" (called "tasks" above, but the scope is more general) and "constraints". A constraint is of the form <e1, e2> for two elements e1 and e2. The meaning of such a constraint, particularly in the case of tasks, is: "e1 must come before e2".

The input to the topological sort consists of the elements and the constraints. The output of the topological sort is a list of the elements in an order that respects the constraints.

The requested library should support, at a minimum, the following functions:

- Enter an element.
- Enter a constraint.
- Remove an element.
- Remove a constraint.
- Given elements and constraints, display a graphical representation as a graph (you can use the freely available graph visualization software Graphviz for that purpose).
- Produce a topological sort if it exists.
- If there is a cycle, document it, and produce a topological sort of the non-cyclical part.

This description of the functions is general and has to be made precise as part of the requirements document (cf. Milestone 1).

Possible bonus functions, subject to the same observation:

- If there is a cycle, display it graphically.
- Provide a textual user interface for the modification operations (such as the first four above, adding/removing elements/constraints).
- Provide a graphical UI for them (hint: consider the EiffelVision library).
- And any cute idea that you may have of fancy things to add (as long as the basics as outlined above are covered)!

Examples

As part of the project, you have to prepare a minimum of five example programs that build an element-constraint set and produce a topological sort using the library. The requirements for the examples are the following:

- One of the programs must cover the example given on the topological sort page of Rosetta Code. The program should satisfy the acceptability criteria of the Rosetta Code site. In other words, you should run your program on the Rosetta input example and check that it gives the exact result requested.
- One of the examples should be a simple *make* file. You only need to support a simplified version of the *make* syntax: the input is a sequence of lines each of the form

  elem_0: elem_1 elem_2…elem_n

  where $elem_0$, $elem_1$, $elem_2$,... are simple words (sequences of characters not involving a space or new line — in the actual *make*, they are usually file names). Such a line describes that $elem_0$ depends on $elem_1$, $elem_2$,... The output should be a list of all the elements appearing in the input file, in an order such that, if there is a line such as the above, elem_1, elem_2,... appear before $elem_0$. In analyzing such a *make* file, you may assume that it satisfies this format (i.e. you do not need to do error processing on the format), but you should be able to process an input that describes a cyclic structure (and in that case report that there is a cycle).

- For the next three examples, choose any convention that you find convenient to enter elements and constraints. You don't actually need to list the elements. A very simple possibility is to have a sequence of lines each consisting of two integers, e.g.

  ```
  25 34
  ```

  describing the constraint that *25* must come before *34*. Then, the elements are simply all integers that appear in any of the constraints. But feel free to use any other convention. You may even reuse the *make*-like convention of the previous example. Just make sure that all three examples use the same convention.

  The first example should have about 10 constraints, the next one about 1000 constraints, and the last one about 100,000 constraints. The respective number of elements can be about 4 in the first example (more realistic than 2 for 10 constraints), the second example 200 and the third example 2000. Your implementation of the topological sort should have a linear running time, i.e.

  O (N + M)

  where *N* denotes the number of elements and *M* the number of constraints. You should give the execution times: if the time is *t* for the first example (even with 4 constraints rather than 2), it should be roughly 100 * *t* for the second one and 10,000 * *t* for the third one. To measure execution time, you may use the class *TIME* from the EiffelTime Library. As the 1000-constraints and 100,000-constraints examples are too tedious to prepare by hand, you may generate the values automatically using a random number generator such as the class *RANDOM* from the *EiffelBase Library* (also have a look at this article).

Criteria

All code (library and examples) must follow good design principles as introduced in class, make appropriate use of contracts, and satisfy the Eiffel style guidelines, which you can find for example at https://www.eiffel.org/doc/eiffel/Style_Guidelines.

Initial Setup

The project is carried out in groups of three students. You must register for a group on **OLAT** ("Group Registration") until **Sunday, 23.09.2018, 23:59**.

We will use Git for version control. All submissions (documents and source code) will be delivered through your repository that will be created for you on GitHub. It will be a private

repository within our GitHub organization and will be available only for the respective team members under the name *sc-hs18-groupX* where *X* is your group number.
To add you to your group's GitHub repository, please provide your assigned tutor with your GitHub username **in the first lab session**.

Grading and Project Structure

The project will be graded and contributes 50% of the final course grade.

Please note: you need to have at least a 4.0 in **both** the project and the exam to pass the course.

To help you plan your workload during the semester, we have adopted a project structure with 3 milestones:

- Software Requirements Specification (SRS)
- API Design and Test Plan
- Implementation

In addition, there will be a project presentation (one short talk for each group) at the end of the semester.

Each phase has a strict deadline and deliverables that need to be handed in (in other words: no extensions will be granted). Additionally, you will have to give two presentations, one about your requirements document and the final project presentation.

Milestones and deadlines

Please be aware that some of the details may be subject to change as the course and project progress. In any such case you will be alerted to the change through OLAT email.

1 Software Requirements Specification (SRS) [30 points]

Duration:	Monday, 24.09.18 – Sunday, 14.10.2018 (3 weeks)
Deliverable:	SRS document (10-20 pages)
Presentation:	in the lab sessions (15./16./17.10.18) [individual time for each group]
Template:	OLAT
Submission:	sc-hs18-groupX/documents/requirements.pdf [SRS document]

The purpose of a requirements specification (SRS) is to describe what qualities a system must exhibit and what goals it must reach. It does not describe how these qualities and goals are achieved. Your task is to develop an SRS document[1] for your project, which will clarify its scope and features and define priorities for the subsequent implementation. When you write your SRS, make sure the document adheres to the 15 quality goals discussed in the lecture. The document has to describe both the library and the example programs.

The presentation should give an overview of your SRS and explain the most important points. If there were any issues during the SRS writing phase that created discussion among the team members, or any specific difficulties you encountered, they could make an interesting presentation topic. Note that presentation time will be strictly limited. Although all project members should preferably be at the presentation, if only to help with questions, you are free to organize it as you see fit, e.g. all members presenting or just one -- which may be the better solution since time will be strictly limited (the exact limit will be specified later but it will be no more than 10 minutes, typically with 5 slides or so).

2 API Design and Test Plan [20 points]

Duration:	Monday, 15.10.2018 – Sunday, 11.11.2018 (4 weeks)
Deliverables:	Design document, Eiffel classes for public API, Test plan
No Presentation	
Template:	OLAT
Submission:	sc-hs18-groupX/documents/design.pdf [Design document]
	sc-hs18-groupX/src [Public API classes]
	sc-hs18-groupX/tests/ [Test suite]

API Design:
Write a set of Eiffel classes that will be the public API of your library. The set of features has to satisfy the requirements described in the SRS. The Eiffel classes have to be equipped with class and feature comments. The features don't have to be implemented yet but should be specified (including contracts when applicable). In the design document, add a class diagram of the library using UML and describe which design patterns you have used. For each design patterns, specify which classes are part of the pattern and why you used this particular pattern.

Test Plan:
Write a test plan for your project. This means that you will take the role of an external test engineering team. Write tests for each functional requirement of the SRS. Annotate each test with the requirement that it exercises. In addition, write tests for each public API routine. Again, annotate each test with the routine under test.

[1] https://ieeexplore.ieee.org/document/720574

3 Implementation [40 points + 10 points (Presentation)]

Duration:	Monday, 12.11.2018 – Sunday, 16.12.2018 (5 weeks)
Deliverables:	Implementation of library, examples and tests and if necessary revised design document and test plan, implementation notes
Presentation:	Last lecture week [individual date/time for each group]
Submission:	sc-hs18-groupX/src/ [Implemented classes] sc-hs18-groupX/documents/design.pdf [Revised design document] sc-hs18-groupX/tests/ [Revised test suite]

Use the prioritized functional requirements to decide what functionalities should be implemented first. Make sure that your code satisfies both the functional and the nonfunctional requirements. If you have to change the design or API, revise the design document and add a chapter where you explain what changes were made in the design and why.

Given the API developed in the previous step and the test plan you have devised, write automated tests for the project. Run the test suite that you have created in the test plan phase. If there were changes in the API, you might have to adapt your tests.

Examples of what we expect in the implementation notes: document all bugs that you have uncovered using the issue tracker of your GitHub project; document and justify any change to the API; document important implementation choices; document problems encountered; etc.

In the final presentation, you should describe how you worked on the project, how you distributed the work and handled communication in the group. You can also show a demo or code snippets to show the usability of your library. As a conclusion of the course, give your personal impression of the project: are you happy with the result? Would you do some part differently now?

General rules

When you work in a group of students, we expect that everyone does his or her fair share. Not necessarily equal contributions (which would be defined to measure anyway), but making sure that the result includes significant work by all group members.
If a problem arises in this respect, it's OK to bring it up explicitly; this is better than accumulating frustration. If you have the impression that one of your teammates is under-contributing, the procedure is:

- Discuss it within the group. The teaching team will not consider complaints unless there has been such an internal discussion first.
- If possible, resolve the question within the group and take any necessary corrective actions. This resolves the matter.

- If not, bring it up with your tutor. The teaching team will examine the matter, getting information from all those concerned.
- Bring up the problem **early**. Experience shows that such problems generally can be resolved amicably if one does not wait until the last minute.

This project is focused, meaning relatively small in scope (based on just one algorithm, and that algorithm was introduced in class!), favoring depth rather than breadth. Quality is of the essence. We hope that you will do a great job, will learn a lot in the process, and (not to be forgotten) enjoy the experience.