Documentation

This document is for the final project of the course Programming 3: Interfaces and Techniques, which I completed in 2022. This is normally a project work for 3 students, but as explained below, most of the important work was done by me. While the project is far from polished, and I didn't have time to finish every feature we planned for, I still feel its a decent show of skills, and was definitely a learning experience.

The project was about creating a study planner that imports the degree structures of study programmes in Tampere University from the Sisu API. The learning outcomes were to practice object-oriented programming, using external libraries and API:s, implementing a GUI and working on a program as a team.

Structure

The core of the program is the Database, that holds and delivers all the necessary information for the rest of the program. The degree structures are modelled closely to the Sisu API implementation, and these are used to represent whole programmes, study modules and individual courses. KoriAPIGetter is responsible for the importing of the degree programmes and initializing the Database. The structures and classes within the Database follow Sisu's type hierarchy for describing degree structures. Module is an abstract supertype for DegreeProgramme, StudyModule and GroupingModule. CourseUnit is a separate type, that are found under GroupingModules and StudyModules. The result is a treelike module hierarchy. Student class contains information about the student, and it is linked to Database, and Sisu is the UI class of the program, which gets all needed information through Database.

Functionality of the program and extra features

Functionality of the program starts with initializing the database with KoriAPIGetter. At the start of the program, it adds all degree programmes to the database. These are displayed on the second scene. After selecting a degree program KoriAPIGetter imports the contents of the degreeprogramme to the database, which gets shown on the third scene.
Sisu Handles all the user made actions and builds the UI and changes it accordingly to the user's actions. Sisu contains some functions to help with user made actions. All information Sisu needs is in Database.
Database contains information of modules and courses.
Classes Student, DegreeProgramme, StudyModule, GroupingModule and CourseUnit make it possible to get information of different types of entities that are in the database. Module is a superclass for DegreeProgramme, StudyModule, GroupingModule.

Extra features:

• Program is comprised of three separate scenes, which can be changed from and to.
This means student data can be updated and it is tracked in the program live.
• When clicked, modules and courses in the third scene treeView open an info table with extra information. The item name in the info table can be hovered to display its full name as a tooltip. Extra information includes min/max credits and descriptions provided by the course.
• Used css to style the program
• IdMode in the third scene: The course names can be changed to the course GroupId and back in the TreeView by activating IdMode and pressing an item name. IdModeButton also has a tooltip explaining its relevance.
• Student can mark a course as done, and see the credits accumulated so far as they press more courses done while in the degree programme view.

Classes containing pre and post conditions

As we understand it, only class that has some conditions is KoriApiGetter since it needs the exact webpage to work. Importing all the information is dependent on the url working which opens the information from the API.

The agreed and actual division of work

The agreed division of work was going to be roughly as follows:
Everyone chips in in creating the importing of the API, as that was essentially the starting point where everything else was built upon. A desired number of 3 from the project was thought to be the goal. After the import we thought we'd dynamically delegate jobs depending on what we want. Initially no one had strong interest in the graphical design, so Pyry agreed to take the brunt of that, while the others would focus on the backend more.
The actual division of work was roughly as follows:
Pyry ended up mostly doing the importing himself. The first live Mayday celebrations in three years took its toll on some of the team more than others, and until almost the end of May, most commits to the API getter, the UI and the database were done by one person. After this Karri got along and started helping as well, but Jiri never really pulled it through, with only some help with the class initializing. The biggest additions by Karri would be the creating and utilizing the Student.java, several elements and functionalities in Sisu.java and most of the initializing of the classes used (both in the classes and KoriAPIGetter.java). Pyry did most of Sisu.java, the styling of it, most of KoriAPIGetter, the database, the logging shenanigans explained later below and various other functionalities.

Short instructions for using the program:

The functioning of the program has been divided into three scenes.
In the first scene, which opens when starting the program, you can input the student's name and student number. Student name needs to be either in form NAME_LASTNAME with one space between, or NAME with no whitespace. Student number must contain only numbers, letters and "-" -symbols, and must be between 5-12 characters long. The default examples in the fields intentionally don't let the user proceed. User can continue to the next scene with the proceed button, when student has a name and student number is a viable six long digit. The first scene also has an exit button which quits the program.
In the next scene which opens after proceeding from the student information scene, you can choose the desired degree-program. The program can be searched from the search field. To choose the degree program user must double click the program or press the select button after clicking once on a program in the table. Users can also go back and change their student information with the back button or quit the program with the exit button.
In the third scene the degree-program and its contents are displayed on the left in a TreeView. Users can click on these to show more information about the module/course on the right and switch the name of the clicked element to show its id. When clicked another time it switches back to display the course name. Course name is also displayed in the header of the info table, and it can be hovered to see the full name. User can also go back from this scene to change the chosen degree program with back button and exit the program with the exit button.

Known bugs, afterthoughts, and missing features:

A lot was left to be desired with the UI of the program. There was simply not enough time to get creative with it, as everything else took most of the time for us. If I would do this project again, I would put much more emphasis on the usability and visual appearance of the UI. There is also a lot of code that doesn't have adequate commenting. The comments are also irregular between different team members, which was never unified during the project. This leads to some of the importing and data visualisation to be quite hard to understand.

Resizing was generally a functionality we didn’t have enough manhours to explore. While 2/3 of the scenes are resizable, it generally adds little to no value to the program, as the Views are not resizing with the scene.

Info in the infoTable (Sisu.java) is not always shown perfectly sometimes a little part of the text is not readable, some adjustments could have fixed that.

Some of the degree programmes contain an insane number of modules and courses, and so going through them recursively will cause a stackoverflowerror. This wasn’t fixed because a) recursion is inherent from the API, and creating an import system that uses for-loops acting as pseudo-recursors to avoid this error would have been time-intensive, b) changing the language stack size sounded scary and “hardware-malfunction-tier-bad" and c) so far we found only one program where this happens which is “Kielten kandidaattiohjelma.” Others might exist as well.

If the program was coded without custom classes, but instead straight up utilised the imported JSONObjects from the API, we think we could have ended up with around 30% less code. The reason we chose to use our own classes was because the project assignment specifically stated one of the learning goals for the project is to practice Object-oriented programming, so we interpreted that as a possible minus should we choose to only work with JSON strings, even if they would have been handy in a lot of places. In terms of learning the systems, this was better overall.

In long and large degree programmes that do load into the TreeView, there are other problems though. When a very long (For example “Automaatiotekniikan DI-ohjelma") programme is shown in the TreeView, and the user scrolls all the way down, if a submodule is minimized while the bottom empty white is shown in the TreeView, a message “INFO: index exceeds maxCellCount.” is thrown to the console by javafx.scene.control.skin.VirtualFlows addTrailingCells function. This is caused because in short, the function thinks we run out of space to add cells, even if they were 1px high in the TreeView. This does not crash the program, but we were not able to get rid of it either, because we weren’t able to access the logging library used by the imported VirtualFlow class. Things that we tried to overcome this problem were, in short:
1. Learn the basics of log4j and log4j2 logging systems and utilize a logging .xml file that would disable logging for a) the class b) any of its superclasses c) INFO-level logs in general. It remains slightly unclear could have this worked if we had more time learning the systems, or, as we suspected was the case, was this doomed to fail because the class used a logging solution we had no access to.
2. Tried to implement a custom TreeViewSkin and VirtualFlow class for our project we could use to override the function in place to remove the message in unnecessary situations. We imported around 3500 lines of source code, tried to learn how to make sense of the 2 classes, and were able to create custom classes that didn’t give compiling errors (not included in the final version but happy to share if the reviewer wants to see/give insight into them). Unfortunately, the plan didn’t work because our custom TreeViewSkins VirtualFlow kept on initialising as null, and we didn’t have enough hours and understanding of the source code to fix this. After 10 total hours on this, the plan was sadly scrapped.
3. We found a workaround that could have disabled ALL logging for good, but we didn’t think that was a feasible solution to a relatively minor issue like this. This would have obviously solved the problem, but potentially created a lot of new ones.