Università
della
Svizzera
italiana

**Software
Institute**

BUILDING A PLATFORM FOR THE MANAGEMENT AND EVALUATION OF COMPUTER SCIENCE CURRICULA

Jason Naldi

September 2021

Supervised by
Matthias Hauswirth

Co-Supervised by
Michele Lanza

SOFTWARE & DATA ENGINEERING MASTER THESIS

Abstract

Periodically, groups like ACM and IEEE publish documents that provide guidelines to help readers understand the evolving landscape of computer science. These guidelines are designed specifically to guide those professors who design computer science study programs for universities. These guidelines however are not enough. Universities that offer computer science study programs have to keep their study programs up to date and they have to evaluate said study programs. Furthermore, there is no concrete connection between guidelines and study programs.

In this thesis, we present a novel platform designed to help universities manage study programs, courses, and instructors. This platform binds study programs and courses to an underlying guideline, makes it possible to contextualize courses and study programs within said guideline, and allows users to explore the guideline itself.

Dedicated to my family, my advisors, and
whoever is reading this thesis.

Acknowledgements

Thank you to my advisors professor Matthias Hauswirth and professor Michele Lanza for the guidance, thank you to Csaba Nagy for helping me set up the GitLab CI pipeline, and thank you to my family for being supportive during the creation of this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Every couple of years, ACM and IEEE get together to produce a document that describes the field of computer science. The results of this joint effort are then published. These documents are targeted towards those professors who are tasked with the creation and/or revision of computer science study programs (also called "curricula"). These documents are meant to be used as guidelines, points of reference when creating computer science study programs. The most recent of these guidelines is the Computer Science Curricula 2013 [10]: in total, it consists of 518 pages and it presents 2′222 topics and outcomes that can be part of courses of a computer science study program. The 2′222 topics and outcomes are grouped into 163 knowledge units, which are in turn grouped into 18 knowledge areas.

The guidelines published by ACM and IEEE show not only that the field of computer science is complex, but that it is also constantly changing, and it is expected to keep changing for the foreseeable future [10, p. 21]. Topics grow and shrink in importance, and new discoveries and break-throughs keep the field evolving.

Within this intricate and evolving landscape, universities create study programs (also called "curricula") that are later offered to their students. Universities face three challenges that are presented in the following sections: Section 1.1, Section 1.2, and Section 1.3.

1.1 Guidelines and a Missing Link

When creating or revising study programs, universities are faced with the task of choosing which contents should be part of the study program.

There exist documents (later called "Guidelines") that are designed to provide recommendations and be a point of reference for those who design study programs. Metaphorically, these documents act as maps that help study program designers navigate the field of computer science.

For the field of computer science, the guideline most frequently cited among the literature we explored is published by ACM and IEEE, in any of its many revisions. In this thesis, we focus on the most recent edition of the ACM and IEEE guidelines: CS2013 [10]. This guideline is organised as a structured collection of information describing suggested contents for various computer science study program types. An explanation of this document is presented in Chapter 2.

We identified three problems pertaining to this document. First, this document is offered only in a textual, PDF format. Second, it is dense with information. A professor that intends to rely on CS2013 needs to sift through the entire document in order to find the topics and outcomes that might be relevant for his or her courses. The third problem is the following: CS2013 offers a granular view of computer science, down to individual topics and outcomes. This granularity is, however, lost outside the document. There is no indication within CS2013 as to how the presented classification of computer science in individual topics and outcomes might translate into actual study programs. This problem is caused, in part, by the fact that this document is provided as a textual PDF.

1.2 Managing Study Programs

Changes to a study program vary from slight adjustments to the topics covered in a course, to a complete overhaul of an entire study program, replacing most, if not all, of its old courses with new ones, and anything in between.

Among the literature we explored, there is no standardized way to represent study programs and, likewise, there is no specific way to approach the task of applying changes to a study program. To tackle these issues, various universities have created in-house solutions. Some of these solutions consist of processes to deal with changes in a study program, whereas other solutions consist of actual platforms that can be used to manage changes in study programs. A more detailed overview of each approach found is presented in Chapter 2.

1.3 Evaluating Study Programs

Revisions to a study program can result in an overall change in the topics covered by said study program. For example, assume a university is revising their study program focused on programming languages: if half of the programming courses of this study program are replaced with cybersecurity courses, this study program will no longer be fully focused on programming languages. Therefore, universities have a need for ways to measure how well their study program fits a given field, or even discover which fields their study programs best fit in.

In Chapter 2 we present various metrics we found in the literature to measure and classify study programs.

1.4 Contributions

Change can bring forth a set of problems that universities must deal with. First, when universities need to revise their study programs, depending on how these study programs are represented, the task might be cumbersome and/or unstructured. Second: after a study program has changed, it is possible that the new revision is no longer equivalent to the previous, so there is a need for metrics that allow to quantify and compare study programs. Third: relying on existing guidelines can be difficult, because they are presented as organized text documents, and there is no accessible way to bind guidelines to study programs.

To tackle the problems illustrated in this chapter, with this thesis we present a novel platform for the management of university study programs that:

- Allows users to create new study programs and revise existing ones.
- Implements a visual contextualization of study programs within the CS2013 guideline.
- Allows users to explore said guideline itself.

Furthermore, in this thesis we present:

- An overview of existing literature on the subjects of managing and evaluating study programs.
- A novel model we used to develop the platform that binds study programs and guidelines together.
- Methods we used to develop and evaluate the platform.
- Inputs on how to further extend said platform.

1.5 Structure of This Thesis

In Chapter 2 we present an overview of the literature we found pertaining to the problems presented in the current section.

In Chapter 3 we present the platform we developed to address the problems identified in this section, including its user interface, its codebase, the underlying model, and other aspects related to the platform.

In Chapter 4 we provide an overview of the development of the platform and our approach to incrementally evaluate it.

In Chapter 5 we discuss all the contributions of this thesis, which include the platform, its development, and its evaluation.

In Chapter 6 we draw conclusions and we list various ideas on how to further develop and improve the platform we presented in Chapter 3.

Chapter 2

Related Work

In this chapter we present an overview of literature we found that is focuses on the problems of study program management, study program evaluation, and the exploration of guidelines.

We split out overview of literature into three groups based on the topics each document treated: guidelines, management of study programs, evaluation of study programs, and exploration of guidelines. This grouping does not define a partition: for example, the paper by Ajanovski treats both study program management and study program evaluation [1].

2.1 Guidelines

Guidelines are documents created with the specific intent of providing a point of reference to study program designers when designing study programs. In the literature, these documents are referred to as "bodies of knowledge". We opted for the name "guideline" because we found it to be more brief and descriptive.

Overall, the most common type of guideline we found in the literature is any of the revisions produced by a joint effort between ACM and IEEE. This guideline was first published by ACM in the 1960s [10] and it is still being revised every few years. The current revision of this guideline was published in 2013 [10], and, at the time of writing, a 2020 revision is being developed [5], [4].

2.1.1 CS2013

This thesis focuses specifically on the 2013 recommendations from ACM and IEEE (later called CS2013 [10]). We chose this document because it is the most recent revision of the ACM-IEEE guidelines, because it is the one most used by other authors we found in the literature, and also because it was used at USI as a reference for the development of the Master in Software and Data Engineering.

The most granular units in CS2013 is the topic — which represents something a student should learn — and the learning outcome — which describes something the student should know or be able to do. Knowledge units are collections of related topics and learning outcomes. For a given knowledge unit, after having learned all its topics, a student is expected to know all of the concepts, and be able to do all the tasks, described in each of its learning outcomes. The last concept in CS2013 is the knowledge area, which consists of a collection of related knowledge units.

When working with CS2013, professors are expected to identify the knowledge areas that are most relevant to their study program, and then rely on the knowledge units of each knowledge area as a guideline for designing or updating a study program [10]. The topics presented in knowledge units can either be tier 1 core, tier 2 core, or elective. According to CS2013, a computer science study program should aim at covering all tier 1 core knowledge units, most of the tier 2 core knowledge units, and a group of elective knowledge units. For tier 1 core and tier 2 core knowledge units, CS2013 provides an estimate amount of time, in hours, that should be set aside to teach the specific unit.

2.2 Management

In the context of managing study programs, the oldest example we found is from Gorman et Al., who, in 1970, presented a methodology to guide the design of study programs tailored to the skill requirements of manufacturing companies [6]. This methodology is divided into multiple steps: given a job, understand the required skills, draft a study program to teach those skills, let a committee review the draft and update it as necessary, assemble all the resources required to teach the study program, and finally teach the study program.

A more recent approach to study program design was presented by Ajanovski, who proposed a method that guides a team of professors through multiple iterations of a study program, until a general consensus on the contents of a study program is reached [1]. Each iteration begins with planning and design, followed by a review. Part of the evaluation of each study program is performed by manually mapping the proposed study programs to an underlying guideline (specifically, CS2013 [10]) and establishing dependencies and/or relations between courses in each study program.

Another approach we found in literature comes from Herbert et Al., who presented how their university revised some of its study programs through a process of interviews with professors and industry representatives [8]. The goal of each interview was to gather answers to a set of predetermined questions pertaining to the university's study programs. The authors then analysed these answers manually and used them to better understand what expectations the various interviewees had about the university's study programs.

A different type of approach was used by Nordstrom et Al., who, in their effort to reach ABET accreditation, relied on CS2013 [10] as a point of reference for redesigning study programs offered by their university [13].

2.3 Evaluation

Various authors showed that study programs can be measured by mapping them to an underlying guideline.

Krishna et Al. used questionnaires based on the CC2001 guideline [17] to obtain a coverage metric for their university's previous and current study program [11]. This metric is defined as the sum of expected hours covered for each field in the reference guideline. The authors obtained multiple coverage measures that were then used to compare two versions of their study program.

Camilloni et Al. defined a method to count the number of hours their students spent on each topic of their study program, tracking both in-class time based on the study program, and also by asking students to track time spent outside of class [3]. Then, they compared their findings with the GSwE2009 guideline [14] to evaluate their study program.

Nordstrom et Al. presented a few metrics for evaluating their university's study programs, such as measuring the rate of student enrollment, the rate of graduation, and the percentage of students that signed a working contract within six months of graduation [13].

Rowe et Al. presented a scoring system that relies on the IT2008 guideline [12]. This system can be used to evaluate and compare Information Technology study programs [15]. Briefly, the system performs a weighted sum of scores awarded to each suggested course from the underlying guideline based on how much each is covered by the actual study program.

2.3.1 Other Methods of Evaluation

Although not a metric, Ajanovski implemented study program evaluation by means of a visual mapping to the underlying guideline. This visualisation is designed to help faculty identify which topics of the guidelines are more covered and which are less covered.

A different type of evaluation strategy we found in literature consists of expert evaluation. This process was used by Gorman et Al. in their method to design study programs, by Ajanovski in their recursive approach to managing study programs [1], and by Herbert et Al. in their attempt at improving the study programs offered by their university [8]. In each case, this type of evaluation was not quantified, but it provided the various authors with additional points of reference to evaluate their study programs.

2.4 Exploration

In the literature, we found various techniques to explore both study programs and guidelines, all implemented in a visual way.

Auvinen et Al. presented a tool that allows faculty to specify courses as sets of learning outcomes, and then builds a graph of dependencies between courses, distinguishing between required or recommended courses [2]. Faculty can then use this graph to identify shortcomings of the study program by seeing, for example, which topics of a knowledge unit are not covered enough in the study program. The tool also offers students the ability to create custom study programs by defining a set of desired program outcomes, based on which the platform attempts to create per-student programs by comparing desired program outcomes and actual course learning outcomes.

Sekyia et Al. used natural language processing techniques on course syllabi for a study program of their university in order to heuristically build visual maps [16]. These maps were then used to understand relations between courses. An interesting aspect of this technique is that it doesn't require the direct participation of professors in the process, whereas other approaches required professors to provide all the data necessary to build these types of visualization — for example the one used by Auvinen et Al. [2], led to an estimated effort of about two hours per course.

Jafar et Al. proposed a few visualization techniques for CS2013 [10] to help faculty explore its contents [9]. Such visualizations include trees that allow the user to explore different depths of information in the guideline, or heat maps to compare relations between knowledge areas.

2.5 Conclusion

The work we found in literature shows various attempts to deal with the problems presented in Chapter 1. These solutions are focused on either the management of study programs, evaluation of study programs, exploration of a guideline, or a combination of these.

Of the literature explored, the two most promising results are the ones from Ajanovski and Auvinen et Al. Both authors presented a platform. Ajanovski implemented study program management and evaluation, with study programs linked to an underlying guideline. Auvinen et Al. also implemented study program management and evaluation, but their platform does not rely on an underlying guideline. A summary of the literature explored is presented in Table 2.1.

Paper	Has Platform	Management	Evaluation	Guidelines Exploration
Gorman et Al. [6]		✓	✓	
Ajanovski [1]	✓	✓	✓	
Herbert et Al. [8]		✓	✓	
Krishna et Al. [11]			✓	
Camilloni et Al. [3]			✓	
Nordstrom et Al. [13]			✓	
Rowe et Al. [15]			✓	
Auvinen et Al. [2]	✓	✓	✓	
Sekyia et Al. [16]	✓		✓	
Jafar et Al. [16]	✓			✓

TABLE 2.1: Summary of the literature explored.

With this thesis we propose a novel platform that integrates the management and evaluation of study programs and the exploration of the underlying guidelines, with study programs and courses linked to an underlying guideline.

In Chapter 3, we present the concrete implementation of the platform presented in this section, including a walk-through of the UI, an explanation of the underlying model, a detailed overview of its codebase, and other code-related elements.

Chapter 3

Platform

In this chapter we present the platform that we developed for this thesis. In Section 3.1 we illustrate the underlying model we designed for the platform. In Section 3.2 we provide a walk-through of the UI of the platform, showing how it works and how it is supposed to be used. In Section 3.3 we present the architecture of the platform we developed, including an overview of technical decisions we took, such as which technologies to use, how to organize our repository, and more. Finally, in Section 3.4 we describe how to build the platform, how to deploy it manually, and how to deploy it automatically.

3.1 Model

In this section we present the model we created for the domain of the platform. Figure 3.1 shows a UML representation of the model.

In a nutshell, universities offer study programs (modeled as `StudyPrograms`) to their students. These study programs are updated over the years. Each revision of a study program is modeled as a `StudyProgramEdition`. Conceptually, a `StudyProgram` is just a collection of `StudyProgramEditions`. Each edition of a study program contains courses. A course, modeled as a `Course`, can be revised year after year just like study programs. Each revision of a course is modeled as a `CourseEdition`. Similarly to `StudyProgram` and `StudyProgramEdition`, a `Course` is a collection of `CourseEditions`. Each `CourseEdition` is taught by instructors (modeled as `Instructors`) and the contents of each `Course` are modeled according to a `Guideline`. For now, the only guideline supported by the platform is `CS2013`. A description of all entities that appears in the model (see Figure 3.1) is presented in Subsection 3.1.1.

3.1.1 Model Description

Introductory note: each class in the model has a property named `id` that is intended to uniquely identify every instance of that given class among the others.

A `StudyProgramEdition` represents a concrete edition of a study program that was offered to students. A `StudyProgramEdition` has the following properties:

- `studyProgram`: the `StudyProgram` of which the current object is an edition.
- `requiredCourses`: a list of `CourseEditions` that students enrolled in the current `StudyProgramEdition` are required to take.
- `electives`: a list of `CourseEdition` that students enrolled in the current `StudyProgramEdition` can electively choose to take.
- `guidelines`: the reference `Guideline` that has been used to describe the contents of each `CourseEdition` that is part of the current `StudyProgramEdition`.

A `StudyProgram` represents a conceptual study program that has zero or more editions. A `StudyProgram` has the following properties:

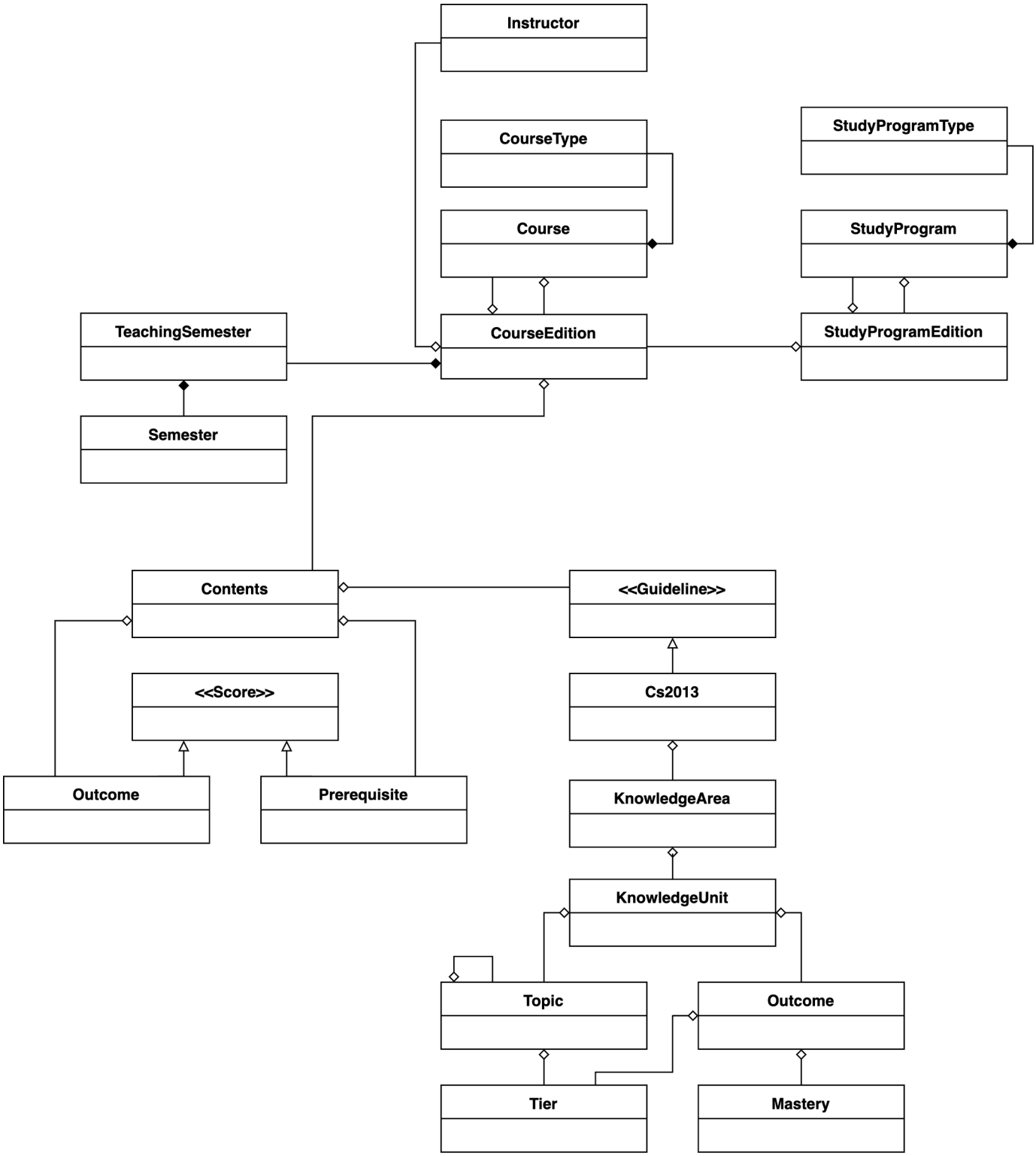

FIGURE 3.1: Model of the platform presented in this thesis.

- `acronym`: a user-friendly name used to identify a specific study program.
- `type`: the type of a specific study program. It can be any of `BACHELOR`, `MASTER`, `PHD`.

- `targetCredits`: the number of credits that a student must obtain in order to complete any of the editions of the study program.
- `editions`: the list of editions of the given study program.

`Credit` is an interface to be extended when modeling a specific type of credit. In our implementation, we modeled ECTS credits using the `Ects` class, whose sole parameter `credits` is a number representing an amount of ECTS.

A `CourseEdition` represents a concrete edition of a course that was offered to students. A `CourseEdition` has the following properties:

- `course`: the course of which the current instance is an edition.
- `name`: the name of the course for the current edition.
- `briefDescription`: a short description of the given course edition.
- `fullDescription`: a complete description of the given course edition.
- `notes`: any extra textual notes that don't fit into any of the available fields for course editions.
- `type`: the type of the given course edition. It can be any of `ATELIER`, `SEMINAR`, `LECTURE`.
- `prerequisiteCourses`: all courses every student is required to have taken in order to be able to take the given course edition.
- `credits`: a map that describes for every possible study program type, the amount of credits awarded to students from said study program type who pass the given course edition. This map makes it possible to describe situations where one course has a certain value in terms of credits for master students, and a different value for PhD students.
- `teachingSemester`: the year and semester during which the given course edition was taught or will be taught.
- `contents`: the contents of the given course editions, whose value depends on the chosen guideline. This parameter makes it possible to bind a guideline to the course.

A `TeachingSemester` represents a specific year and semester. It is composed of two properties: a number `year`, and a `semester` which is one of `SPRING` or `FALL`.

A `Course` represents a conceptual course that has zero or more editions. A `Course` has the following properties:

- `acronym`: a user-friendly name used to identify a specific course.
- `editions`: the list of editions of the given course.

`Contents` models the contents of an actual course. `Contents` has the following properties:

- `guideline`: the name of the guideline the given `Contents` refer to.
- `prerequisites`: a map that associates each outcome id from the chosen guideline to a value that is any of `REQUIRED`, `GOODTOKNOW`, `NOTREQUIRED`.
- `outcomes`: a map that associates each outcome id from the chosen guideline to a value that is any of `WILLBETAUGHT`, `MAYBEWILLBETAUGHT`, or `WILLNOTBETAUGHT`.

`Instructor` models an instructor that taught zero or more course editions. An `Instructor` has the following properties:

- `email`: email address of the instructor.
- `firstName`: first name of the instructor.
- `lastName`: last name of the instructor.
- `taughtCourseEditions`: a list of all the course editions that have been taught by an instructor.
- `relatedCourses`: all the courses for which the given instructor has taught at least one edition.

`Guideline` is an interface intended to be implemented by a concrete class for each guideline that the platform should support. In our case, we implemented `Guideline` in `CS2013`. `CS2013` has one field, `areas`, which is a list of `KnowledgeArea`.

`KnowledgeArea` represents a specific knowledge area of CS2013. `KnowledgeArea` has the following properties:

- `name`: the name of the given knowledge area as found in CS2013.
- `core1Hours`: the number of core 1 hours for the given knowledge area as found in CS2013.
- `core2Hours`: the number of core 2 hours for the given knowledge area as found in CS2013.
- `elective`: whether the given knowledge area is elective or not.
- `units`: a list of all `KnowledgeUnit` that are part of the given knowledge area.

`KnowledgeUnit` represents a specific knowledge unit of CS2013. `KnowledgeUnit` has the following properties:

- `name`: the name of the given knowledge unit as found in CS2013.
- `core1Hours`: the number of core 1 hours for the given knowledge unit as found in CS2013.
- `core2Hours`: the number of core 2 hours for the given knowledge unit as found in CS2013.
- `elective`: whether the given knowledge unit is elective or not.
- `topics`: a list of all `Topic` that are part of the given knowledge unit.
- `outcomes`: a list of all `Outcome` that are part of the given knowledge unit.

`Topic` represents a specific topic of CS2013. `Topic` has the following properties:

- `name`: the name of the given topic as found in CS2013.
- `tier`: the type of the given topic, any of `CORE1`, `CORE2`, or `ELECTIVE`.
- `subtopics`: a list of `Topic` that are listed as subtopics of the given topic in CS2013.

`Outcome` represents a specific outcome of CS2013. `Outcome` has the following properties:

- `name`: the name of the given outcome as found in CS2013.
- `tier`: the type of the given outcome, any of `CORE1`, `CORE2`, `ELECTIVE`.
- `mastery`: the type of mastery expected of students pertaining to the given outcome, any of `Familiarity`, `Assessment`, or `Usage`.

In Section 3.2 we offer a walk-through of the user interface of the platform.

3.2 Walk-Through

In this section we offer a walk-through of the user interface of our platform, showing every task that can be performed within each page of the platform. Every modification to data shown in the user interface is automatically synchronized with the backend.

3.2.1 Main Page

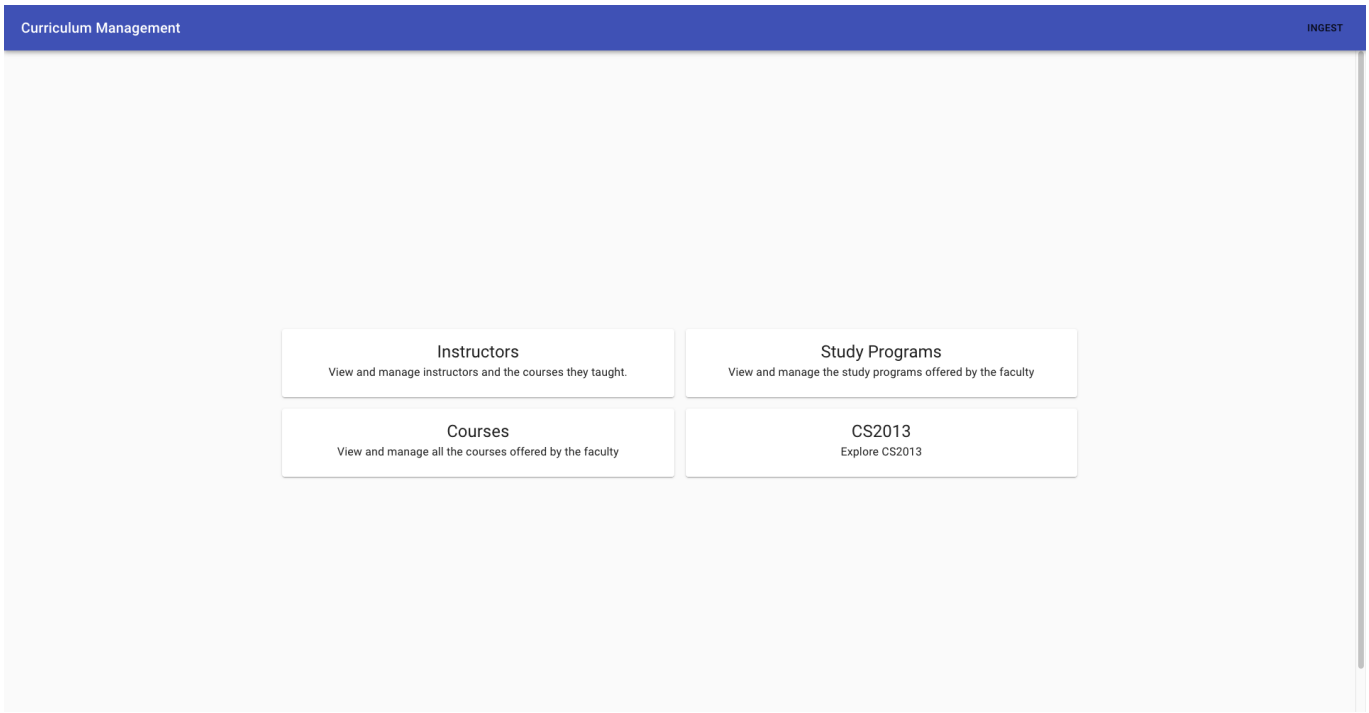

FIGURE 3.2: Platform: main page.

Figure 3.2 shows the main page that greets users when opening the platform. This home page serves to reach the four sections of the platform. Each of the four buttons in the home page redirects the user to the section described in the text of the button. For example, the "Instructors" button will bring the user to the section dedicated to instructor, the "Study Programs" button will bring the user to the section that contains study programs, and so on.

The blue navigation bar on top of the application is present in every page. On the left, it shows a textual description of the page the user is currently viewing. On the right, it shows custom actions when applicable.

Clicking on the "import" button in the navigation bar will open a dialog to import our mock representation of MSDE 2020 as explained in Section 4.2.

3.2.2 Courses

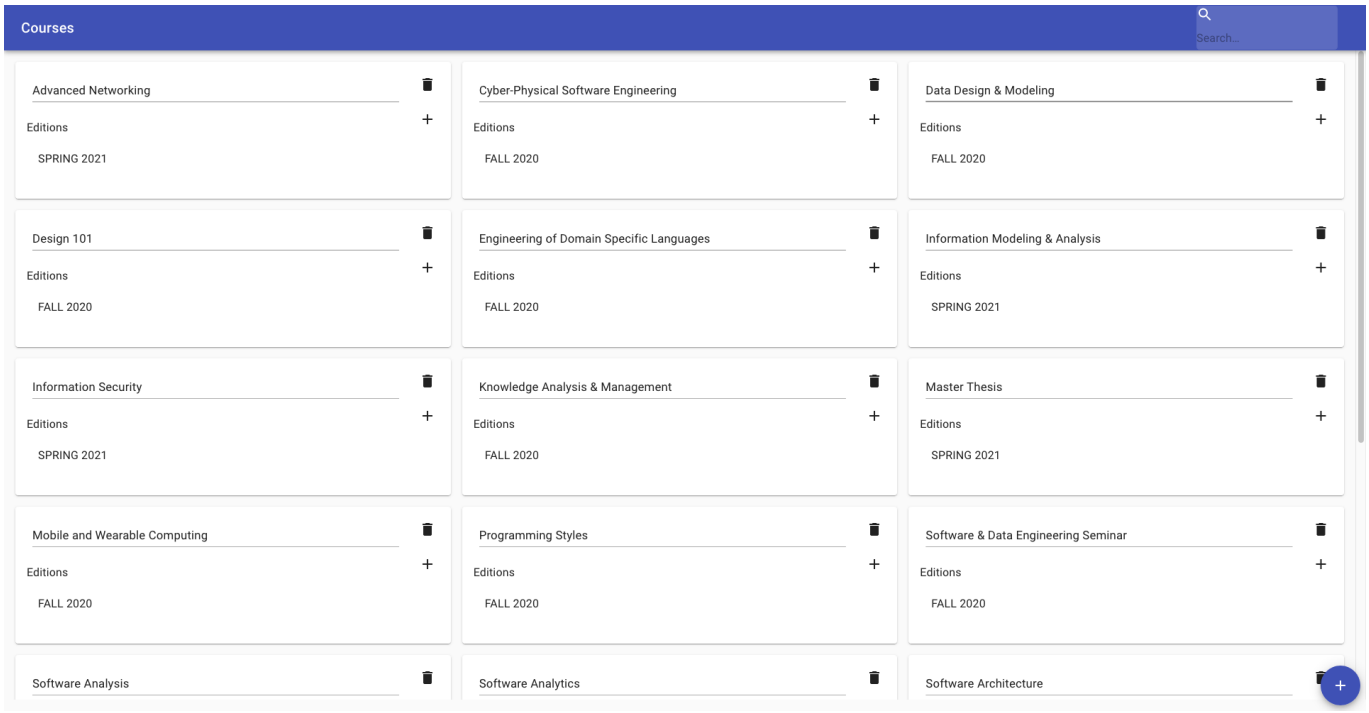

FIGURE 3.3: Platform: courses.

The page shown in Figure 3.3 presents the user with a list of all courses registered in the system. Each course is presented as an interactive card. The contents of each card can be edited by clicking on the text and changing it.

New courses can be created by clicking the blue "plus" button. Existing courses can be deleted by clicking on the garbage bin icon. Since course deletion is a destructive and irreversible action, when deleting a course, the user is presented with the dialog shown in Figure 3.4 and he/she has to confirm they want to delete the course.

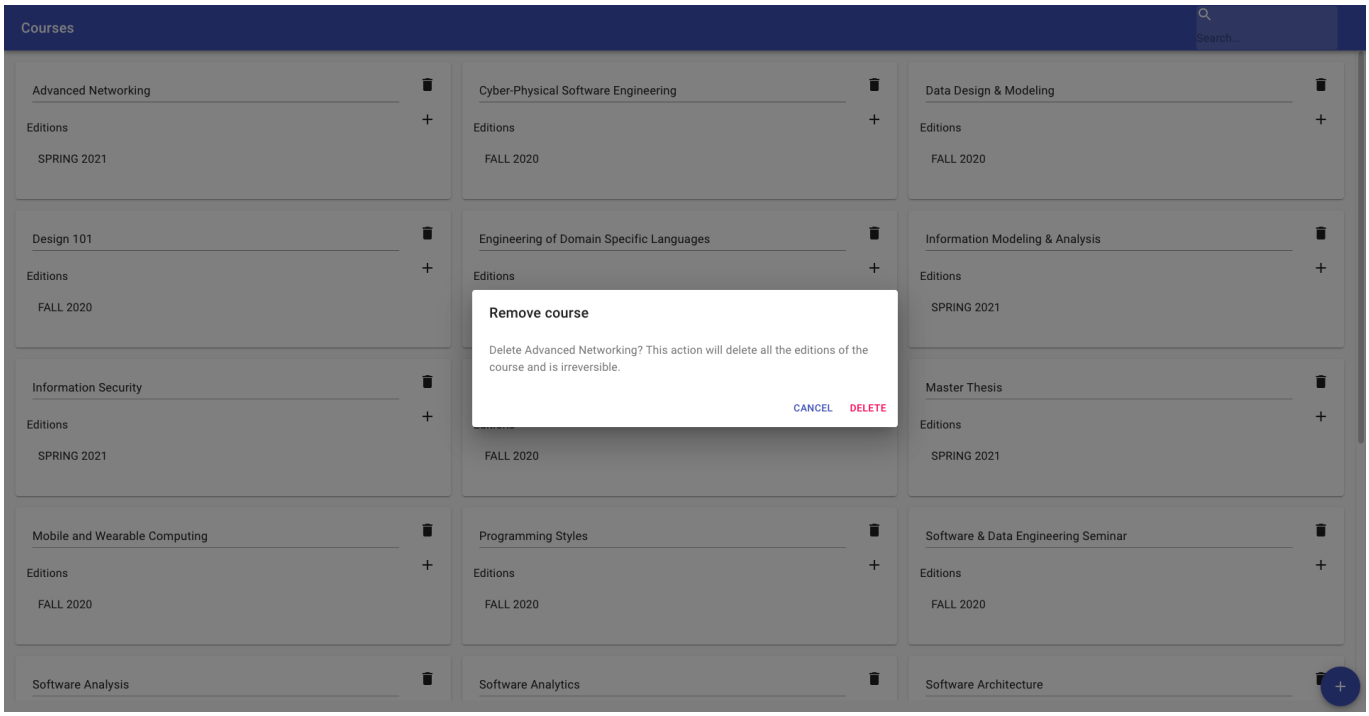

FIGURE 3.4: Platform: course deletion dialog.

To create a new course edition, click the "plus" button inside any given course. The frontend will automatically redirect the user to the newly-created course edition. The page to view and edit course editions is presented in Subsection 3.2.3.

3.2.3 Course Edition

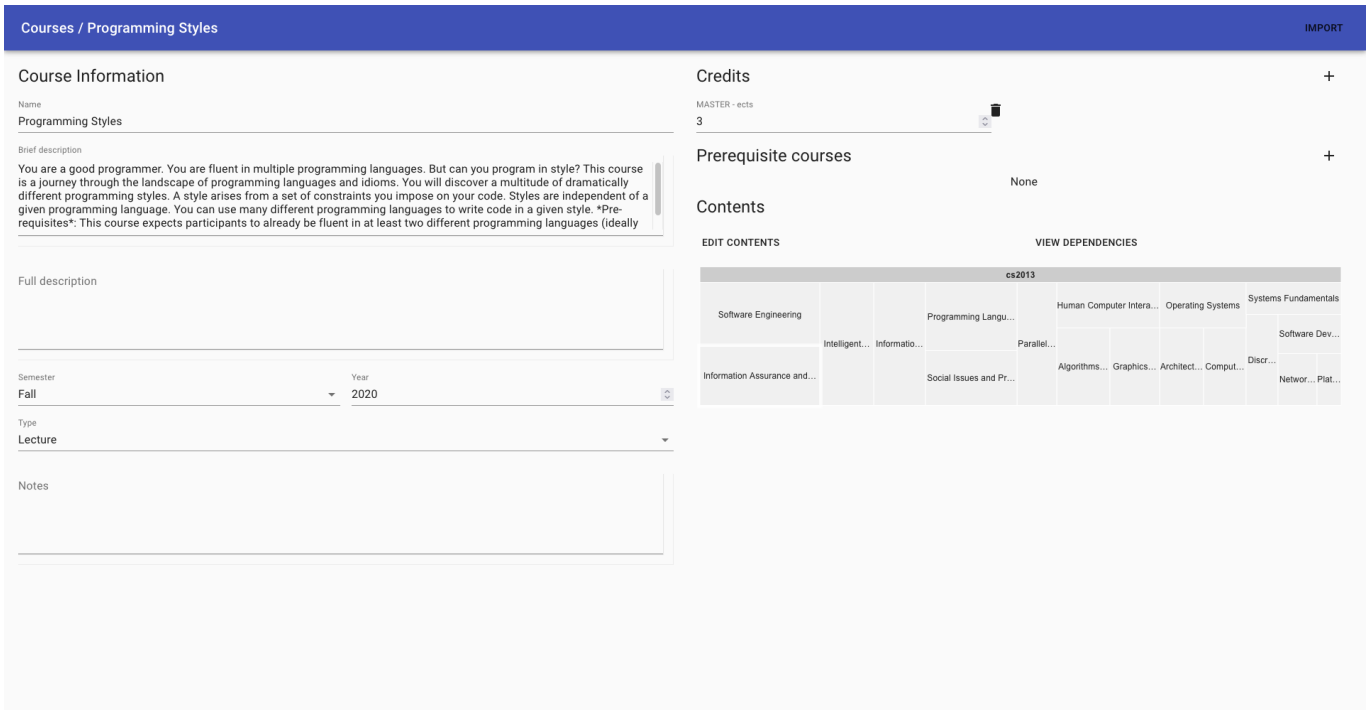

FIGURE 3.5: Platform: course edition.

Figure 3.5 displays the contents of a course edition. The left side of the view contains text boxes and menus for each property of the current course edition that the user can directly edit. The right side of the view contains other properties of the course edition that require more user interaction in order to be edited.

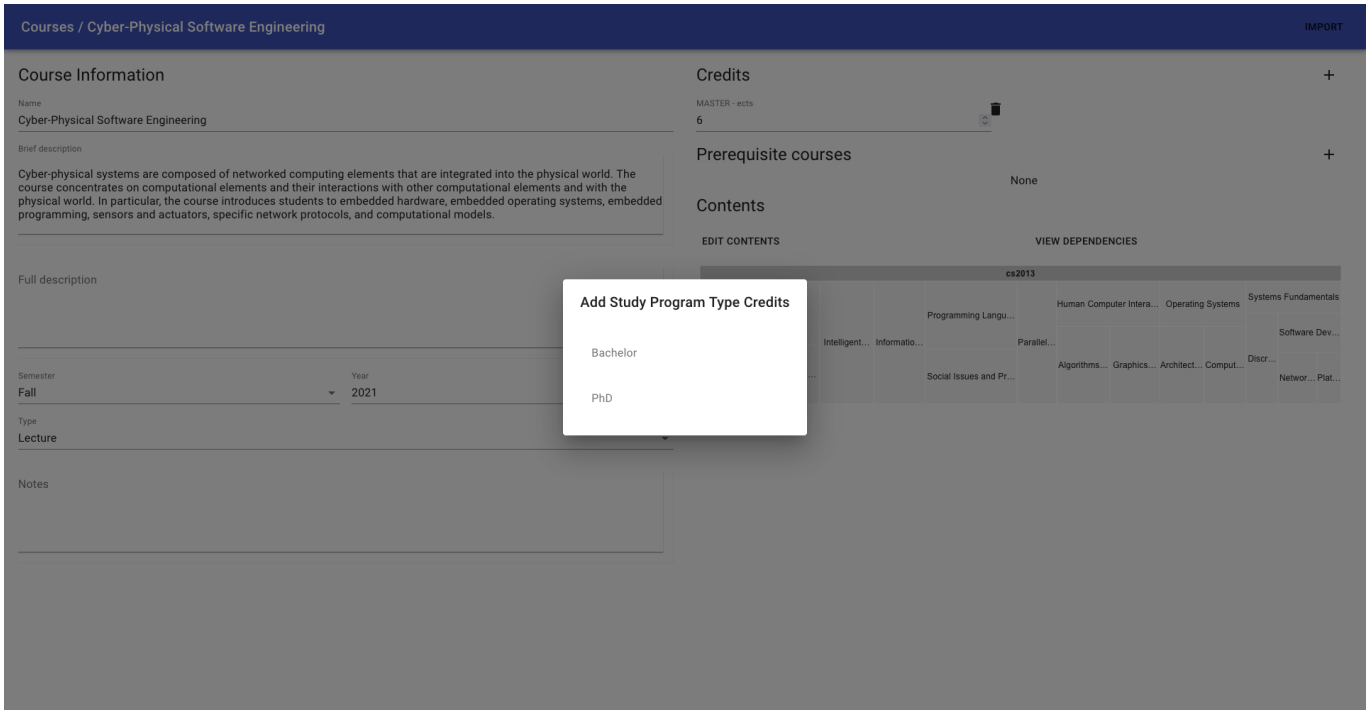

FIGURE 3.6: Platform: course edition add credits dialog.

The user can add credits for a specific study program type by clicking the "plus" button for the "Credits" view. Clicking this button will open the dialog shown in Figure 3.6. From this dialog, the user can select any of the study program types for which no credit amount has been specified yet. Once a study program type has been added, the user can edit the text field containing the numeric value of credits for the given study program type.

Credits for a specific study program type can be removed from a course edition by selecting the garbage bin icon. Since the action is easily reversible, deletion happens automatically, without requiring explicit user confirmation.

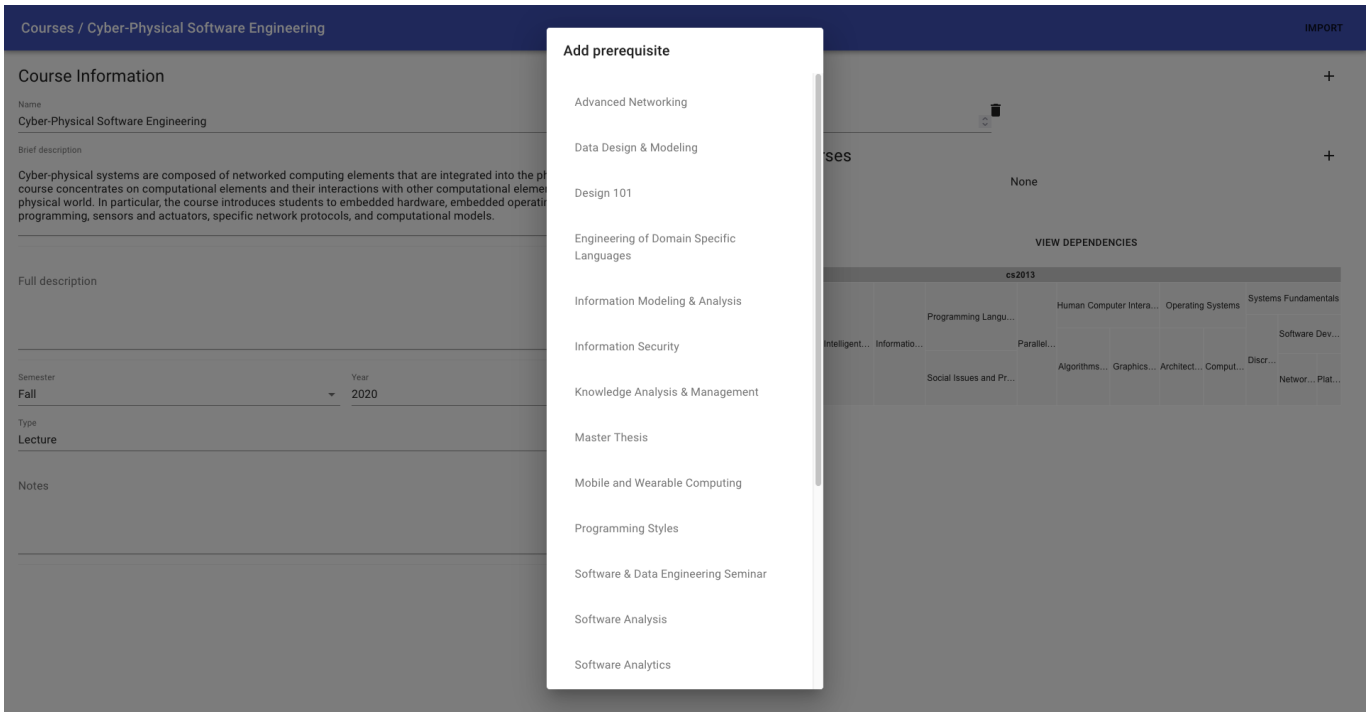

FIGURE 3.7: Platform: course edition explicit dependencies selection.

Below "Credits", the user can specify explicit course dependencies by clicking the "plus" button for the "Prerequisite courses" view. Clicking this button will open the dialog shown in Figure 3.7. From this dialog, the user can select any other course that is required for the current course edition.

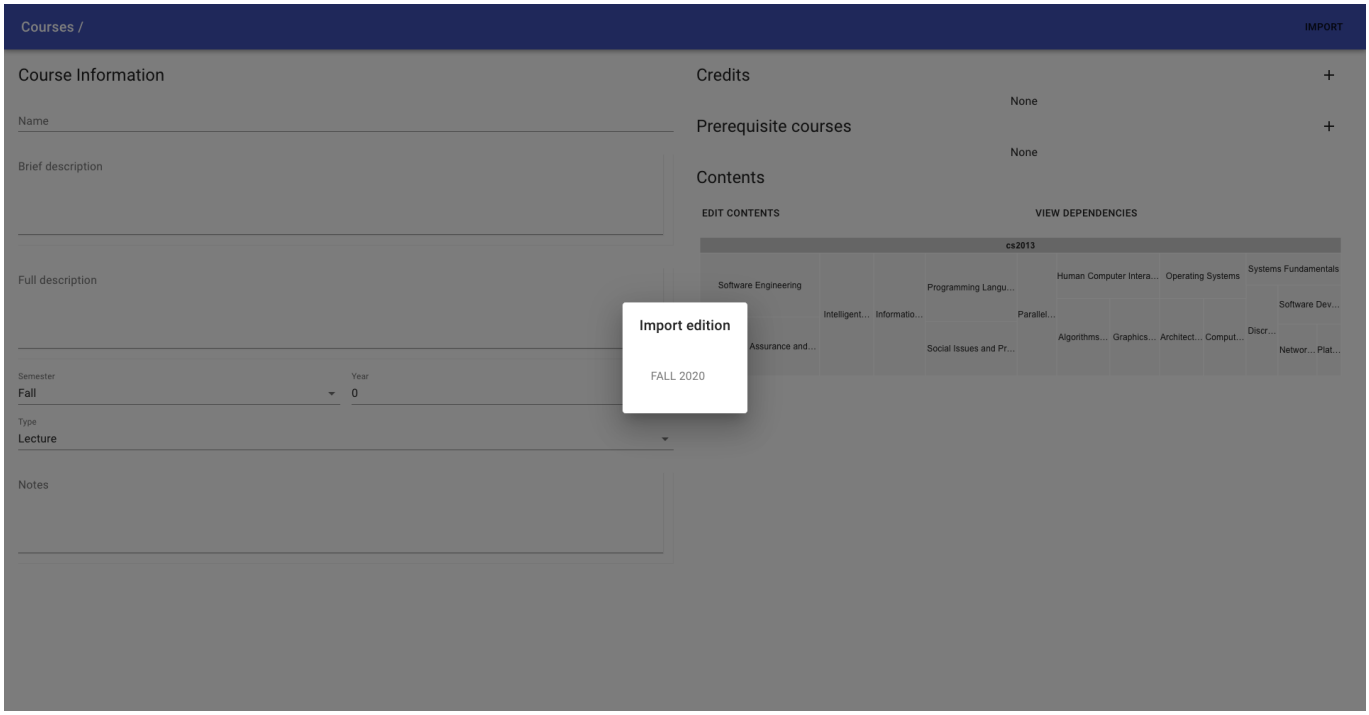

FIGURE 3.8: Platform: course edition import dialog.

By clicking the "import" button in the navigation bar, the user will be able to populate the contents of

the current course edition by copying the contents of another edition of the same course. After clicking said button, the user will be presented with a dialog to select the course edition to copy from as shown in Figure 3.8.

The "Contents" section of this page shows the course contents as they relate to the underlying guideline. For our platform, we currently support only CS2013, so by default, all courses use CS2013 as the underlying guideline. By clicking on the "Edit contents" button, the user will be redirected to the course edition contents page described in Subsection 3.2.4. By clicking on the "View dependencies" button, the user will be redirected to a page displaying the list of courses that cover the contents marked as required or optional for the current course edition. This page is explained in Subsection 3.2.5. Finally, the treemap of the "Contents" section shows how the course's outcomes cover CS2013. An explanation of how to use the treemap is provided in Subsection 3.2.10.

3.2.4 Course Edition Contents

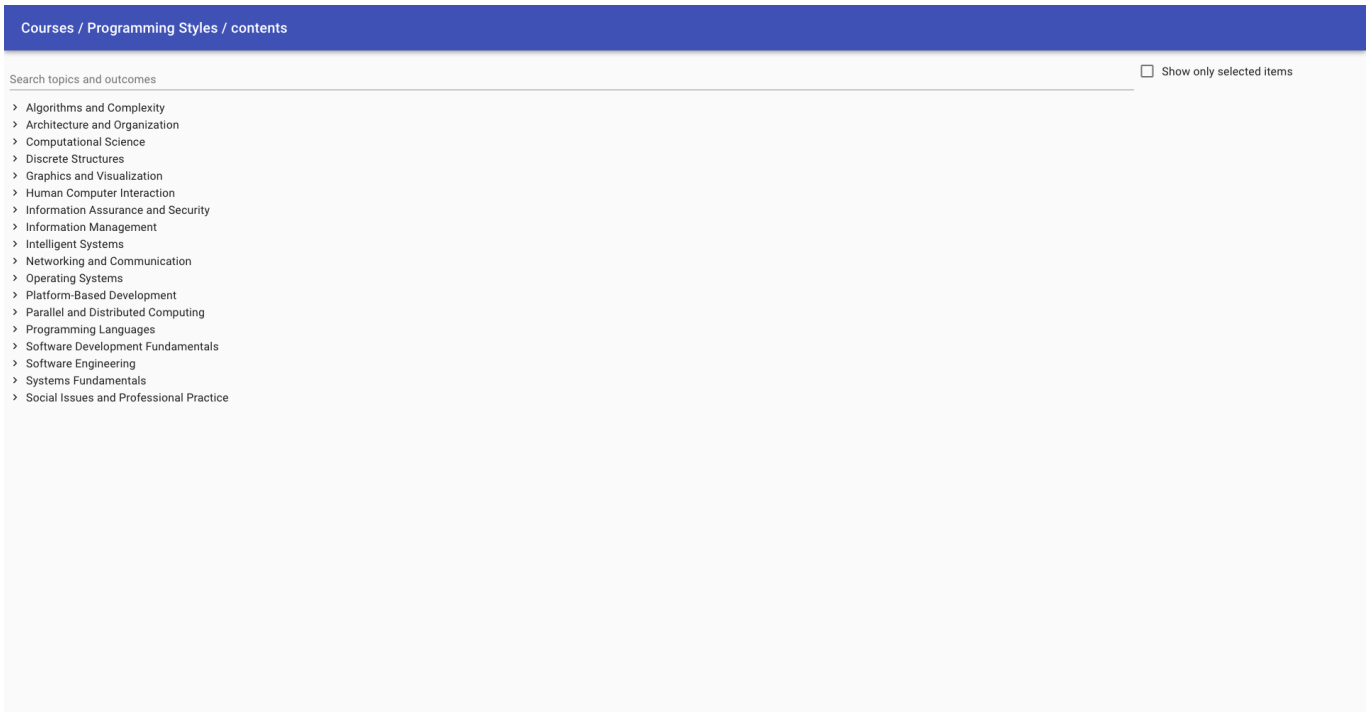

FIGURE 3.9: Platform: course edition contents.

Figure 3.9 shows the page where users can edit the contents of a course edition. Specifically, from this page, the user can specify contents of a course edition from CS2013. As mentioned in Subsection 3.2.3, the only guideline currently supported is CS2013.

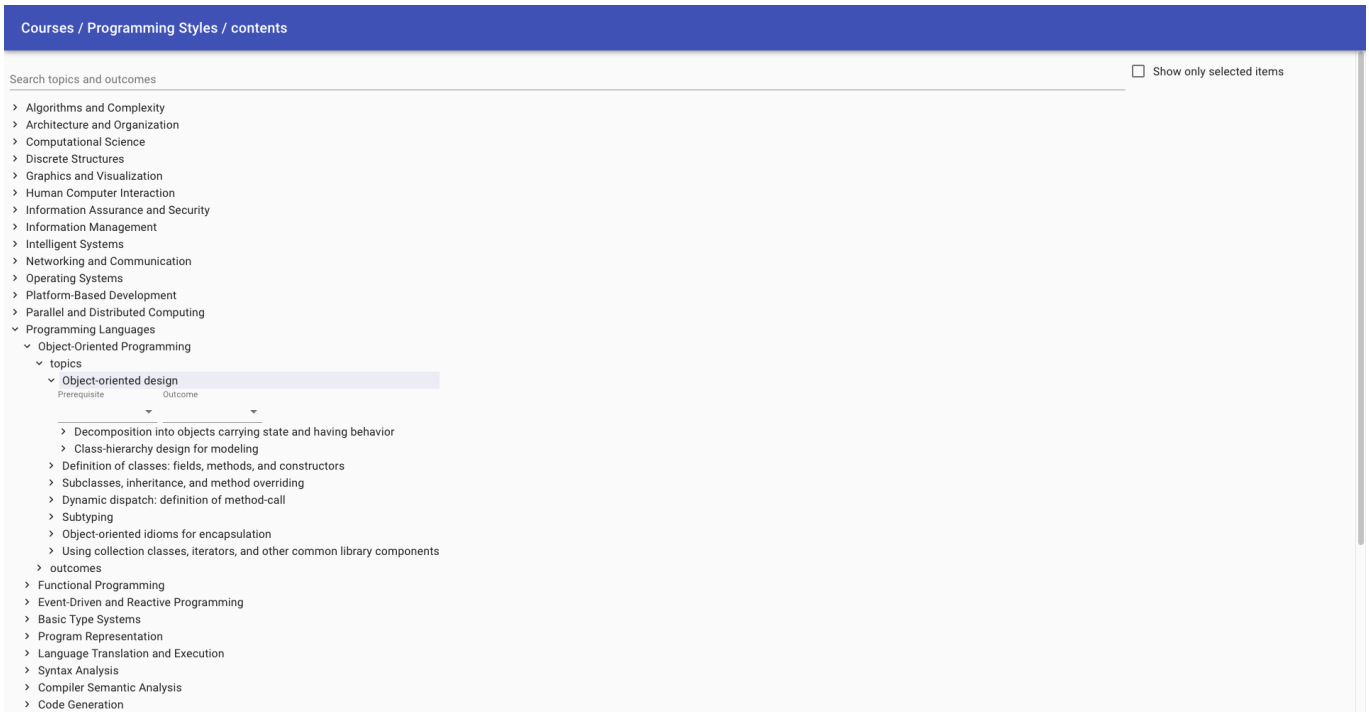

FIGURE 3.10: Platform: course edition contents (expanded).

CS2013 is presented as a hierarchical tree of knowledge areas that contain knowledge units. Each knowledge unit is broken into topics and outcomes. All nodes are collapsed by default.

To navigate the tree, click on any item and, when possible, the tree will show the children of the selected node. For example, by clicking a knowledge area, the tree will show its knowledge units. Expanded nodes can be collapsed with another click. Figure 3.10 shows the expanded view of this tree.

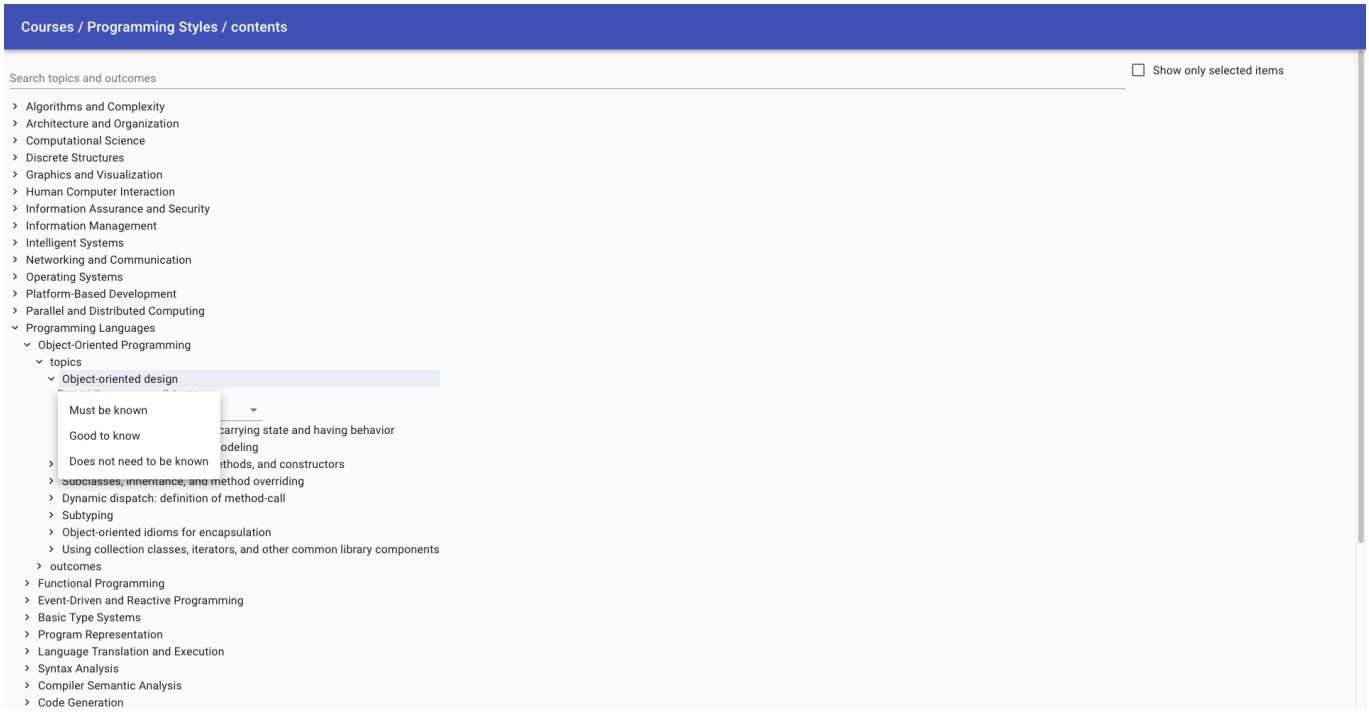

FIGURE 3.11: Platform: course edition contents selection.

In order to specify whether an item is required, optional, or not required, the user must click on the "Prerequisite" menu. To specify whether an item will be covered, might be covered, or won't be covered by the course edition, the user must click on "Outcome" menu. Figure 3.11 shows the menu for "Prerequisites".

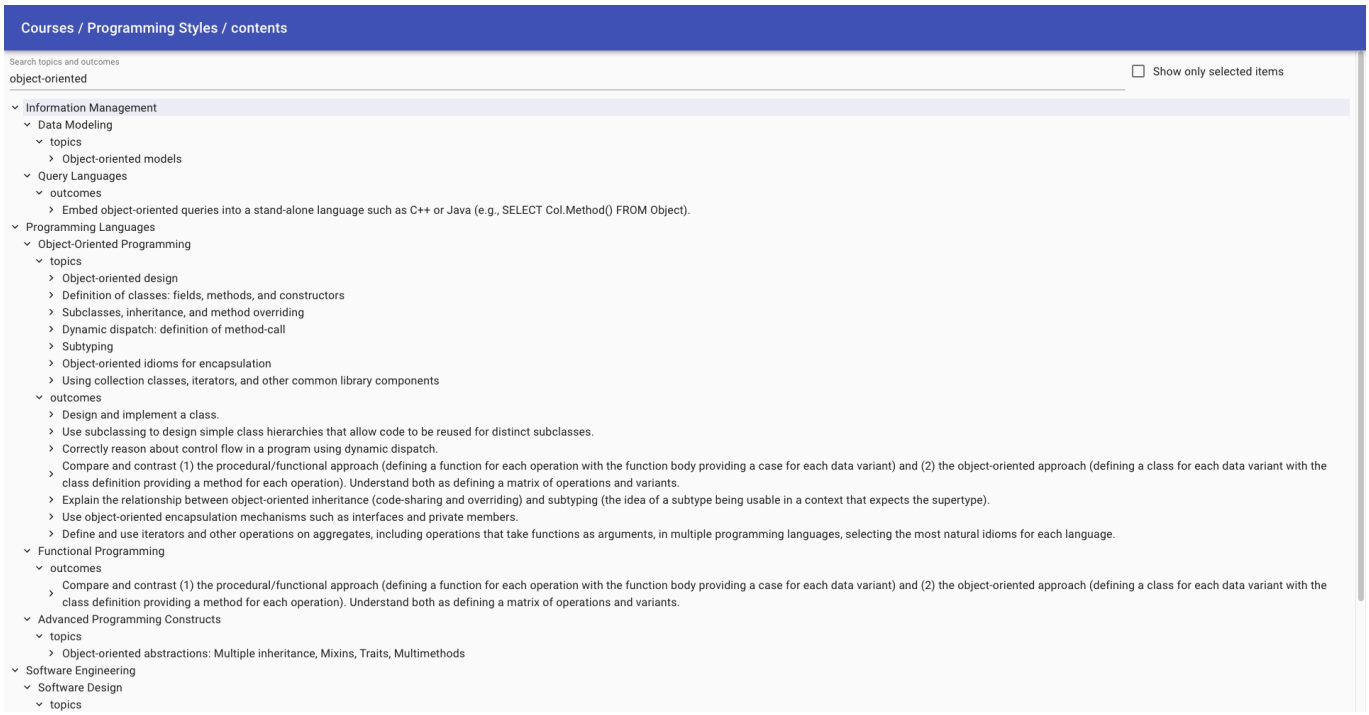

FIGURE 3.12: Platform: course edition contents text search.

Given that the tree contains 2′222 topics and outcomes, it can be cumbersome to navigate. To assist the user in finding specific items, a search bar is provided. When typing a string in the search bar, the page will update to display only those knowledge areas, knowledge units, topics, and/or outcomes that match the user's query. If a knowledge area matches the query, all its knowledge units, topics, and outcomes are displayed. If a knowledge unit matches the query, all its knowledge units and outcomes are displayed. Figure 3.12 shows an example of filtered tree.

Any item that doesn't match the user's query is removed from the tree. To see the entire tree again, remove the query from the search bar.

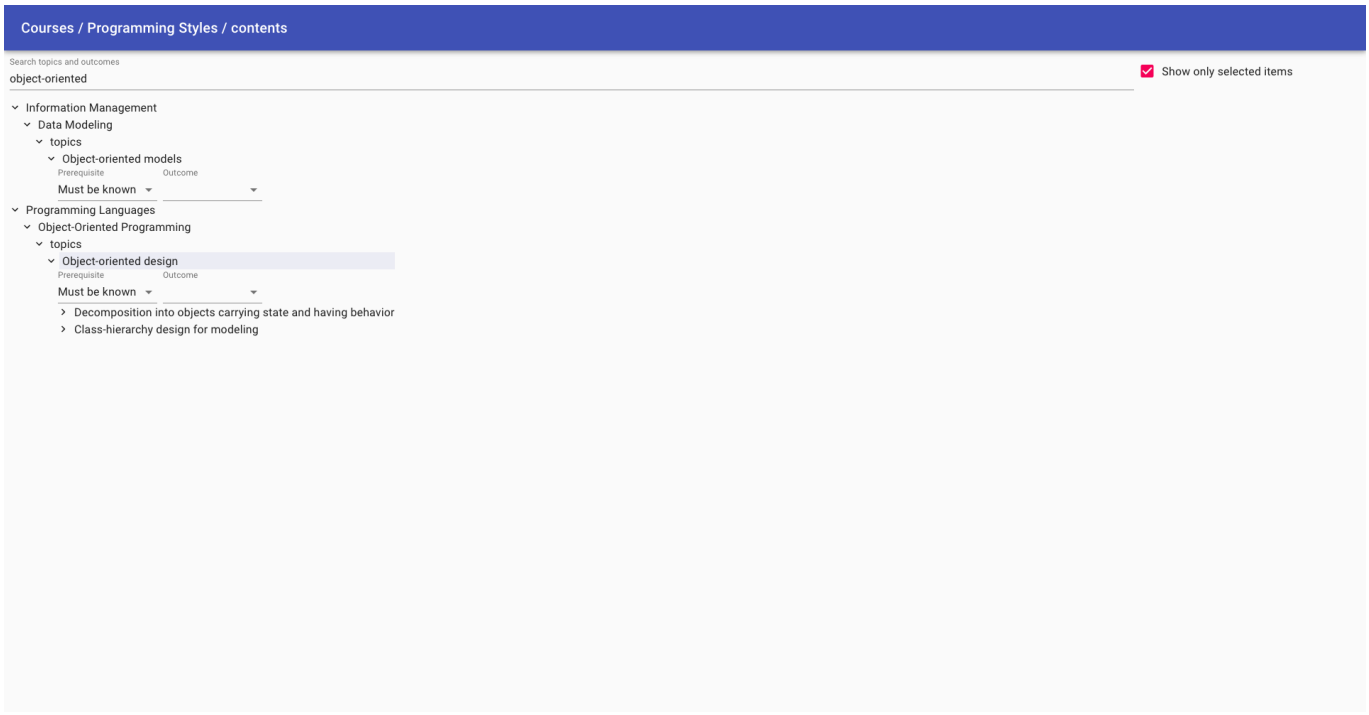

FIGURE 3.13: Platform: course edition contents filter unselected.

In order to retrieve only the items that have already been selected, the user can select the "Show only selected items" checkbox. The view will update to display only those items which have been selected.

3.2.5 Course Edition Dependencies

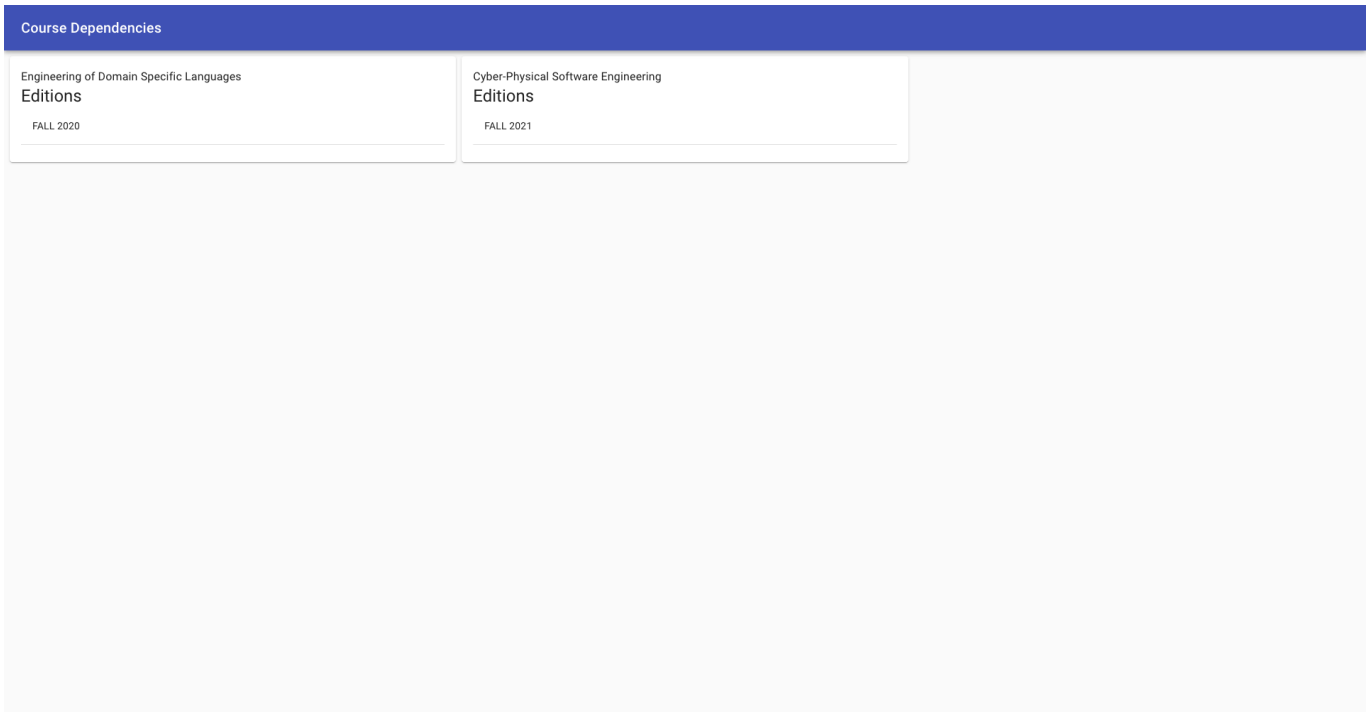

FIGURE 3.14: Platform: course edition dependencies.

The page shown in Figure 3.14 displays the course editions on which a given course edition depends. This dependency relation is computed implicitly from the contents specified in the "Contents" page. Every course edition that appears in this page was marked as covering a topic or producing an outcome that is marked as required or optional for the selected course edition. The page will display a message to the user in case no implicit dependency is found.

3.2.6 Instructors

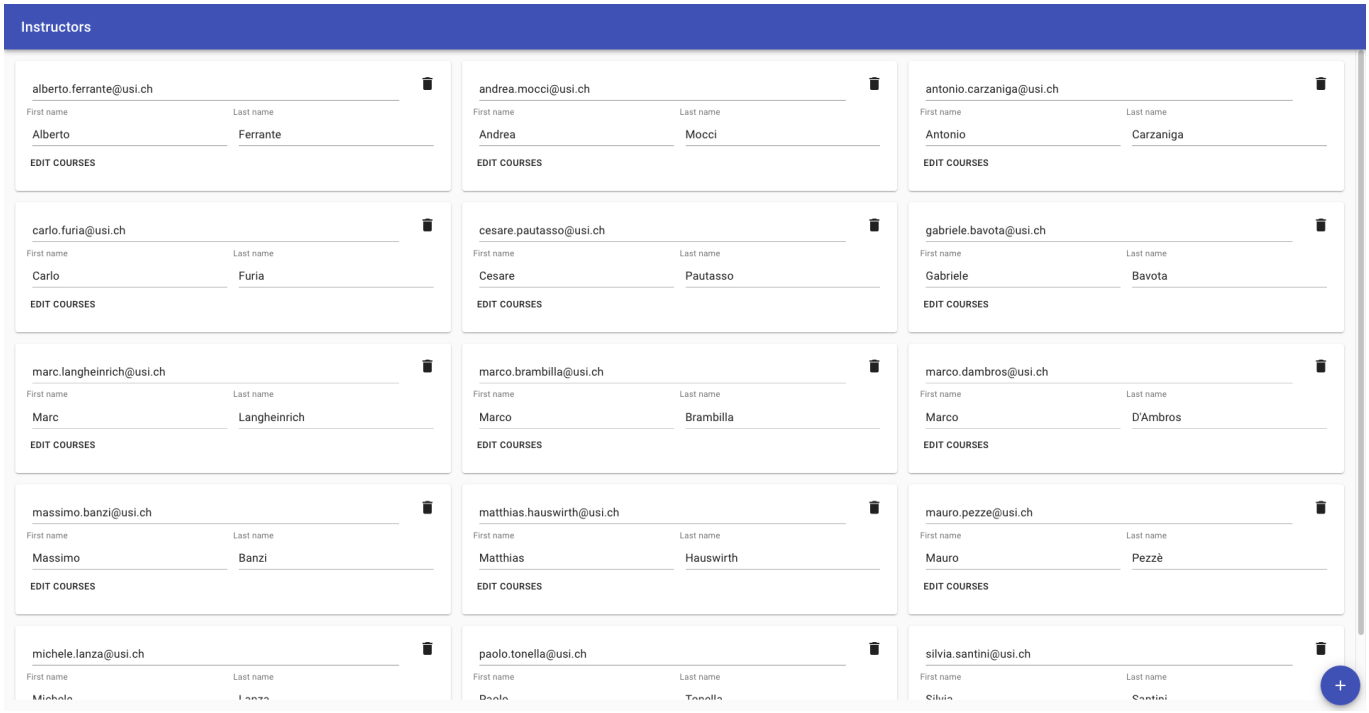

FIGURE 3.15: Platform: instructors.

The page shown in Figure 3.15 displays all instructors registered in the system. Each instructor is presented as an interactive card. Similar to the cards for courses and study programs, the contents of each card can be edited by clicking on the text and changing it.

New instructors can be created by clicking the blue "plus" button. Existing instructors can be deleted by clicking on the garbage bin icon. Since instructor deletion is irreversible, when deleting an instructor, the user is presented with the dialog shown in Figure 3.16 and he/she has to confirm they want to delete the selected instructor.

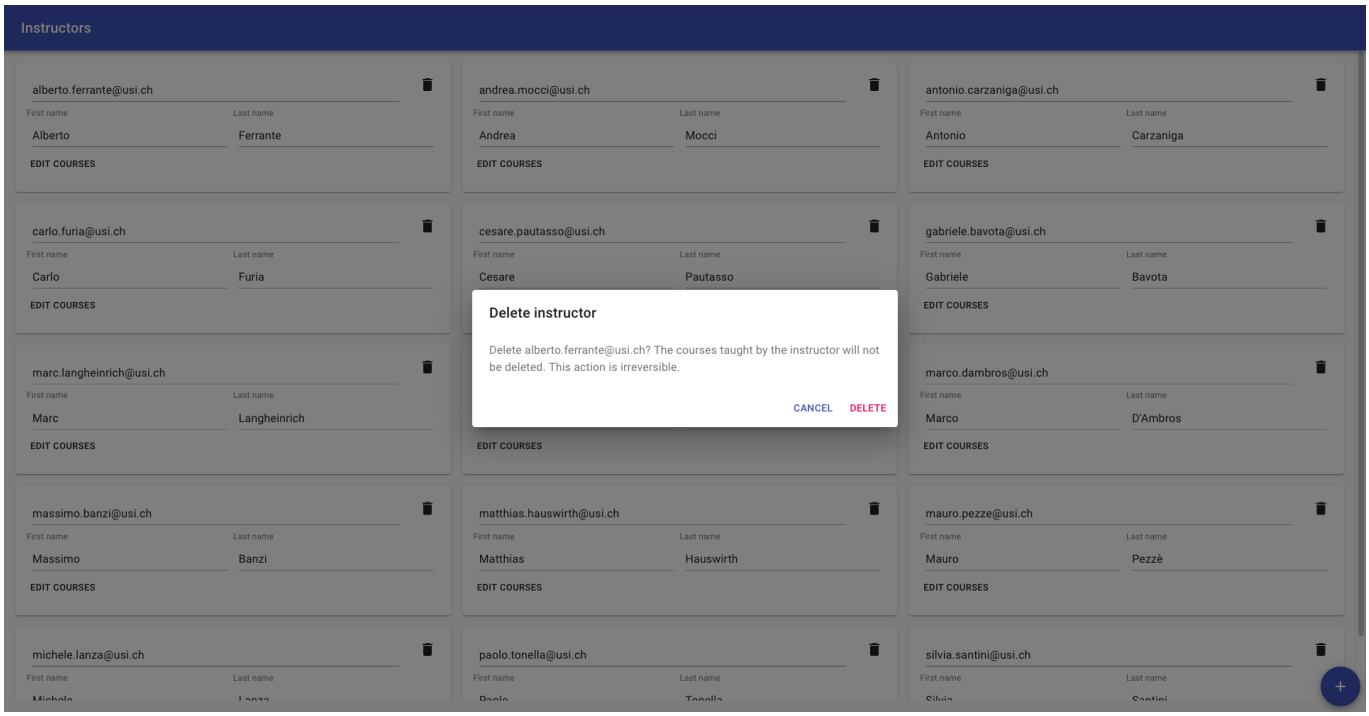

FIGURE 3.16: Platform: instructor deletion dialog.

To specify which courses an instructor has taught or will teach, select the "Edit courses" button: this will redirect the user to the instructor courses page explained in Subsection 3.2.7.

3.2.7 Instructor Courses

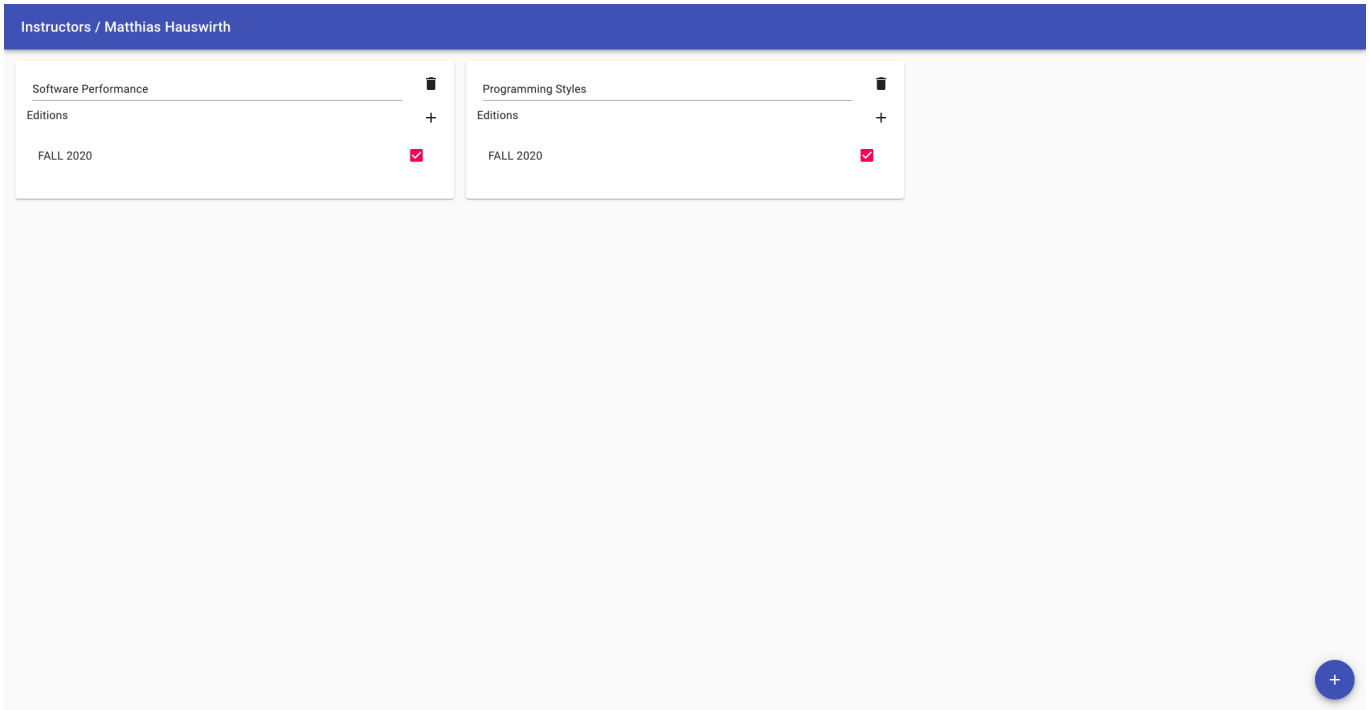

FIGURE 3.17: Platform: instructor courses.

The page shown in Figure 3.17 allows users to specify which courses the selected instructor has taught, is teaching, or will teach. When a course has been added to this page, users can specify which editions of the course the selected instructor has taught by toggling the checkbox: a magenta check mark indicates that the instructor has taught the selected course edition, whereas a missing check mark indicates that the instructor didn't teach the specific course edition[1].

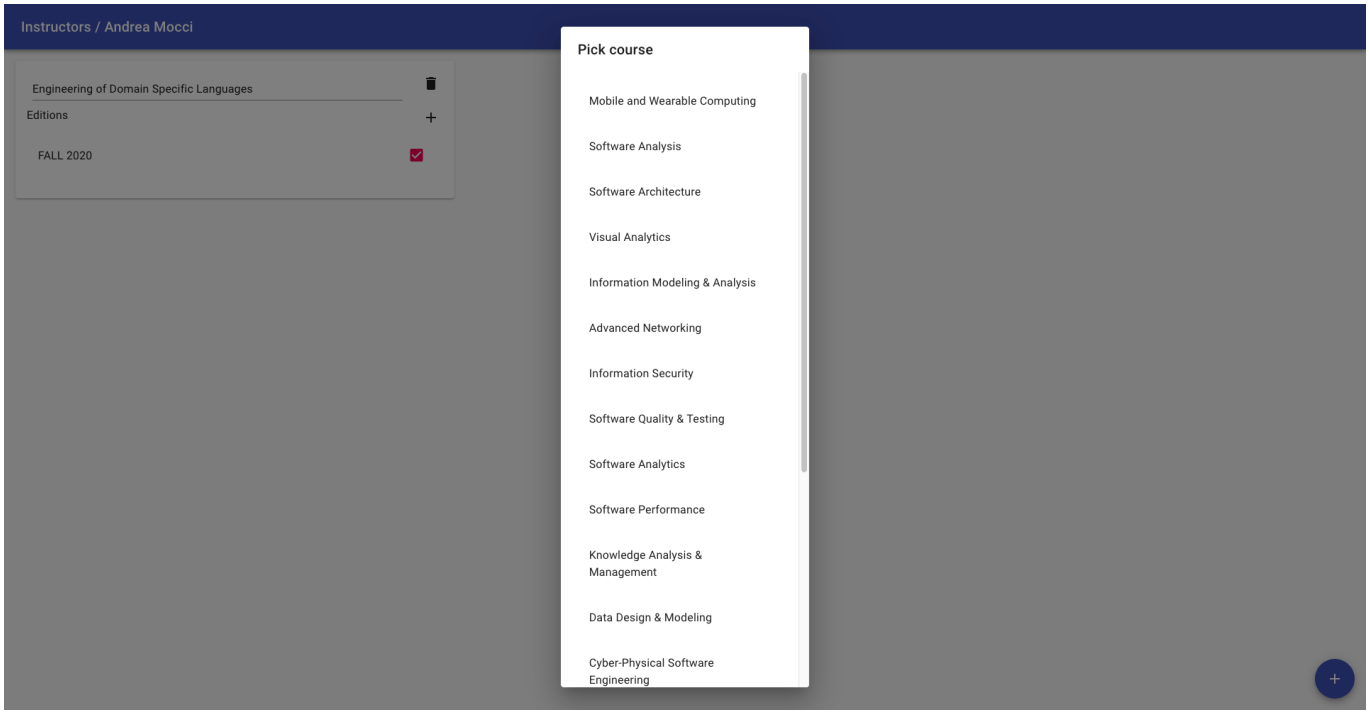

FIGURE 3.18: Platform: instructor course selection.

By hovering on the blue "plus" button, users can choose to either create new courses for the current instructor, or register existing ones. When choosing an existing course, users will be presented with the dialog shown in Figure 3.18. This dialog contains a list of all courses that are not yet registered for the selected instructor. Users can also create new editions of a course by selecting the "plus" button to the right of the "editions" label.

[1]There is a known bug with course edition selection: when toggling the checkbox, the user will be redirected to the course edition selected, even if this was not their intention. We were unable to solve this bug.

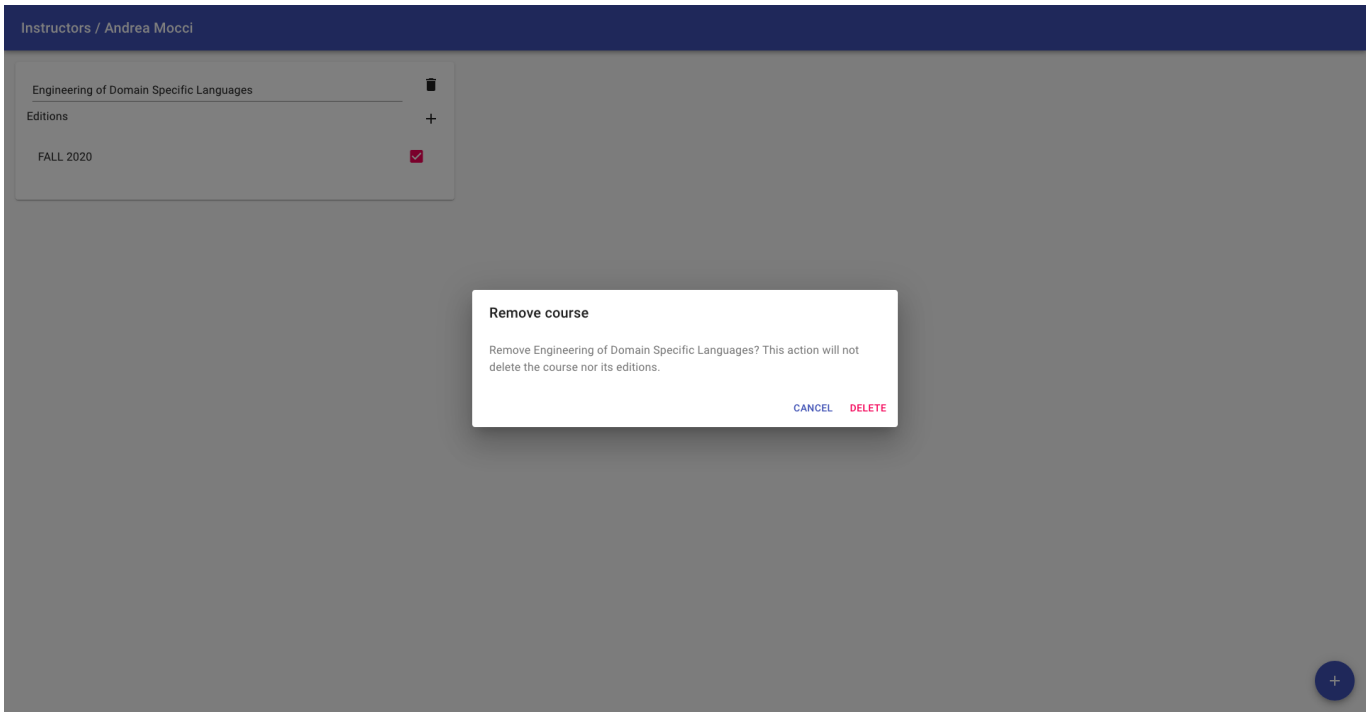

FIGURE 3.19: Platform: instructor course removal.

Courses can be removed from an instructor by selecting the garbage bin button. Since this action is not trivial to undo, users will be prompted for confirmation before proceeding as shown in Figure 3.19.

3.2.8 Study Programs

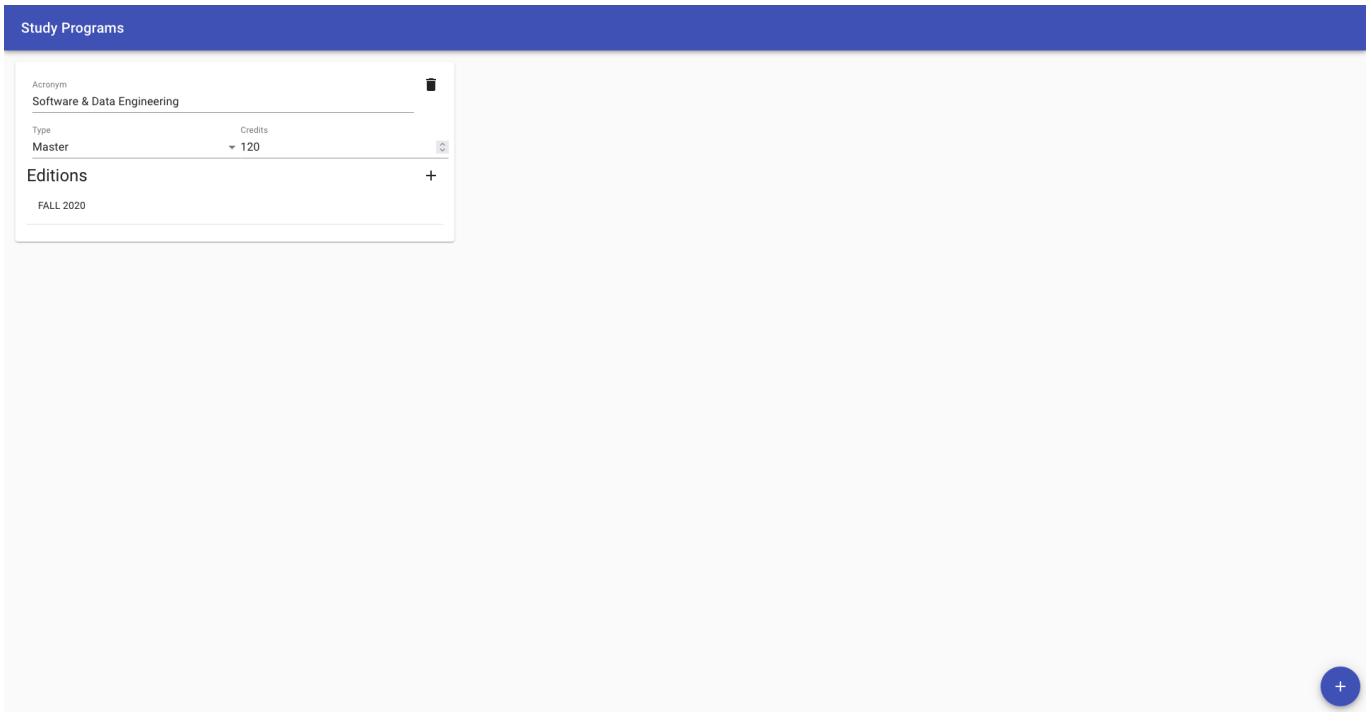

FIGURE 3.20: Platform: study programs.

The page shown in Figure 3.20 shows all the study programs registered in the system. Each study program is presented as an interactive card. The contents of each card can be edited by clicking on the text and changing it.

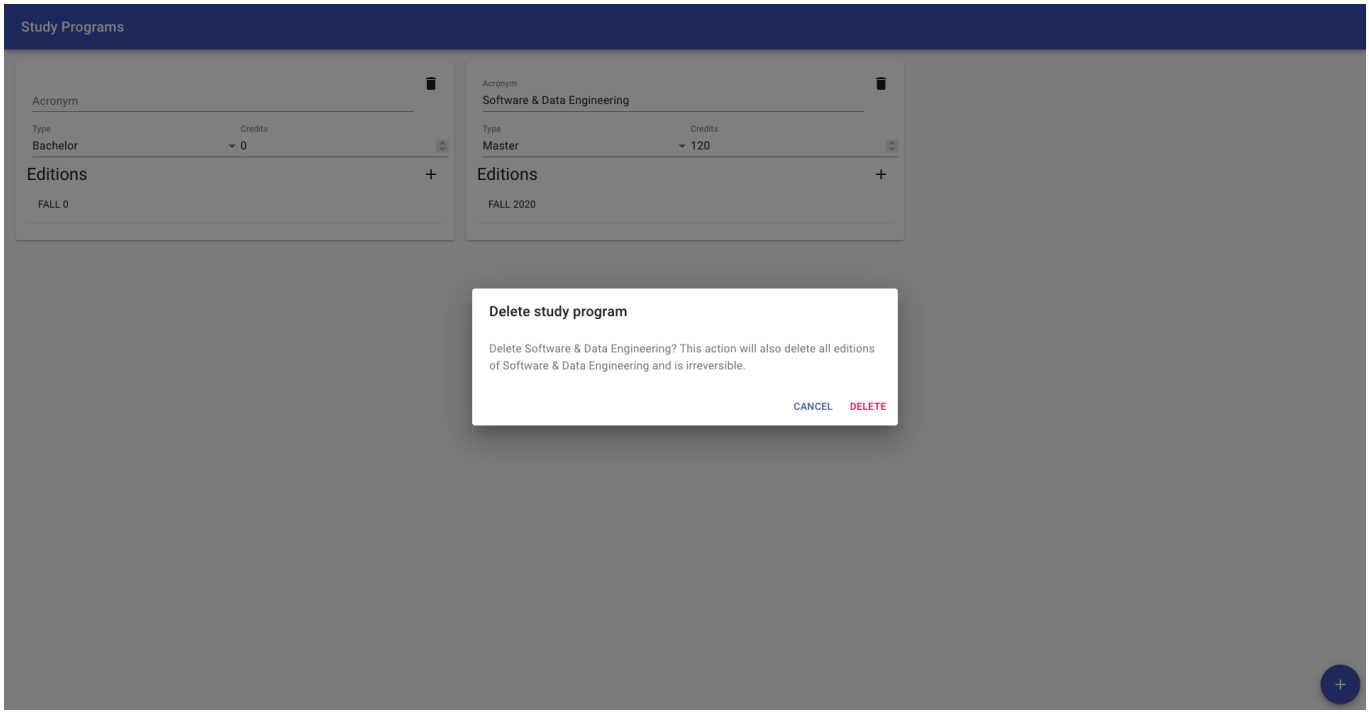

FIGURE 3.21: Platform: study program deletion dialog.

New study programs can be created by clicking the blue "plus" button. Existing study programs can be deleted by clicking on the garbage bin icon. Since study program deletion is irreversible, when deleting a study program, the user is prompted for confirmation as shown in Figure 3.21.

To create a new study program edition, click the "plus" button inside a study program's card. The frontend will automatically redirect the user to the newly-created study program edition. The interface to manage study program editions is explained in Subsection 3.2.9.

3.2.9 Study Program Edition

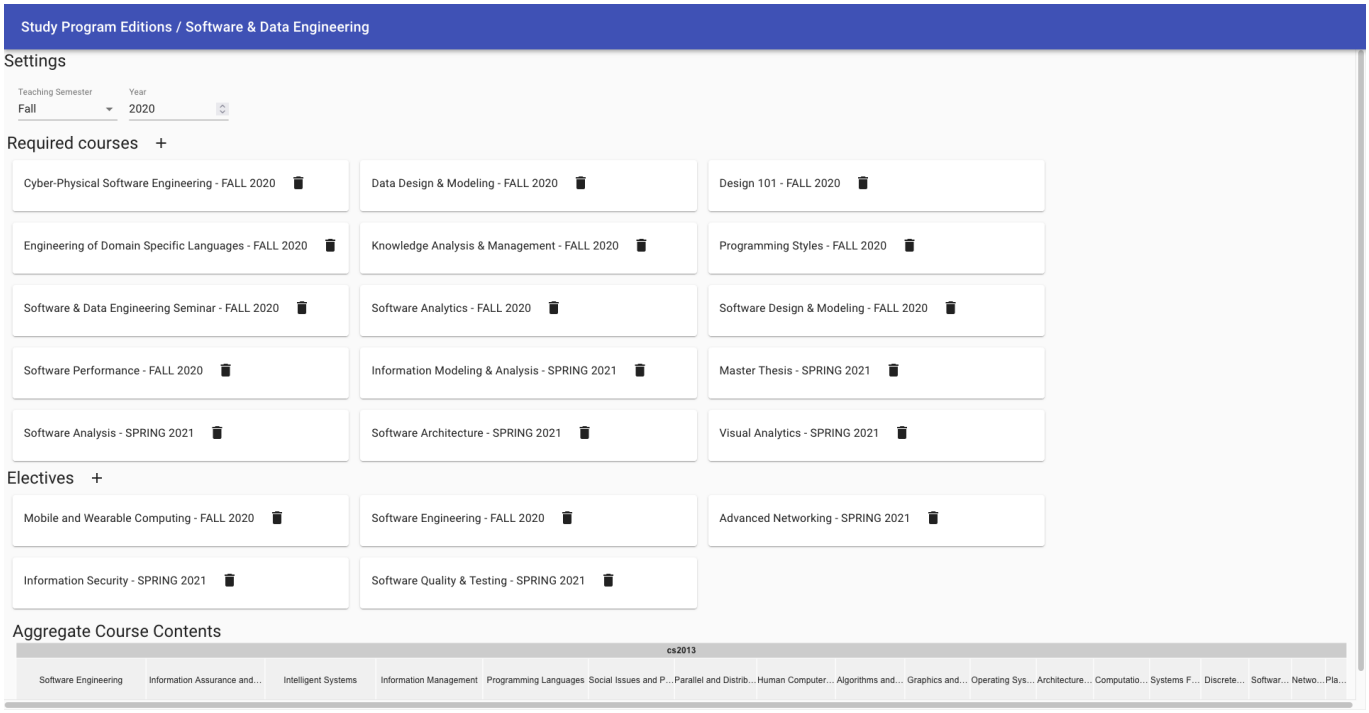

FIGURE 3.22: Platform: study program edition.

Figure 3.22 displays the contents of a study program edition. The page is split into four sections. In the "Settings" section, the user can customize textual properties of the selected study program. In the "Required courses" and "Electives" sections, the user can specify the required and elective courses of the current study program. Finally, the "Aggregate Course Contents" section shows all the topics and outcomes covered by all course editions of the current study program, grouped together, within CS2013.

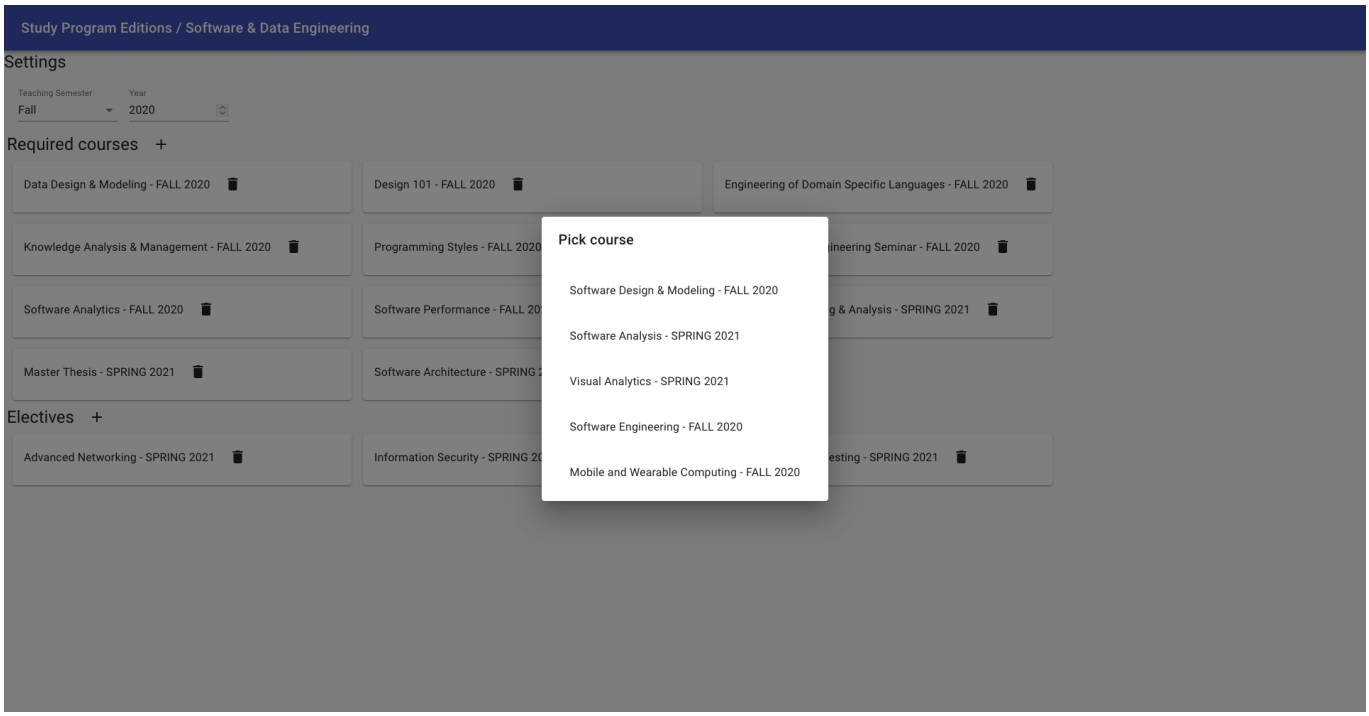

FIGURE 3.23: Platform: study program edition add course edition dialog.

The user can add required or elective course editions by clicking the "plus" buttons next to "Required courses" and "Electives". Clicking those buttons will open the dialog shown in Figure 3.24. Using this dialog, the user can select any of the available course editions that are not part of the study program yet.

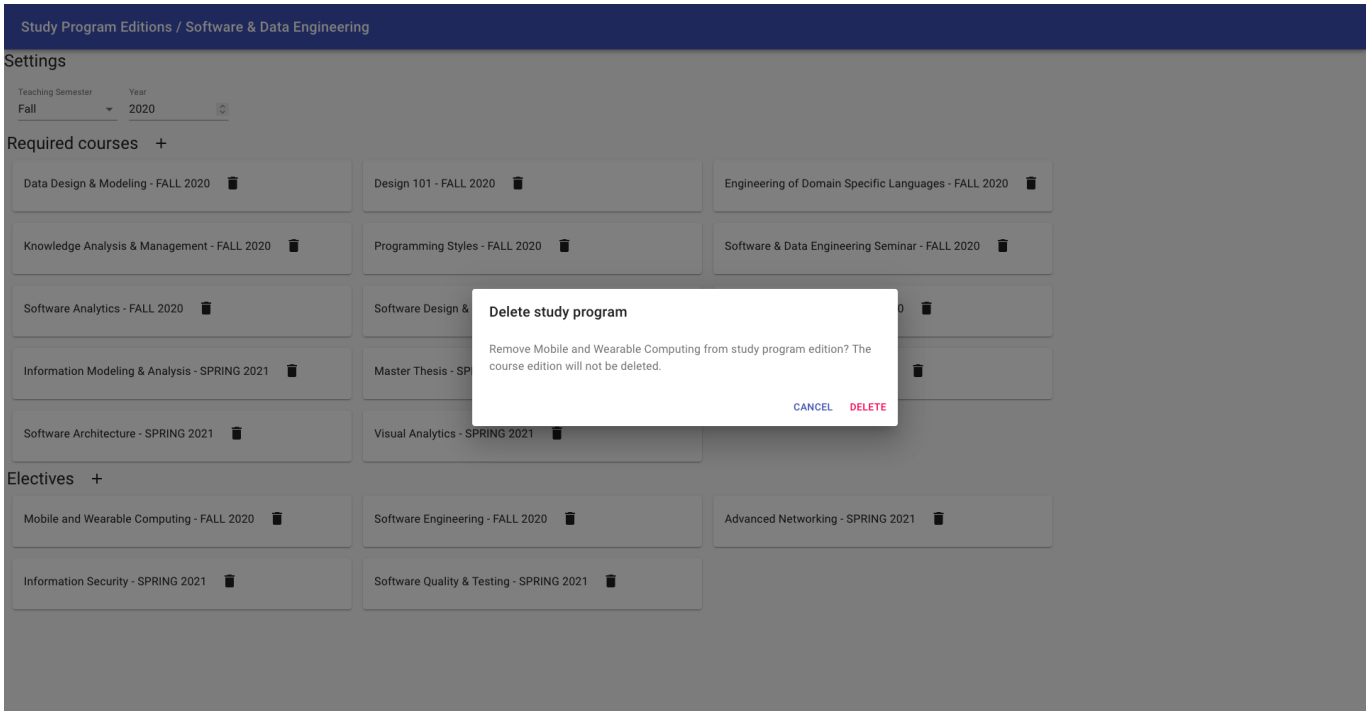

FIGURE 3.24: Platform: study program edition delete course confirmation dialog.

The user can remove required and elective course editions by clicking the garbage bin next to each

course edition card. Before deleting a course edition, users will be prompted for confirmation as shown in Figure 3.24.

3.2.10 Guidelines

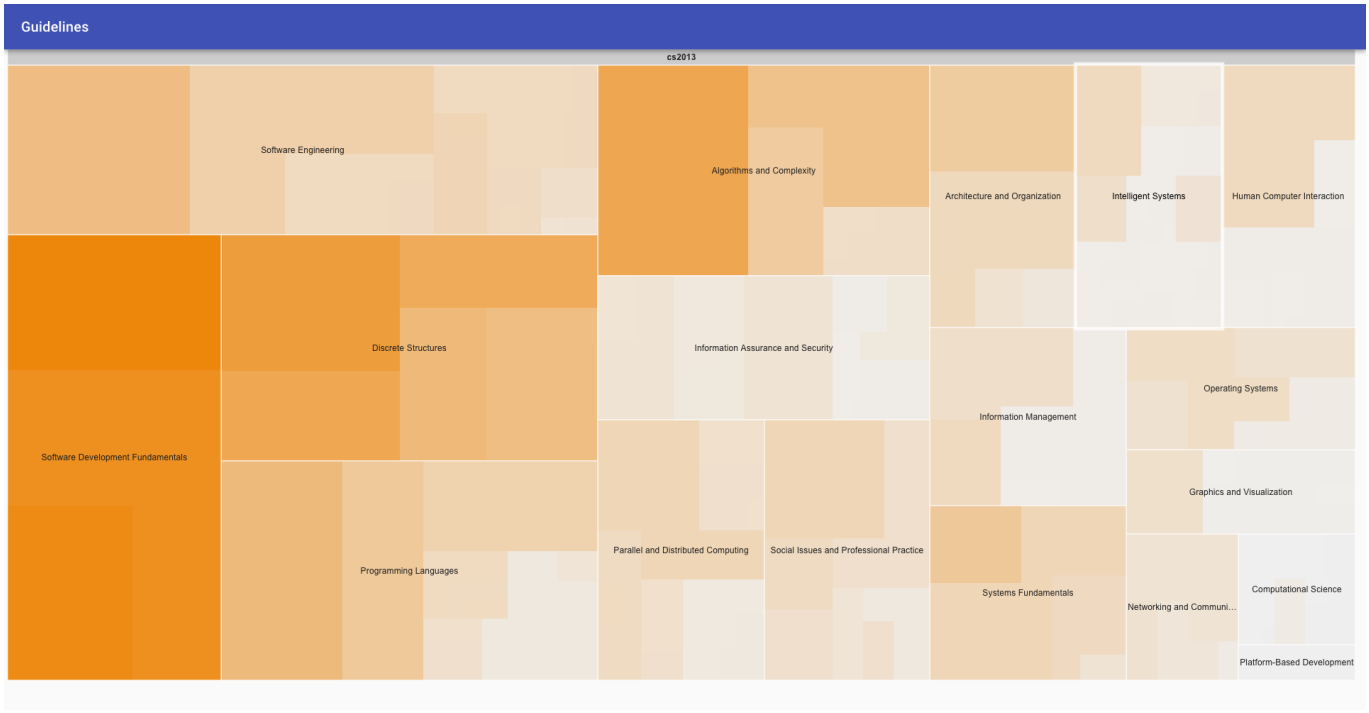

FIGURE 3.25: Platform: CS2013 guideline, knowledge areas.

Figure 3.25 shows the CS2013 guidelines in a treemap view. CS2013 associates a number of hours required to complete for each core 1 and core 2 knowledge unit. The size and color of each tile in the treemap is proportional to the sum of hours associated to each knowledge unit: larger tiles represent more expected hours compared to smaller and greyer tiles. The color of each tile represents the hours per knowledge unit within the knowledge area tile. As a result, each tile might contain multiple gradations of color depending on the distribution of expected hours among the knowledge units that are part of the given knowledge area.

The coloring of tiles is relative to the set of values displayed, and does not take into consideration any value that is currently not visible. As a result, it is not possible to compare color levels across multiple views of the treemap.

FIGURE 3.26: Platform: CS2013 guideline, knowledge units.

Clicking on a knowledge area updates the treemap to display the contents of the selected knowledge area. Figure 3.26 shows the knowledge units of the "Algorithms and Complexity" knowledge area. Similarly to the view of all knowledge areas, the size and coloring of this view is proportional to the number of expected hours associated to each knowledge unit.

FIGURE 3.27: Platform: CS2013 guideline, topics.

Clicking on a knowledge unit will show its topics. Since CS2013 does not offer a measure of expected hours for topics and outcomes, each visible tile will have the same size and color, with the exception of tiles representing topics that contain subtopics: in that case, the tile will be slightly larger and brighter.

To navigate back one level in the visualization, press right click on the treemap.

In Section 3.3 we dive deeper in the actual code and present how we organized the platform, technical decisions, and various other aspects of the code we wrote. More specifically, our explanation is broken in three subsections for the three main areas of the platform: the server, the API exposed by the server, and the web-app that interacts with the server using its API.

3.3 Architecture

In this section we describe the architecture of the platform we built for this thesis. At a high level, the platform is organized as a client-server web application. In Subsection 3.3.1 we explain the architecture of the backend (also called "server" interchangeably). In Subsection 3.3.2 we explain the API that the frontend uses to communicate with the backend. Finally, in Subsection 3.3.3 we explain the architecture of the frontend (also called "client" interchangeably). For both the backend, API, and frontend we present the technologies used, technical decisions we made, and other relevant information on the topic.

For the implementation of this platform we planned to create a usable and extendible prototype, rather than a prototype that is completely unusable and that needs to be thrown away after this thesis. In Chapter **??** we present many insights on how the platform could be extended.

3.3.1 Backend

The backend is responsible for serving the web application to all users and handle their requests. The backend is therefore also responsible for storing any relevant data provided by users and give back said data when requested and/or necessary.

We wrote the backend of our platform in Java. We picked this language because of its strong type system and because we are more familiar with this language compared to other strongly-typed options such as Scala or C++.

Spring and Spring-Boot

The actual server is implemented using the Spring and Spring-Boot libraries. Spring is a Java library that makes extensive use of decorator and enables a very specific programming style in Java. Spring is focused on dependency injection and inversion of control. Furthermore, Spring makes extensive use of decorators, which are a Java feature that allows developers to add new behaviour to Java classes. For example, by using the `@Autowired` decorator before an object property, upon initialization of the object, Spring will automatically instantiate an object of the specified type and assign it to the property decorated with `@Autowired`.

We decided to use Spring because it is a requirement to use Spring-Boot, a Java library for the creation of HTTP servers that we are most familiar with. To create an HTTP server in Java using Spring-Boot, the only required step is to decorate a class using the `@SpringBootApplication` decorator. Rest endpoints for the HTTP servers can be created using the `@RestController` decorator, followed by the `@RequestMapping(route)` decorator, where the route parameter is a string specifying the API endpoint that the given endpoint is serving.

MongoDB and Mongo-Data

To implement data persistence in platform we chose to use MongoDB, a NoSQL, document-based database management system. We decided to use MongoDB because the model presents many-to-many relations:

for example, each instructor can teach zero or more course editions, and each course edition is taught by one or more instructors. Being unfamiliar with database management in Java, we decided to use MongoDB in order to let the database handle these types of relation rather than having to figure out how to implement such relations in Java using a SQL database.

Binding between our Java application and the Mongo database is handled using Mongo-Data, a library built on top of Spring-Boot that offers various decorators and other facilities to interact with MongoDB from Java. We chose to use Mongo-Data because it is recommended by the Spring documentation[2], and we also estimated that learning and using this library would be faster than implementing each query ourselves manually.

Mongo-Data offers object-relational mapping tools that make it possible to treat MongoDB logic as Java objects and methods. Specifically, this library allowed us to create plain Java objects that, by use of the `@Document` decorator, were automatically translated into MongoDB documents. Every time a new object from class decorated with `@Document` is instantiated, Mongo-Data takes care of running the query necessary to create a new document matching the given class inside MongoDB. Likewise, when any field of a object whose class is decorated with `@Document` is modified, Mongo-Data automatically runs a query to update the document representation of the object in MongoDB so that the document matches the values of the various fields of the object.

On top of document creation and modification, Mongo-Data makes it possible to perform database query on MongoDB by simply declaring Java functions. Support for MongoDB queries is available only in interfaces that extend the `MongoRepository<T, U>` interface. The actual process implemented by Mongo-Data consists of translating each function name, return type, and parameters into a MongoDB query. For example, the function `public List<CourseEdition> findByCourse(Course course)` is translated to a request to find all `CourseEdition` that match the given `course` field. Similarly, the function `public void deleteByCourse(Course course)` is translated to a query that deletes the given `course`.

Source Code Organization

The source code of the backend is located in directory `src/main/java/app`. Every `.java` file contains a single Java class, interface, or enum.

For the organization of the code in our backend, we decided to follow the names used throughout Spring-Boot and Mongo-Data.

The `controllers` directory contains all classes that are marked using Spring-Boot's `@RestController` decorator. Each of these classes models a route exposed by the API. An explanation of the API is provided in Subsection 3.3.2.

The `requests` directory contains all classes that model HTTP requests coming into the HTTP server. Each controller in the `controllers` directory instructs Spring-Boot on which class from the `requests` directory to use when automatically translating incoming requests.

The `responses` directory contains all classes that model HTTP response bodies coming out the HTTP server. Controllers in the `controllers` directory emit Java objects as responses, and Spring-Boot takes care of automatically converting these Java object to JSON.

Finally, the `model` directory contains all files that model data pertaining to the platform. The names of the files in this directory matches the names of the classes in the model we presented in Section 3.1.

The `model` directory is organized according to the language used in Mongo-Data. This directory contains all classes that are part of our model from Section 3.1. Objects of each class in this directory are accessible to clients through the API.

The `entities` directory contains all classes that make use of the `@Document` decorator. When objects from one of these classes are initialized, Mongo-Data automatically creates a new document in the

[2] https://spring.io/projects/spring-data-mongodb.

database, and when said objects are modified, Mongo-Data automatically updates their document representation in the database.

The `entities` directory contains interfaces that extend the `MongoRepository` interface. When launching the server, Mongo-Data automatically creates a Java object representing a repository that implements each interface described in this directory. These repositories are used for retrieving and deleting documents. Mongo-Data automatically creates queries for each function declared in these interfaces.

The `guidelines` directory contains an interface to be extended in classes that model guidelines, and a concrete implementation of said interface for the CS2013 guideline. The implementation of CS2013 is composed of classes that model knowledge areas, knowledge units, topics, and outcomes, thus preserving the language used by CS2013.

The remaining directories, not described here, contain groups of classes that are closely related to each other. The `contents` directory contains classes to model the contents of a course, the `credits` directory contains the `Credit` interface to model credits and the `ECTS` implementation of the `Credit` interface, and the `time` directory contains classes to model time as pairs of years and semester.

3.3.2 API

The backend exposes an API to allow clients to interact with the underlying model and database. We implemented this API as a simple RPC interface. Each of the requests a client can make to the API can be imagined as traditional function calls that deal with an action on a specific system resource.

There are seven routes exposed by the backend. When a request expects a body, Spring-Boot will try to use said request body to create an object of a specified type dependent on the route. If Spring-Boot fails to create such object, it will throw an error.

The seven available routes are: `/course`, `/courseEdition`, `/studyProgram`, `/studyProgramEdition`, `/faculty`, `/instructor`, and `/guideline`. All of these routes, with the exception of `/faculty` and `/guideline`, expose endpoints to retrieve, create, update, and delete each resource of the type matching the route name. The `/guideline` route only exposes an endpoint to retrieve specific specific guidelines. The `/faculty` route is deprecated and remains only for compatibility reasons with the frontend: we provided a more thorough discussion of why this is the case in Section 5.3.

A complete explanation of each endpoint exposed by the API is offered in Subsubsection 3.3.2.

Explanation

The `/course` route exposes the following endpoints:

- `GET`: retrieve all courses.
- `GET /{id}`: retrieve the course that has the given `id`.
- `POST`: create a new course. The body of this request will be translated to a `CourseRequest` object. The response to this request is a JSON object representing the newly-created course.
- `PUT /{id}`: update the course that has the given `id`. The body of this request will be translated to a `CourseRequest` object.
- `DELETE /{id}`: delete the course that has the given `id`.

The `/courseEdition` route exposes the following endpoints:

- `GET /{id}`: retrieve the course edition that has the given `id`.

- `GET /{id}/dependencies`: retrieve the dependencies of the course edition that has the given `id`. This route is separate from `GET /{id}` because it involves additional computation that might be expensive to run.
- `POST`: create a new course edition. The body of this request will be translated to a `NewCourseEditionRequest` object. The response to this request is a JSON object representing the newly-created course edition.
- `PUT /{id}`: update the course edition that has the given `id`. The body of this request will be translated to a `CourseEditionRequest` object.
- `DELETE /{id}`: delete the course edition that has the given `id`.

The `/studyProgram` route exposes the following endpoints:

- `GET`: retrieve all study programs.
- `GET /{id}`: retrieve the study program that has the given `id`.
- `POST`: create a new study program. The body of this request will be translated to a `StudyProgramRequest` object. The response to this request is a JSON object representing the newly-created study program.
- `PUT /{id}`: update the study program that has the given `id`. The body of this request will be translated to a `StudyProgramRequest` object.
- `DELETE /{id}`: delete the study program that has the given `id`.

The `/studyProgramEdition` route exposes the following endpoints:

- `GET /{id}`: retrieve the study program edition that has the given `id`.
- `POST`: create a new study program edition. The body of this request will be translated to a `NewStudyProgramEditionRequest` object. The response to this request is a JSON object representing the newly-created study program edition.
- `PUT /{id}`: update the study program edition that has the given `id`. The body of this request will be translated to a `StudyProgramEditionRequest` object.
- `DELETE /{id}`: delete the study program edition that has the given `id`.

The `/faculty` route exposes the following endpoints:

- `GET`: retrieve the id of all study programs.

The `/instructor` route exposes the following endpoints:

- `GET`: retrieve all instructors.
- `GET /{id}`: retrieve the instructor that has the given `id`.
- `POST`: create a new instructor. The body of this request will be translated to a `NewInstructorRequest` object. The response to this request is a JSON object representing the newly-created instructor.
- `PUT /{id}`: update the instructor that has the given `id`. The body of this request will be translated to a `InstructorRequest` object.

- `DELETE /{id}`: delete the instructor that has the given `id`.

Finally, the `/guideline` route exposes the following endpoints:

- `GET /{guidelineName}`: retrieve the guideline that has the given `guidelineName`. The only supported guideline for this thesis is `CS2013`.

3.3.3 Frontend

In this subsection, we describe the source code of the frontend that users use to interact with the platform. A walk-through of the user interface of the frontend is available in Section 3.2.

The frontend is a web application. We wrote the frontend of our platform in JavaScript. There exist languages like TypeScript, which is an object-oriented superset of JavaScript, but being unfamiliar with such languages, we preferred to use JavaScript.

React

After some initial considerations, we realized that the frontend could present several complexities. In order to manage such complexity, we decided to rely on React, a framework developed by Facebook for building web application. React allows developers to model elements of a web app as JavaScript objects. JavaScript and HTML object manipulation are performed directly in JavaScript, and rendering is automatically handled by React.

React presents a few concepts that appear in this subsection.

React files use the JSX syntax, which consists of JavaScript with support for embedded HTML. From a practical point of view, JSX makes it possible to work with HTML components directly within JavaScript. Translation from JSX to JavaScript is handled by Webpack, a supporting application that bundles the entire frontend into a single JavaScript file. The output of Webpack is stored in `src/main/static`, the same directory from which the backend serves the frontend to users.

A component is a function that returns an HTML element. This function is expected to be pure, that is: for a given set of inputs, the function must always return the same output. Pure functions make it possible for React to cache rendered components and thus avoid unnecessary duplicate calls to these component-rendering functions.

To keep the number of arguments to these component render functions limited, React offers a facility called "hook". A hook is a block of logic encapsulated within a function. Every time a component is re-rendered, React runs each hook that is used in the component.

Redux and React-Redux

One problem we encountered while working with React is that the application deals with state that is shared across components, for example both instructors and study programs deal with courses. React does not offer conveniences to handle shared state, hence we decided to rely on Redux and React-Redux. Redux is a library designed to manage shared state in JavaScript, and React-Redux is a library that provides bindings between React and Redux. We also chose to use Redux because of its extensive documentation and because we were already familiar with this library.

Redux introduces a set of terms that are found throughout the entire frontend. State is an object that contains all data shared across the application. An action is an object that describes the name of some change that happened, and contains all the data that describes this change. A reducer is a function that given the current state and an action, modifies the state to reflect the change described by the action.

Every time state is supposed to change, Redux requires that this change be described using an action. This description is then dispatched using Redux, and it will be processed by any of the reducers that

recognizes the action dispatched. Redux itself allows multiple reducers to handle a unique action. In our platform, we forced Redux to allow at most one reducer to handle actions in order to avoid conflicts in case two reducers mistakenly process the same action.

A critical aspect of Redux is that state is meant to be immutable. Every time a reducer processes an action, it will return a new object that models the state of the application after the given action has occurred. The React-Redux binding works correctly only if state is never mutated, because this binding relies on simple equality operations to understand which parts of the state have been modified. If reducers did not create new objects for everything mutated in the state every time an action is processed, React-Redux would not be able to understand that state changed, and as a result, it would fail to re-render the application after a reducer has processed an action.

In our frontend, we introduced the concept of "event". Redux actions are meant to be dispatched immediately. There is no built-in support to handle the case where a Redux action is supposed to be dispatched only after some asynchronous action has terminated. To obtain this functionality in Redux, there exist additional libraries, for example redux saga, redux thunk, or others. To avoid relying on too many packages, we preferred to create our own ad-hoc solution. From a technical point of view, this solution is a simple design pattern where asynchronous logic is put inside of a function that, when called, returns a promise. This promise, once resolved, automatically dispatch all relevant Redux actions to update the application's state accordingly to the results of the asynchronous action. In the application, every asynchronous action pertains to communication with the backend via its API. Other types of asynchronous actions that are supported but don't appear in the application include: complex logic ran on a separate web worker (conceptually, a separate thread), timed events like a countdown, or file IO.

Material UI

On top of React, we chose to use Material UI, a library that provided us with helpful abstractions of native HTML elements. When building a web app using Material UI, Material UI provides a default look that we deemed better than that obtained using default HTML elements. We estimated that focusing on the style of the web app could have consumed a significant amount of time, so, for this initial prototype, we preferred to rely on an existing library that provides built-in styling of common HTML elements like paragraphs and buttons.

Other Libraries

For the frontend, we used two other libraries: `react-router-dom`, which provides abstractions to separate independent pages, and `react-google-charts` to display treemaps in the frontend.

Source Code Organization

The codebase of the frontend is located at `src/main/js`. This codebase is broken into directories, one directory for each page in the web app. Each page directory can be found in the `pages` directory. Each page directory contains the source code of a specific page of the application matching the directory name and is organized as follows:

- `index.js`: file that exports of the complete page. At a high level, a page is composed of one or more components located in the `components` directory of the page. These components are laid out together on the page in `index.js`.
- `actions`: directory that contains one file per Redux action that might be generated while the user is in the given page.
- `components`: directory that contains one file per React component that is part of the page.

- `dialogs`: directory that contains one file per Material UI dialog that shows up in the page. Since dialogs are a specific type of component that shows up on top of the page, we decided to separate them from the actual components directory.
- `events`: directory that contains one file for each asynchronous function that might trigger an action.
- `hooks`: directory that contains one file for each custom React hook used in the given page.
- `reducers`: directory that contains every reducer for the actions that can be dispatched in the current page.

For each page directory, in case an item from the list above is missing, it means that the given feature is not present in the page. For example, the guidelines page doesn't have any custom hook, hence there is no `hooks` directory in the `guidelines` directory.

The logic within page directories is not meant to be isolated within each directory. In order to avoid duplicate logic, we decided to allow imports of files that belong to one page into another page. Specifically, when applicable, actions, components, events, hooks, and reducers, might be shared across pages. For example, the `Course` component of the `dependencies` directory relies on the `useCourse` hook of the `Instructor` directory.

The `shared` directory follows the structure of all other page directories, but is intended to contain all those elements which are shared across all pages, such as the logic to display errors or the navigation bar.

The root directory of the frontend, `src/main/js`, contains some additional files that deserve a brief explanation, because they relate strictly to the frameworks we used:

- `index.js`: the entry point of the application. The logic contained in this file creates the React application and binds it to Redux.
- `reducer.js`: groups together all Redux reducers, so that Redux has one, single reducer to work with.
- `Router.jsx`: is the component that contains all pages. It takes care of rendering the correct page based on the current url path, and it also extracts url parameters in case a page is meant to dynamically render data, like different courses or study programs.
- `store.js`: initializes the global state of the frontend. It relies on Redux's `configureStore` method to create the actual store with proper bindings to the reducer that is created in `reducer.js`. The value exported from this file, called `store`, represents the application's inital state managed using Redux.
- `theme.js`: contains theme settings for Material UI. In our application, the only modification to Material UI's defaults is to disable every button's ripple effects, so that when the user moves the mouse on a button, the default bubbly animation applied by Material UI doesn't show up.
- `template`: contains the application's sole html file, index.html, to which the React application is appended inside `index.js`.

In Section 3.4 we explain our setup with regards to developer operations. This includes how we versioned the codebase of our platform, how to build the platform, and how to deploy it.

3.4 Operations

In this section, we present how to build, run locally, and deploy automatically, the platform presented in the current chapter. In this section, we also provide an overview of other aspects related to the code, such as how we versioned it, how we checked for code quality, and more.

3.4.1 Versioning

In order to keep track of the changes to our codebase, we chose to version it. Specifically, we chose to use git. Git is a software for distributed versioning. Git allows developers to create snapshots of a codebase (called "commits"), and organize these snapshots sequentially, such that, starting from the first snapshot, one can visualize the history of changes tracked by the git repository snapshot by snapshot. Git makes it possible to create parallel histories of snapshots (called "branches") starting from a common origin snapshot. These histories can later be merged back together into a single one. Depending on the differences between two merged branches, it might be necessary to resolve said differences manually.

We stored the entire codebase within the same git repository. The branching feature of git allowed us to track multiple parallel versions of the platform. We created a `master` branch that contained only production-ready snapshots of the platform. Simultaneously, we kept a second branch called `dev` which held incremental, partially-working snapshots of the platform. Finally, we implemented new features in separate, temporary branches, each named with the `feature-` prefix. These `feature-` branches were intended to keep new and potentially breaking changes contained from the `dev` branch, and only be merged after those new changes had been extensively checked. This way of organizing a git repository is conventionally known as git-flow[3]. For convenience, we did not strictly follow the `feature-` convention, often opting to implement minor changes directly into `dev`.

We did not make use of the distributed features of git, preferring to keep a single remote repository hosted on our private GitLab instance. We provide an overview of GitLab in Subsection 3.4.3.

3.4.2 Repository Overview

We organized the repository of our codebase as follows:

- `data`: directory that contains mock data to test the platform (more details in Chapter 4).
- `docs`: directory for all documents related to the platform.
- `src`: directory for all source code of the platform.
- `.dockerignore`: Docker-related file that specifies which files should be excluded from the Docker image.
- `.eslintrc.js`: linter configuration for JavaScript.
- `.gitignore`: files to be excluded from the git repository.
- `.gitlab-ci.yml`: pipeline configuration for GitLab.
- `Dockerfile`: a Docker script that specifies how to build a Docker container for our platform.
- `README.md`: markdown file that describes the platform.
- `build.sh`: user-executable shell script to build the application.
- `checkstyle.xml`: linter rules for Java.
- `docker-compose.yml`: a file that instructs docker-compose on how to deploy the platform.
- `mvnw`: maven-related file.
- `mvnw.cmd`: another maven-related file.

[3] https://nvie.com/posts/a-successful-git-branching-model/

- `package-lock.json`: file used by npm to track installed packages.
- `package.json`: configuration file for npm, includes scripts to build, test, and run the frontend.
- `pom.xml`: configuration file for mvn, includes scripts to build, test, and run the backend.
- `todo.md`: file to keep track of to-dos within the repository.
- `webpack.config.babel.js`: configuration for Webpack, the transpiler that converts the frontend's source code from JSX to executable JavaScript code.

Overall, the backend is composed of approximately 2'000 lines of Java code, and the frontend is composed of approximately 5'500 lines of JavaScript code, computed using `cloc`[4].

3.4.3 GitLab

We hosted the git repository for this platform in a private GitLab instance owned by the Software Institute.

We chose to rely on GitLab because of its many features that support project management. Specifically, we chose to use GitLab because it offers an issue tracker and Continuous Integration/Continuous deployment pipeline (later CI/CD pipeline) integrated within the same UI.

To manage issues, our approach was split among two different techniques: we used GitLab for high-level issues, we relied on a markdown file within the repository to track less refined and clear issues, and we added direct to-do notes within the actual code in the case of low-level issues like bug fixes or simple features.

GitLab offers integrated CI/CD pipelines. The purpose of a CI/CD pipeline is twofold: the continuous integration part of the pipeline exists to run a standardized set of commands to build and test code in the repository based on certain conditions, whereas the continuous deployment part of the pipeline is designed to automatically deploy the code in the repository if no problems were found in the continuous integration part of the pipeline.

In our case, we were only able to implement the continuous integration part of the CI/CD pipeline. We were unable to implement continuous deployment due to technical limitations in the local GitLab setup we were using in our server. As a result, we resorted to deploying the platform manually. Our CI pipeline is set up to build both frontend and backend, then run linting checks on both frontend and backend.

3.4.4 Initial Setup

The remaining subsections of this section describe how to build, test, and run the platform presented in this thesis. In order to run the commands presented in the remaining subsections, first, a few programs must be installed. Installation is dependent on the operating system. The programs are: npm, mvn, Java and JDK version 11, and MongoDB version 5.0. To run automated deployment, only docker is required.

- npm is a package manager for JavaScript applications.
- mvn is a package manager for Java applications.
- JDK 11 is the Java Development Kit, version 11. This bundle of applications includes a Java compiler, `javac`, used internally to build the backend.
- MongoDB version 5.0 is a NoSQL, document-based database management system.
- Docker is a suite of programs that make it possible to describe and easily create fixed running environment for software.

[4] https://github.com/AlDanial/cloc.

To install the required programs on MacOS using the Homebrew package manager, run the following commands:

- `brew install npm`
- `brew install mvn`
- `brew install java11`
- `brew install mongodb-community@5.0`
- `brew install -cask docker`

3.4.5 Code Quality

In order to ensure consistency in the writing style of software across the entire codebase of our repository, we decided to set up linters that constantly check whether a set of customized rules are being respected or not. The actual rules we chose are explained in Section 4.4.1.

In the repository, linting is done by two separate programs. For the frontend, we chose to use ESLint. For the backend, we chose to use Checkstyle.

ESLint and Checkstyle are not included in the platform's repository, therefore they must be installed. In order to run the installation commands for ESLint and Checkstyle, make sure that the target system already has all the programs listed in Subsection 3.4.4.

To install ESLint, run the command `npm install` within the root directory of the repository. This command will add an executable version of ESLint in the `node_modules/.bin` directory.

Installation of Checkstyle is dependent on the operating system where the software will be installed. On a MacOS installation, using the Homebrew package manager, Checkstyle can be installed by running the `brew install checkstyle` command.

3.4.6 Building the Platform

This subsection explains how to build the platform. To follow the steps explained in this subsection, it is necessary that the target system already contains the required programs listed in Subsection 3.4.4.

The repository contains a shell script `build.sh` in the root directory. This script runs all the commands required to build the platform on a Linux/Unix system. The script cannot be run on a Windows system. The script is as follows:

```
1. #! /bin/sh
2.
3. mvn package -Dmaven.test.skip=true
4.
5. cd src/main/js/
6.
7. npm ci
8. npm run build:prod
```

Line 1 informs the operating system to use the `sh` program located at `/bin/sh` to run the script.

Line 3 builds the backend using the mvn package manager. The `mvn package` command creates an executable `.jar` file located in the `target` directory. The name of this executable is `curriculum-platform-0.0.1-SNAPSHOT.jar`.

Line 5 changes the current directory of the `sh` interpreter to `src/main/js`, where the frontend source code is located.

Line 7 asks the frontend's package manager to install all required packages for building the frontend.

Finally, Line 8 tells the frontend's package manager to run the script that builds the runnable version of the frontend. In turn, the package manager uses Webpack to build the frontend. Webpack transpiles the frontend's source code to be runnable in the browser and bundles all files into a single file: `bundle.js`. The runnable version of the frontend is stored in `src/main/static`, a common directory from where the backend will serve static content.

3.4.7 Running Tests

We added unit tests to the repository of our platform. An overview of unit tests is provided in Subsection 4.4.2. Overall, we wrote tests for the server using JUnit, and tests for the frontend using Jest. All tests cover various aspects of the code. To test the user interface, we relied on manual testing.

In order to run the server tests, from the root directory of the repository, run the command `mvn test`. To run the tests for the client, from the root directory of the repository, run the command `npm test`.

To run tests, it is necessary to set up a running environment as described in Subsection 3.4.4.

3.4.8 Manual Deployment

This subsection explains how to manually deploy the platform. An explanation of how to set up the running environment is provided in Subsection 3.4.4.

Before deploying the platform, it is necessary to build it. An explanation of how to build the platform is provided in Subsection 3.4.6.

After the platform has been built, it is necessary to launch MongoDB, otherwise the platform is not capable of storing and retrieving data. The procedure to start MongoDB is dependent on the operating system on which the platform is being deployed. On a MacOS installation where MongoDB has been installed using the brew package manager, it is sufficient to run the `brew services stop mongodb-community@5.0` command.

Once MongoDB is up and running, run the `mvn spring-boot:run` command. This command will instruct maven to start the backend of the platform. Once maven has printed the message `Tomcat started on port(s): 8080`, the platform will be reachable at following URL: `localhost:8080`.

3.4.9 Automated Deployment

This subsection explains how to automatically deploy the platform. Conceptually, our automated deployment setup consists of running the steps described in Subsection 3.4.8 within a Docker container.

Given that the platform depends on MongoDB in order to function properly, we chose to use `docker-compose` to manage a multi-container environment. Docker-compose makes it possible to run multiple docker images simultaneously within the same environment, with a shared network and disks. Our docker-compose setup consists of two images: the Docker image for our platform and a Docker image for MongoDB, with two shared volumes. Our configuration instructs docker-compose to build the platform using the included Dockerfile.

Deployment is performed by running `docker-compose build` followed by `docker-compose up`. The first command instructs docker-compose to build all necessary images, and the second command tells docker-compose to set up the running environment and launch all necessary images.

Similarly to manual deployment, after the spring-boot server has printed `Tomcat started on port(s): 8080`, the platform will be reachable at `localhost:8080`.

In Chapter 4, we will provide an overview of how we developed and incrementally assessed the platform presented in the current chapter.

Chapter 4

Development and Evaluation

In this chapter we present the development process we adopted to build the platform presented in Chapter 3, including an explanation of how we incrementally evaluated the platform, keeping track of the features we still had to implement and how evaluated how to improve existing features.

4.1 Development

We planned the development process for the platform presented in this thesis at the start, before working on the actual platform.

During a couple of preliminary meetings, we discussed various features that could have been part of the platform. Creating an initial plan allowed us to estimate what we could have been able to implement within the time available. The development of the platform proceeded as follows.

Throughout the month of December 2020 we started exploring features that could have been interesting to have in the platform, and we discussed a preliminary model of its domain.

For the month of January 2021, having a clearer idea of what features we wanted in the platform, we focused more thoroughly on modeling its domain. During this month, we also set up the overall repository, configuring the GitLab CI pipeline, issue tracker, and docker scripts.

During the month of February 2021, we started to build the backend of the platform. During this month, we decided to hold weekly meetings in order to keep track of the status of the platform.

While building the backend, we started to notice some inconsistencies in names and features of our model, so we decided to build some mock data, guessing that this could have helped in better understanding our model. Therefore, we modeled the 2020 edition of the Master in Software and Data Engineering offered by the Software Institute in Università della Svizzera italiana. For the purposes of testing our platform, we also created a JSON representation of CS2013 that could be ingested by the platform. An explanation of how we prepared this mock data can be found in Section 4.2.

For the month of March 2021, we further finalized the first version of the backend of the platform, complete with data storage and retrieval from the database and an API to interact with the server. During this month we also created a template frontend application, setting up all the necessary boilerplate code for the frontend technologies we picked.

Throughout the months of April and May 2021 we implemented the actual frontend as presented in Section 3.2. During this phase, we went back and forth between working on the frontend and backend, because we discovered some inconsistencies in the API that made it impossible for the frontend to properly interact with the backend. At the end of May 2021, we had a working implementation of the platform for the management of courses, instructors, and exploration of guidelines.

During the months of June and July 2021, we had no further developments.

Finally, in the month of August 2021 we finalized the frontend, implementing the management of study programs. This thesis was also written during the same month.

4.2 Preparing the Data

As mentioned in Section 4.1, in February 2021 we decided to create some sample data to test the platform. Since the platform is focused on study programs and guidelines, we decided to model the Master in Software and Data Engineering offered by Software Institute (later MSDE 2020) and CS2013.

4.2.1 Modeling a Master Program

To model the Master in Software and Data Engineering, we relied on the syllabus of each course offered within said master program during the academic year 2020/2021. From each course syllabus, we extracted all necessary information to build a course object that matched the model we implemented in our backend. After creating objects for each course of this master program, we created an object to represent the study program itself, and objects to represent each of the instructors of the courses in this edition of the master program. We stored these objects inside a JSON object that can be retrieved in the `data` directory of the repository.

Our initial representation of the master program was incompatible with the implementation of our backend. This problem arose because we modeled the master program before having a working implementation of the backend, and we weren't able to spot certain inconsistencies before the backend was ready. As a result, we created the `migration-0001.js` script, retrievable in the `data` directory. This script updates our JSON representation of the master program to a format that can be read by the server.

To ingest the study program, we decided not to alter the existing API exposed by the server. This API allows the user to create and update entities modeled by the platform. Hence, we implemented logic in the frontend that loads our JSON representation of the master program and, if not already existing, creates each course, study program, and instructor described within this JSON. An explanation of why we decided to set up master program ingestion this way is presented in Chapter 5.

4.2.2 Modeling CS2013

To model CS2013, we started from the actual CS2013 document from ACM and IEEE. First, we converted CS2013 to JSON using the process described by Hauswirth in [7].

Our initial JSON representation did not match the model we created for this platform, because said JSON was created before we started working on the platform. Hence, we created a conversion script `cs2013-migration-0001.js` that converts the original JSON representation of CS2013 into a new representation that matches our model of guidelines.

Ingestion of the CS2013 guideline is done automatically by the server. When calling the `GET /guidelines/cs2013` API endpoint, the platform will parse the JSON representation of CS2013.

Guidelines and migration scripts are located in the `src/main/resources/guidelines` directory.

4.3 Assessing UI

As mentioned in Section 4.1, starting from February 2021, we held weekly meetings to take a look at the progress of the platform. During these meetings, we discussed the implementation of existing features, and ideas for upcoming features to be implemented. The purpose of these meetings was to perform an incremental assessment of the platform.

Each meeting lasted at most an hour, and it was organized as follows. First, we went through the latest changes to the platform by doing a demonstration of them. Then, we discussed these new changes, and any potential issue that might have arose since the previous meeting. If time allowed for it, or if necessary, we also discussed ideas on upcoming features in the platform. Except for this overall format, meetings were held in an unstructured way, where each member provided feedback when they best saw fit.

Given physical restrictions caused by the health situation during the spring of 2021, we held our weekly meetings online in a videoconference using Microsoft Teams.

Meetings were not the only setting in which we tested the user interface of the platform. Throughout the entire development phase, we incrementally tested the user interface every time features were introduced or modified in the platform. These assessments didn't follow any standardized procedure.

4.4 Assessing Code

In Section 4.3 we presented how we assessed the user interface of our platform. In this section, we present how we assessed the code of our platform.

Our methodology to assess code was split into three different tasks. The first one doesn't deserve a subsection of its own: it was the most frequent, and it consisted of simple self-review sessions. Each day, before starting to work on the code, we reviewed all changes from the previous day to check whether they contained any potential mistake, code smell, or other types of problem.

In Subsection 4.4.1 we explain the code style checkers we set up to enforce a common style across the entire backend and frontend, and in Subsection 4.4.2 we explain our setup for unit testing.

4.4.1 Style Rules

To enforce style rules, we set up two linters: ESLint for the frontend, and Checkstyle for the backend. An explained of how to install these two linters can be found in Subsection 3.4.5.

For the frontend, we configured ESLint to apply the style rules recommended by AirBNB using the `airbnb-base` package. On top of that, we also followed the default ESLint rules using the `eslint:recommended` package and React rules using the `plugin:react/recommended` rules. We also relied on the `plugin:react-hooks/recommended` package for rules pertaining specifically to hooks: a recent feature of React that isn't covered by the rules of `plugin:react/recommended`.

Since the platform's UI is written in JavaScript using React as the underlying UI framework, we configured ESLint to also check the style of `.jsx` files using the 2018 version of JavaScript.

For the backend, we configured Checkstyle to use the included Google style rules. The only modification we applied to this set of rules is to allow lines of code of length up to 150 characters. We made this modification because some types and methods have long names and the default line length of 100 character caused Checkstyle to require many unnecessary line breaks that reduced code readability.

4.4.2 Unit Tests

In order to assess certain aspects of the platform, we devised various unit tests that. These tests focused on the API endpoints of the platform and the flow of data in the frontend.

To run tests, it's necessary to set up a running environment as explained in Section 3.4. Each test is designed to assess a specific feature of the platform. Unit tests are stored in separate directories for the server and the client. We used JUnit to test the server and Jest to test the client. In each unit test, either the server or the client is run. No client test requires the frontend to be running.

In Chapter 5 we will offer a discussion of all the aspects of our platform presented in this thesis.

Chapter 5

Discussion

In this chapter, we offer a discussion of all the results, decisions, and contributions presented throughout this thesis. Specifically, in Section 5.1, 5.2, 5.3, and 5.4 we discuss the platform presented in Chapter 3, and in Section 5.5 we discuss our development process and evaluation methods presented in Chapter 4.

5.1 Discussion of Model

In Section 3.1 we presented the model of our platform. We built this model first, before writing code for the actual platform. We believe that starting by modeling was the right decision, because this model guided us during the development of the actual platform. As a result, the terminology and attributes of the model appear thoroughly throughout the entire codebase of the platform.

5.1.1 Separation of StudyProgram and StudyProgramEdition

One challenging aspect of the model is the distinction between `StudyProgram` and `StudyProgramEdition`. While modeling `StudyProgram`, we realized that, in its current form, the model didn't make it possible to track multiple editions of a study program. For this reason, we chose to create a new entity called `StudyProgramEdition`. The study program describes a conceptual study program, whereas the edition of a study program represents an actual study program to which students can enroll. Since every edition is part of the same study program, `StudyProgramEditions` are grouped together in a `StudyProgram`, which acts as a container of `StudyProgramEditions`. Starting from a `StudyProgram`, it's possible to retrieve all its editions, and from any `StudyProgramEdition` it's possible to retrieve their `StudyProgram`. The same type of reasoning applies to `Course` and `CourseEdition`.

This distinction between study programs and study program edition, just like the distinction between course and course edition, turned out difficult to follow. During our weekly meetings, we confused the two concepts several times.

In hindsight, we believe that a better way to track multiple editions of a study program or course would have been to add two properties `previousEdition` and `nextEdition` to each study program or course, thus creating a linked list of study programs or courses. Then, given any study program or course, it is possible to retrieve all editions by traversing the list. With this modification, there would no longer be a need to separate study programs and courses from study program editions and course editions, thus simplifying the model. Implementation of such a modification would require a thorough refactoring of the entire codebase, since the separation between study program and study program edition, just like the separation between course and course edition, is found throughout the entire backend and frontend. The UI would also require some degree of redesign, because, as presented in this thesis, there is a page for study programs and a separate one for study program editions, just like there is a page for courses and a separate one for course editions.

5.1.2 Guidelines

We are still unclear as to whether it was a good decision to allow developers to implement new guidelines as they best see fit. The alternative we considered is to create a common language for describing guidelines, and when a new guideline is added to the platform, its contents would need to be translated to this internal representation rather than allowing developers to model this new guideline as best they see fit. The disadvantage of this modular design is that developers would have the extra cost of also implementing a UI to use new guidelines. Also, it becomes more difficult to make comparisons between courses that use different guidelines, because the concepts of one guideline might not match the other guidelines depending on how each guideline was implemented. These problems would be avoided by using a single common model to which all guidelines are required to adhere. However, forcing a single model means that it could be impossible to implement certain guidelines. For example, if the model requires a hierarchy of knowledge areas made of knowledge units made of topics and outcomes, it would be impossible to implement a guideline that offers plain topics and outcomes without any order among them.

5.1.3 Instructor

Another problem of the model relates to instructors. As presented in this thesis, an instructor is described by their first name, last name, email, and taught courses. One idea we discussed during our weekly meetings is to track an instructor's skills as they relate to a guideline, so that the platform could then recommend substitute instructors for a given course given the course's requirements and each instructors' skills.

During our meetings we also identified that there might be multiple types of instructors. For example, an instructor might be a professor or a teaching assistant. This type of differentiation is, for now, not modeled. If the system introduced such distinction, it could be possible to build a recommendation system where, given a set of courses, the platform automatically picks the best teaching assistants for each course. If the model of courses is extended to track individual lectures, the platform could attempt to ensure that there are no scheduling conflicts for teaching assistants and courses.

5.1.4 Modeling Learning Resources

Our initial plan for this thesis included the development of a database of tagged learning resources, meaning books, videos, and any other type of learning material that instructors can recommend to their students. This database would allow users to catalog any new or existing learning resource and mark the contents of each learning resources in terms of mapping to an underlying guideline. With a database of tagged learning resources, the platform could provide recommendations on which study materials, among the known ones, best fit the outcomes of a course or study program.

5.1.5 Modeling Students

During our meetings we also discussed the idea of modeling students, but decided not to focus on it because we estimated it would not fit within the time available for the thesis. When modeling students, it could be interesting to track student performance and student skills as they relate to an underlying guideline. This type of information would allow the platform to provide recommendations for additional courses and study materials to students based on their skills.

Furthermore, students could provide feedback to courses and study programs directly through the platform, either as text or some type of metric. In the case of metrics, these could be aggregated to provide additional measures for both courses and study programs. A more thorough discussion of metrics is provided in Subsection 5.2.5.

5.1.6 Course Edition

We believe that the model for course editions can be more granular. During our meetings we noted that a course edition is made of lectures, students receive assignments, and it might also contain some exams. This type of information is not modeled for now because it is not needed, but it might be useful, for example, to track student data during each lecture, or to track student performance on each assignment as mentioned in Subsection 5.1.5.

5.2 Discussion of User Interface

In Section 3.2 we presented the user interface of our platform by offering a walk-through of the various operations that can be performed using said user interface. This user interface allows users to view and manage study programs, courses, and instructors. Furthermore, this user interface also allows users to explore a chosen guideline.

5.2.1 Unnecessary Complexity

During our weekly meetings, we realized that the user interface needs thorough improvements. The overall problem we identified is that in order to perform any action, the user needs to step through many pages and menus.

The main page of the web app is cumbersome. It doesn't provide any meaningful information to the user, and just acts as a navigation page to redirect the user to any of the four main sections of the frontend: instructors, courses, study programs, and guidelines. This page requires the user to read each of the four navigation buttons presented in order for them to figure out which button will lead them to the page they are looking for.

In the course edition contents page for CS2013, it's complex to navigate the tree of knowledge areas, knowledge units, and topics and outcomes. Furthermore, it's difficult to specify whether a topic or learning outcome is required, good to have/know, or not required. The user is forced to navigate through this tree structure and select, for each topic and outcome, the appropriate requirement level from a menu. To ease this task, the options available in the menu could be embedded in the tree itself, so the user is not required to perform an additional click to open the menu every time. Considering that CS2013 contains 2′222 topics and outcomes, the effort saved by freeing the user from the need of performing 2′222 extra clicks could be significant.

Navigation and understanding of the treemap we used to visualize guidelines is unnecessarily complex. These problems arises from how the treemap was implemented by the maintainers of the charting library we used. The two problems we identified are the following: it's unintuitive that the tiles of the treemap can be clicked to dive one level deeper in the underlying guideline, it's unintuitive that a right click is necessary to go back one level in the underlying guideline, and finally, the coloring logic is unclear, because the library colors tiles based on the relative highest and smallest value visible, rather than the overall highest and smallest values available in the entire dataset displayed. Our intent with the coloring of treemaps was similar to that of Jafar et Al., who presented heatmaps whose colors highlighted the relation between knowledge areas in CS2013.

Course dependencies are presented as a list. This view is not easy to navigate through. Since graph dependencies form a sparse tree, we believe that a graph view of dependencies could be more intuitive to use. Such a visualization was implemented, for example, by Auvinen et Al.

5.2.2 Management Process

During our meetings, we discussed whether to create a specific process for the management of study programs.

In the literature, Ajanovski implemented an iterative process where professors progressively come to a decision on which courses should be in a study program. We chose not to follow this approach because we estimated that this degree of rigidity would restrict users too much.

Likewise, Herbert et Al. implemented a management process that involves interviews with professors and industry experts to better assess the contents of a study program. We chose not to follow this approach too, because we did not want to introduce the concept of interviews in the workflow.

Finally, we found another management process presented by Gorman et Al. We chose not to implement this process because it is tailored for the creation of ad-hoc training programs for industry, whereas our platform is focused on universities.

As a result, the workflow for the management of study programs in our platform is highly permissive. Users of the platform can decide to adapt more involved processes like those presented by Ajanovski, Herbert et Al., or Gorman et Al. The only requirement imposed by our platform for the creation and management of study programs is that, when specifying the courses of a study programs, said courses must already exist in the platform. Hence, instructors are required to create courses before they can be added to any study program. However, there is no requirement that all courses be implemented before the creation or modification of a study program, so a study program can be created in parallel to its courses. Furthermore, courses need only to exist in order to be added to a study program: the contents of a course can be added later.

5.2.3 Style Issues

The overall styling of the application is not that intuitive. We relied on Material UI to handle most of the styling, but we think that it's not sufficient. The web app's style is not sufficiently responsive, causing certain elements, like card data for instructors, to become invisible at smaller display sizes. The color scheme doesn't provide any meaningful information to the user. Another issue with the UI is that the card layout is not scalable: as the number of instructors, courses, and study programs grows, users become more and more dependent on the search feature, because traversing the entire view of cards becomes challenging.

The core mistake we identified in designing the frontend of our platform is that we built it step by step, without having a complete view of what the end product should have looked like. As a result, this design can be summarized as a patchwork of features that were stitched together in order to display all relevant information.

Overall, we believe that the frontend of our platform needs a thorough redesign that takes more consideration of the user and presents information in a more intuitive and accessible way.

5.2.4 Lack Of User Identity

There is no concept of user identity. Hence, the platform makes no distinction between its users. During our meetings, we identified that there are different types of users who could use this platform: instructors who teach and manage courses and other instructors who manage study programs. If the platform is expanded to support other ideas, like tracking learning resources or students, then there will be more user types, such as those who manage learning resources and students.

If user profiles are introduced in the model, there will be a need to present said user profiles to the users in the frontend. We discuss learning resources and students in Section 5.1.

5.2.5 Metrics

Courses and study programs can be evaluated by looking at how they cover the chosen guideline using treemaps. This view can provide interesting insights, like highlighting knowledge areas that are more predominantly covered by the selected course or study program.

We did not implement any of the metrics found in the literature due to time constraints. Our original plan included computing and displaying the metric proposed by Krishna et Al., where instructors specify the contents of their courses according to a guideline and, in the case of CS2013, the weights associated to each content by the guideline are summed together to produce a value that represents the total number of core 1 and core 2 hours associated to a course.

During our meetings, we considered all other metrics we found in literature, but we preferred to avoid them because their computations are more involved. Specifically, the metric proposed by Camilloni et Al. requires students to track how much time they spend on each course outside of lecture hours, the metrics proposed by Nordstrom et Al. require modeling and tracking students and their performance, and finally, the metrics presented by Rowe et Al. rely on a subjective scoring of the chosen guideline.

5.3 Discussion of Architecture

In Section 3.3 we presented the backend, API, and frontend of the platform: in this section, we discuss these three aspects of the architecture we designed. Specifically, in Subsection 5.3.1 we discuss the backend, in Subsection 5.3.2 we discuss the API, and in Subsection 5.3.3 we discuss the frontend.

5.3.1 Backend

We implemented the backend of our platform in Java. For the backend, we relied on the Spring, Spring-Boot, and Mongo-Data libraries.

Libraries and Technologies

We faced some technical challenges in learning how to use Mongo-Data and get a better grasp of how Spring works. These challenges slowed us down by approximately two weeks, time during which we weren't able to make any progress. Still, despite these challenges, we believe that picking these libraries was the right choice, because they provided us with various meaningful abstractions that made implementing the backend relatively easy.

While developing the backend, we realized that MongoDB might not have been the right choice for the database. While it's true that our data is more easily modeled using the document metaphor rather than normalizing it and storing it in traditional SQL tables, we don't have any concrete reason to conclude that MongoDB was the right choice.

File Structure Organization

From the point of view of file organization, storing files in separate directories depending on their contents made it easier to navigate the repository. For example, all controllers are within the same directory and all classes that deal with the model are in a separate directory.

We find that the names we chose for some classes are too long. For example, the class `NewStudyProgramEditionRequest` is 29 characters long. Such names are cumbersome to deal with. We believe that during a refactoring of the server, it is advisable to find shorter names for such classes.

5.3.2 API

The server of our platform exposes an API. We built this API one route at a time, while developing the frontend, extending it when needed. An example of this can be seen in the `/faculty` route, which exposes the deprecated concept of "faculty": this concept was presented in early iterations of our model until we deprecated it, and it remained in the API.

When interacting with the API, users can perform direct actions on specific resources of the platform. For example, users can retrieve, create, update, and delete courses, course editions, study programs, study program editions, and instructors.

Leaking Implementation Details

In hindsight, we believe that our lack of design for the API resulted in a poor outcome. This API is implemented as an RPC interface. Hence, implementation details are leaked from the server to the client. For example, when retrieving a study program, clients have to worry about then retrieving any necessary study program edition.

Missing Access Control

The API lacks any form of access control. Everyone can call every endpoint exposed by the API, including destructive endpoints like `PUT`, which overwrites a resource, and `DELETE`, which removes a resource.

Missing Bulk Import/Export

Data is stored in a structured way inside the database, but the API offers no endpoint to export it in a structured way, and likewise, the API offers no endpoint to bulk-ingest data. When importing our representation of MSDE 2020, the frontend is in charge of creating a new resource for each course, instructor, and study program defined within our representation.

Limited Granularity

Granularity in the API is down to the object. Depending on the type of property of a given object, it could be useful to expose endpoints for the retrieval and manipulation of specific properties of said object. Currently, the only example of this increased granularity can be found in the `GET /courseEdition/id/dependencies` route, which computes the dependencies of a given course edition.

5.3.3 Frontend

To allow users to interact with our platform, we built a web application in JavaScript using React, Redux, and other JavaScript libraries.

React

We believe React was a good choice for managing the complexity of the platform. React allowed us to model pages and components as functions. Using no framework would have required us to re-implement many features that are already present in React, for example the automatic re-rendering of components when their contents change.

We considered the option of serving static pages. This option would have freed us from the need of creating a web application. We decided not to choose this option because creating static pages would have prevented us from modeling the various parts of the frontend as JavaScript functions. Furthermore, we estimated that in the long term, upcoming features could require the frontend to be dynamic, for example automatic updates and user sessions, hence we preferred to start by creating a web application that is dynamic since inception.

There exist other frameworks for managing web applications, like Angular or Vue. We decided to use React because, being already familiar with React, we estimated it would save us time both in terms of development and debugging.

Redux

Another decision that helped us immensely was to set up shared state using Redux. React doesn't have a concept of shared state, and to achieve such functionality in plain React, it is necessary to either create and track duplicated data across components, which is a bad option because if changes to a duplicate piece of data are not reflected properly across all copies, it might result in inconsistencies. The other alternative to achieve shared state in React is to hoist shared data up into an intermediate object designed entirely to store shared data, then pass this data down to each component through props. This second option is bad too, because it requires developers to introduce additional elements in the page that have no real purpose visually, and rather just serve as data containers. Redux prevented us from having to rely on these two bad practices, offering an easy, albeit verbose, way to share information between components in React.

Since the UI is driven by user interaction, we found it useful to model each interaction as a pair of Redux actions, Redux reducer, and when dealing with asynchronous actions, also events.

We bound Redux's state to React using the React-Redux library. Redux state needs to be treated as immutable, otherwise React-Redux might be unable to notice parts of state that changed and thus fail to refresh React components. One option to enforce state immutability is to use immutable.js, a library for state immutability. We considered using immutable.js but ultimately chose not to, because immutable.js makes regular JSON objects forbidden in the state and as a result, the classes of immutable.js would spread throughout the codebase everywhere state is used, and this would make future refactorings harder to perform for no significant good reason. Furthermore, objects created using the classes exposed by immutable.js cannot be accessed using property accessors (square brackets). Finally, if we chose to use immutable.js, a new developer picking up the codebase of our platform would be required to learn one extra library, which we deemed to be not worth the benefit of having guaranteed immutability for the state of our frontend.

Originally, we planned to have one file per reducer in our Redux setup. However, due to time constraints, we weren't able to ascertain whether this would have been a better option than keeping all reducer functions within a single file. As a result, each reducer directory contains a single file `reducer.js` that holds a map of action type to reducer function for each action of each page. If one file per reducer is found to be a better option than having all reducers in one file, the map structure within `reducer.js` must be preserved, while each individual function can be moved to a separate file, then be imported in `reducer.js` and added back to the map of actions to reducers.

Material UI

We chose Material UI to have a default style for the various HTML elements we used, such as paragraphs and buttons. We also picked this library, like many other ones, because of familiarity. However, while building the frontend, we realized that Material UI pushes the programmer to style user interfaces in a very specific, "material" way. Customizing the UI was at times difficult and required us to fight against the framework. As a result, several parts of the UI look unpolished. Search bars are too tall, buttons in the main menu are not centered, cards have uneven spacing between components, the list of visual defects goes on. In order to solve these problems, rather than continue to deal with Material UI, we believe it could be easier to refactor the user interface using a different, more permissive library that doesn't force users to follow a certain visual style.

Charts

Google's library for charting turned out to be too difficult to use. Documentation for this library was severely lacking, especially for the React version. As such, it was not easy to understand how to use it, and we had to resort to trial and error until it seemed to produce the right results. Still, it was the only implementation that offered charts close enough to what we were looking for.

In a future revision of this platform, we believe it's best to implement a custom library to display treemaps. Such library might be built using HTML's canvas for rendering charts. The two main challenges we encountered with the Google implementation of treemaps are cumbersome navigation and lack of configuration for the logic used to color tiles.

File Structure Organization

From the point of view of the frontend's codebase, code is organized according to the page in which said code is used. This distinction looked reasonable at first, but as the complexity of the frontend grew, we realized that there were much more files that contained logic shared across pages than we expected. Hence, in hindsight, we believe it would have been better to have all files grouped together by type, so that all actions are in a single directory regardless of which page they are used in, events are all in the same directory, and so on. The only type of file that we believe can reasonably be grouped by page is those containing components, because in the frontend, the only component that is shared across all pages is the navigation bar.

Other Aspects

Another issue we encountered when developing the frontend is that, at times, the lack of types in JavaScript made it harder to track which objects were being passed around between functions. Such a problem could have been solved by using TypeScript, a strongly-typed superset of JavaScript. We chose not to use TypeScript because we are unfamiliar with it, and we estimated that using a new language to build the frontend would have slowed us down too much, making both development and debugging harder.

5.4 Discussion of Operations

In order to facilitate development of the platform, in Section 3.4 we presented the tools and scripts we set up to automate certain tasks of our codebase. We used git for versioning. We used GitLab for hosting the repository remotely, track requirements, and automatically run tests. We also built shell and Docker scripts to automate the tasks of building and deploying the platform.

5.4.1 Versioning

Versioning our codebase from the start allowed us to keep track of how the project evolved. Versioning was also helpful to retrieve past versions of the platform when we decided to roll back some changes to the model and frontend.

As mentioned in Section 3.4, in our repository we used a branching strategy with the following rules: release-ready versions of the repository are kept in the `master` branch, working but not release-ready versions are kept in `dev`, and every feature that breaks the platform is developed on a separate `feature` branch. We didn't strictly adhere to this way of organizing versions of our code, opting to sometimes keep broken versions of the platform in the `dev` branch. We chose not to be rigid in our management of the repository in order to avoid development hiccups, getting stuck on a specific feature and slowing development of others.

Storing the entire codebase within the same git repository made it easier to track changes across both the frontend and the backend. When refactoring both the frontend and the backend, we were able to keep track of all changes within a single commit. Had we chosen to use separate repositories for the backend and frontend, we would have needed to either devise methods to preserve the connection of related commits between the two repositories, or just renounce to tracking this information, so we believe that using a single repository was the better option in this case.

5.4.2 GitLab

We originally planned to use only GitLab to track features and issues, because it provides a comment section for each feature and issue, thus enabling asynchronous discussions compared to the synchronous approach of our weekly meetings. In practice, we didn't rely much on GitLab's issue tracker, opting to keep most of the features and issues in a file within the repository and discuss each point at once during our meetings.

We identified two problems with our approach: first, modifications to the file require updates to the repository, so our repository contains additional commits where only this file tracking features and issues is modified, and second, since the file contained only things left to do, we didn't track the list of things done. In contrast, GitLab's issue tracker does not require changes to be committed to the repository, and completed features and issues are automatically archived rather than lost.

We didn't use GitLab's CI pipeline in practice. We found that building the platform and running tests remotely took longer than running them locally, so we stuck to building the platform and running tests locally. This task was further facilitated by our support scripts, which made it possible to build and run the entire platform using a single command.

5.4.3 Scripts, Docker, and Other Helpful Programs

To automate the task of building our platform, we created the `build.sh` script, and to automate deployment of our platform, we set up a Docker image and a docker-compose script.

We relied extensively on the build script, because the steps to build our platform remained constant throughout its entire development. In contrast, we didn't rely much on the Docker and docker-compose script, mostly because manual deployment requires only a single command (`mvn spring-boot:run`) and in our experience, it was faster than launching multiple Docker containers.

Finally, package managers were helpful in tracking the dependencies of our backend and frontend.

5.5 Discussion of Development and Evaluation

In Chapter 4 we described how we built and evaluated our platform.

5.5.1 Development

We developed our platform over the span of seven months. We encountered several challenges during development.

Excessive Iteration Time

Several of our estimates for the time required to complete a task were wrong. For example, we underestimated the time required to set up Mongo-Data by approximately two weeks. We also underestimated the time required to set up all the tools and scripts presented in Section 3.4.

Similar to wrong time estimates, we failed to properly evaluate the priority of several tasks. For example, we prioritized setting up the tools and scripts from Section 3.4 at the start of this thesis, and delayed the first prototype of our platform by a couple of weeks.

Overall, we found that our iteration time was too long. We spent excessive amounts of time focusing on smaller details of the task at hand, rather than building a rough prototype first, and fixing details later. Likewise, related to this issue, we put too much attention on preemptive coding early on. We built the entire backend first, then proceeded to build the frontend. As a result, we had no concrete way to ascertain which parts of the backend were actually necessary and which were not. For example, the concept of

"faculty" mentioned in Subsection 5.3.2 became deprecated after we realized, while building the frontend, that it was not necessary.

Lack of Direction

Throughout the entire creation of our platform, we failed to develop a clear vision of what the end product should have looked like. We spent dozens of hours during our meetings evaluating various features and ideas, and we managed to hone in on certain aspects of the platform. This was not sufficient. Many questions remained open, and we explored them too late, while we were in the process of implementing them. As a result, we ended up wasting time to implement features we weren't really sure we needed. For example, an initial iteration of the frontend required the user to open a dialog in order to edit each field of each object (instructors, courses, study programs). Development of those dialog boxes required about a week of time. After the feature was implemented, we realized that it would have been easier to allow the user to directly edit each object property directly from within the main page, transforming labels into editable text fields. As a result, the entire week spent developing dialogs to edit object properties went to waste. Such waste could have been avoided with more thorough planning using less expensive methods, like hand or computer-drawn sketches. Combined with an overall lack of ownership of the project and low sense of urgency, a considerable amount of time ended up being wasted.

5.5.2 Evaluation

We evaluated our platform in three ways: weekly meetings, code linters, and unit tests.

Weekly Meetings

Our primary way to evaluate the platform was to review it during a period meeting held weekly. These meetings were the source of many insights and correction that allowed us to better understand what features to include in the platform and how to develop them.

We faced two issues with our meetings. First, as a group of three, scheduling meetings where everyone was able to attend turned out to be difficult, primarily because of schedule conflicts. Second, our model presents certain abstract concepts, for example the separation between study program and study program edition. As a result, we found it challenging at times to convey ideas and talk about certain abstract aspects of the platform.

Linters

In order to enforce a consistent coding style across the entire platform, we set up two linters: ESLint for the frontend and Checkstyle for the backend. The linter for the frontend turned out to be critically helpful, because it provided many insights on code issues that sometimes are not easy to stop with a naked eye. For example, ESLint automatically raises a warning when a declaration is not used or when an undefined declaration is used. These types of support were less necessary in the backend since it's written in Java, and therefore, those types of error show up at compilation time.

Unit Testing

We built unit tests to assess certain the backend's API and the Redux's reducers. The problem of tests is that we wrote them too late, and we didn't write enough of them. For example, the backend's model is only tested indirectly, through our tests of the API, and the functions that create React components are not tested at all, because we didn't have time to explore the best practices on testing React user interfaces programmatically.

Test Data

To help us better test the application, we built a data representation of CS2013 and MSDE 2020. Encoding CS2013 and MSDE 2020 into two separate files made it possible to test the application in a realistic way. Still, the platform has not been tested in a real-world scenario. Such a test should be performed, because it could highlight problems that we were unable to identify during our weekly meetings and internal testing.

In Chapter **??** we will present a series of ideas on how to further develop and expand the platform we presented in this thesis, providing, where available, insights and other aspects that we identified during our weekly meetings.

Chapter 6

Conclusions

In this thesis, we presented a novel platform for the management and evaluation of courses and study programs. This platform links courses and study programs to an underlying guideline, makes it possible to manage instructors, and it allows users to explore an this underlying guideline.

In Chapter 1 we presented several issues pertaining to the creation and management of study programs. For example, there are documents designed to help those who create and revise study programs, but the link between these documents and the study programs they help create is unclear. Furthermore, there is no defined way to manage and evaluate study programs.

We found existing approaches to deal with the aforementioned problems in the literature as discussed in Chapter 2. Such approaches include that of Ajanovski, who built a platform for the management and evaluation of study programs bound to an underlying guideline, and Auvinen et Al., who implemented a platform for the creation of study programs that isn't bound to an underlying guideline.

For this thesis, we decided to develop a novel platform for the creation and management of study programs bound to an underlying guideline, where courses are also bound to the underlying guideline. This platform also supports the management of instructors, it provides a visual evaluation of study programs, and it allows users to explore the chosen guideline. We presented the model, user interface, and various code aspects of this platform in Chapter 3.

We developed this platform over the span of six months. The platform is a client-server application. The client is a React application built in JavaScript, whereas the server is a Spring-Boot application written in Java.

To progressively assess the platform, we held weekly meetings aimed at evaluating the implementation of existing features and discussion of ideas for the development of upcoming features. Furthermore, we relied on linters and unit testing to assess the quality of the codebase of our platform. We presented our development and assessment of the platform in Chapter 4.

In Chapter 5 we discussed all the aspects presented in this thesis. Overall, we succeeded in creating a proof-of-concept platform that links study programs and courses to an underlying guideline; allows users to create and manage study programs, courses, and instructors; and makes it possible to explore a chosen guideline. There are, however, many ideas left to implement, such as the tracking of study materials, access control, and more ideas that we discussed in Section 6.1.

6.1 Future Work

In this section we present various ideas on ways to further develop the platform, including improvements to existing features and new features that can be added to improve the platform. Some of ideas presented in this chapter were intended to be in the final version of the platform presented, but due to time constraints, we were unable to implement them. When applicable, we provide insights that emerged during our weekly meetings on the idea being presented.

6.1.1 Learning Resources

Our original vision for the platform we developed for this thesis included a tagged database of learning resources. Learning resources are any type of support material that instructors can use in their courses, such as books and videos.

Our idea was to give users the possibility of registering new entries in this database, and to maintain a description of how each entry mapped to the chosen underlying guideline. For example, in the case of CS2013, the idea was to allow users to specify all the topics covered by each learning resource.

The purpose of this database of learning resources is twofold. On one side, this database would be a common location from which to retrieve and discover learning resources based on a user-customisable list of topics of interest. On the other side, this database could be used by the backend to provide study material recommendations given the topics covered in a course.

Addition of learning resources would require a modification to the model of courses: courses would have an additional field `learningResources` that consists of an array of `learningResource`. The model for `learningResource` is to be defined.

Management of the learning resources database would be done through the same UI that users use for all other operations pertaining to the platform. Similarly to instructors, courses, and study programs, the UI for learning resources would consist of a page to create new learning resources, a page to explore existing learning resources, and a page to modify existing learning resources. Furthermore, the page for managing an existing course would include an additional list showing the study materials related to the course and a new page or dialog that shows to the user recommendations about which learning resources best match the topics of the chosen course.

6.1.2 Students

During our initial meetings, when discussing which features to focus on, before creating the actual model, we considered the option of modeling students.

The system currently used at Università della Svizzera italiana asks students to voluntarily provide feedback about each course taken. Such student feedback could be gathered directly from the platform, so that each student, either anonymously or not anonymously, could have the possibility to provide direct feedback about courses. This feedback could either be free text submissions, some numeric metrics, questionnaires, a combination of these, or more.

If the model is extended to include items such as assignments and exams (see Section 6.1.3), it could be possible to track the performance of a student throughout the study program and make statistics about each course and study program in terms of student performance.

Students could rely on the platform to have a clearer overview of courses and to review their progress, for example by seeing their assignment grades and exam results.

In the context of CS2013, the model of a student could be extended to track the performance of students on a per-knowledge-unit basis. Then, the platform could be extended to provide custom recommendations on additional study materials for each student when performance on a specific knowledge unit is below a certain threshold.

Another idea with the concept of students is to allow students to specify which learning outcomes they would like to reach, and based on these target learning outcomes, the platform would recommend a customized study program that could lead the student to reach the specified learning outcomes. Such an idea is described in CS2013 [10, p.41]. Auvinen et Al. developed a similar system.

6.1.3 Courses

The model for course editions could be made more granular by also modeling actual lectures, assignments, exams, and other activities and entities related to courses. This extension to the model would coincide strongly with the modeling of students described in Section 6.1.2.

These additional models (lecture, assignment, exam, and other activities) could follow the same separation used for courses and study programs, where `Lecture` tracks the concept of lecture, whereas the `LectureEdition` is a specific edition of the lecture that was taught at a specific date. Similarly, an exam could model the concept of an `Exam`, whereas `ExamEdition` could model a specific version of an exam that was held at a specific date.

Course contents must be filled in by an instructor. In a future update of the platform, course contents could be extracted automatically by the course's syllabus. For example, the instructor could upload their course's syllabus and the platform could attempt to extract course information from the document using some text analysis methodology. Such an approach was presented by Sekyia et Al.

6.1.4 Study Programs

A useful feature to track the changes of a study program over its various editions could be to provide some view that highlights changes between two editions of a study program. This same feature could be implemented to compare editions of a course and any other future entity for which multiple editions might exist.

Depending on the desired goals when expanding our platform, the current visual representation of study programs and courses within a given guideline could be supplemented with metrics as described in Chapter 2 and discussed in Subsection 5.2.5.

During our meetings, we decided to configure the application such that instructors create courses, and then other users group these courses together to form study programs. We discussed the idea of allowing the users that create study programs to specify the set of desired learning outcomes for a given study program. This feature would make it possible to compare the desired goals of a study program with the coverage of the actual courses that are in the study program. If the desired learning outcomes of a study program are tracked, it is also possible to provide recommendations on which courses could be added to close the gap between desired learning outcomes and actual learning outcomes. From the point of view of CS2013, such a recommendation system could also suggest new courses by relying on its knowledge area/knowledge unit/topic/outcome hierarchical structure.

The platform currently considers a study program as the set of courses a student can take for a given academic year. The platform could be expanded to also track the actual courses that a student will take, spanning across multiple years.

Study programs could be extended to track feedback from selected users. This type of feedback was used by Ajanovski, and also by Herbert et Al. for evaluating study programs. Feedback could be displayed as a set of threaded discussions, where each user has the ability to write new comments or reply to existing comments by any other user.

6.1.5 Guidelines

For the purposing of testing the platform, we modeled only one guideline: CS2013. Since CS2013 is not the only guideline professors can rely upon when creating or modifying study programs, a future version of the platform could support multiple guidelines, such as any previous or future revision of the Computer Science Curricula from ACM and IEEE, or other, specific guidelines, like GSwE2009 or IT2008.

6.1.6 Evaluation

Study program and course metrics dependent on a guideline could be presented in a summary or report. This report could also offer multiple visualizations for various aspects of courses and study programs. For example, a report for a study program might present a directed, sparse graph of courses within the study program, where each node in the graph is a course and each directed edge shows that the course from which the edge originates depends on the course to which the edge points to. The graph would be sparse because it is not guaranteed that every course is dependent on others. Such a graph was displayed in the platform created by Auvinen et Al.

Computation of course dependencies is already implemented in the platform, both in terms of explicit dependencies stated by the users, and implicit dependencies that are computed from each course's requirements and outcomes. Since these metrics and visualizations are dependent on the chosen guideline, it is possible that the type of metrics and visualizations offered in the report depend on the chosen guideline.

Another guideline-dependent metric is a heatmap showing how a study program covers the chosen guideline. This type of visualization has been implemented by Jafar et Al. in their platform.

6.1.7 Instructors

Currently the platform tracks an instructor's first name, last name, email, and taught course editions. The model for instructors could be extended to include the instructor's skills as they relate to a chosen underlying guideline. If the skills of all instructors are tracked, it could be possible to visualize how the skill set of all instructors covers the chosen guideline similarly to how we currently display coverage by a course or study program.

Tracking instructor skills would make it possible to provide recommendations for new instructors of an existing course. For example, if a professor is unable to teach a specific course edition, the platform could recommend other instructors to replace the missing one based on the requirements of the course and the skills of each instructor.

Currently, only instructors are modeled. Teaching assistants could be modeled separately from instructors. If the skills of each teaching assistant are tracked, the platform could provide recommendations on which teaching assistants to assign to each course. If the time at which each lecture is taught, the platform could also take care of resolving scheduling conflicts, avoiding for example that a teaching assistant has to be an assistant for two parallel lectures.

6.1.8 Backend

Changes across multiple parallel clients of the platform are not synchronized. To implement synchronization, the platform could implement WebSocket communication and constantly notify each user when a piece of data is updated. Alternatively, each client could periodically poll the server for changes to each data the user is currently viewing on screen. An automatic refresh of data should be non-destructive: if two users are working on the same course, study program, or some other object, the changes of one user should not destroy the changes of the other user.

6.1.9 API

The server allows users to retrieve, create, modify, and delete each entity in the system. Interaction is on a per-entity basis. It could be useful to allow users to edit multiple entities in bulk. If such a feature is implemented, it would make it possible to easily export and import the entire database.

Currently, in the platform, everyone can perform every action. This means that, for example, every user can delete all courses and study program from the database. A more refined version of the platform could allow only authenticated users to perform actions on the platform, and restrict access to certain actions by

means of roles. For example, the platform could limit destructive actions like deletion or some types of modification only to administrators. Users could be of various types, such as instructors that are allowed to create and modify courses, study program administrators that are allowed to manage study programs, and more.

6.1.10 Frontend

The home page of the client is clumsy to use. The user is required to read each of the four big buttons to decide which page of the application they want to read. A unified UI that shows relevant information within the home page would be preferred, because in its current implementation, the user always needs to perform one extra step in order to reach the page they are looking for.

The current implementation we used to render treemaps is difficult to use, offers limited options to configure it, and relies on an unintuitive way to color each tile. All of these problems could be solved by implementing an ad-hoc treemap library with reasonable coloring based on maximum value within the provided dataset rather than the maximum value among only the visible values, and with a more sane navigation user interface, displaying for example a back button rather than requiring the user to right click in order to navigate back one level in the tree.

The frontend is entirely written in JavaScript. Throughout development, there have been a few instances where we found it difficult to track and remember the contents of objects passed through multiple functions. To solve this type of problem and make the frontend code more strict and precise, it could be useful to rewrite its entire codebase in TypeScript, a dialect of JavaScript that supports object-oriented programming. We estimate that this type of refactoring could be time-consuming, so it is advisable to evaluate whether the time and efforts required to rewrite the entire codebase are worth the benefit of having a frontend written in TypeScript.

As is currently implemented, the treeview to specify course contents from CS2013 is difficult to use. The current implementation requires users to navigate first a view of knowledge areas, then a view of knowledge units, and finally, choose the relevant elements and mark, from a menu, whether they are required, optional, or not required. Considering that in total CS2013 is composed of $2'222$ topics and outcomes, this long sequence of steps makes it time-consuming to properly specify the contents of a course. A simpler user interface would for example require the user to perform less clicks in order to mark whether an item is required or not. The user interface to specify course contents could be updated to reflect the one presented by Hauswirth[7].

The platform, specifically the user interface, should be evaluated in a real-world scenario with end users. This type of evaluation could highlight problems or improvements that we were unable to identify during development.

Bibliography

[1] V. V. Ajanovski. Evolutionary Curriculum Reconstruction: Process Model and Information System Development. In *Proceedings of the 18th Annual Conference on Information Technology Education - SIGITE '17*, pages 89–94, Rochester, New York, USA, 2017. ACM Press.

[2] T. Auvinen, J. Paavola, and J. Hartikainen. Stops: A graph-based study planning and curriculum development tool. In *The 14th Koli Calling International Conference on Computing Education Research, 20-23 November 2014, Koli, Finland*, pages 25–34. ACM, 2014.

[3] L. Camilloni, D. Vallespir, and M. Ardis. Using GSwE2009 for the evaluation of a master degree in software engineering in the universidad de la república. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 323–332, Florence, Italy, May 2015. IEEE Press.

[4] A. Clear, A. S. Parrish, J. Impagliazzo, and M. Zhang. Computing Curricula 2020: Introduction and Community Engagement. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, pages 653–654, New York, NY, USA, Feb. 2019. Association for Computing Machinery.

[5] A. Clear, S. Takada, and E. Cuadros Vargas. CC2020 – Visualization Tool Preview and Review. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, SIGCSE '20, pages 169–170, New York, NY, USA, Feb. 2020. Association for Computing Machinery.

[6] M. M. Gorman, J. F. Rogers, and E. A. Embick. A data processing curriculum development methodology. In *Proceedings of the Eighth Annual SIGCPR Conference*, SIGCPR '70, pages 27–47, New York, NY, USA, June 1970. Association for Computing Machinery.

[7] M. Hauswirth. Curriculus: A tool for exploring, mapping, and comparing computing curricula. 2018.

[8] N. Herbert, J. Dermoudy, L. Ellis, M. Cameron-Jones, W. Chinthammit, I. Lewis, K. de Salas, and M. Springer. Stakeholder-led curriculum redesign. In *Proceedings of the Fifteenth Australasian Computing Education Conference - Volume 136*, ACE '13, pages 51–58, AUS, Jan. 2013. Australian Computer Society, Inc.

[9] M. Jafar, L. J. Waguespack, and J. S. Babb. A Visual Analytics Approach to Gain insights into the Structure of Computing Curricula. *Information Systems*, page 20, 2017.

[10] A. f. C. M. A. Joint Task Force on Computing Curricula and I. C. Society. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. Association for Computing Machinery, New York, NY, USA, 2013.

[11] M. R. K. Krishna Rao, S. Junaidu, T. Maghrabi, M. Shafique, M. Ahmed, and K. Faisal. Principles of curriculum design and revision: A case study in implementing computing curricula cc2001. *SIGCSE Bull.*, 37(3):256–260, June 2005.

[12] B. Lunt, J. Ekstrom, S. Gorka, G. Hislop, R. Kamali, E. Lawson, R. LeBlanc, J. Miller, and H. Reichgelt. Curriculum guidelines for undergraduate degree programs in information technology. Technical report, New York, NY, USA, 2008.

[13] S. Nordstrom, A. Randrianasolo, E. Borera, and F. Mhlanga. Winds of change: Toward systemic improvement of a computer science program. In *Proceedings of the 14th Annual ACM SIGITE Conference on Information Technology Education*, SIGITE '13, page 201–206, New York, NY, USA, 2013. Association for Computing Machinery.

[14] S. I. of Technology. *Graduate Software Engineering 2009(GSwE2009)*. 2009.

[15] D. C. Rowe, B. M. Lunt, and R. G. Helps. An assessment framework for identifying information technology programs. In *Proceedings of the 2011 Conference on Information Technology Education*, SIGITE '11, pages 123–128, New York, NY, USA, Oct. 2011. Association for Computing Machinery.

[16] T. Sekiya, Y. Matsuda, and K. Yamaguchi. Analysis of computer science related curriculum on LDA and Isomap. In *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '10, pages 48–52, New York, NY, USA, June 2010. Association for Computing Machinery.

[17] C. The Joint Task Force on Computing Curricula. Computing curricula 2001. *J. Educ. Resour. Comput.*, 1(3es):1–es, Sept. 2001.