Requirements and Analysis Document for Project Ocean

Einar Ingemarsson, Felix Bråberg, Mattias Retteli, Oliver Karmetun and Joel Postonen

12/9 - 2019
Version 1

1. Introduction

Choosing courses definitely isn't an easy task. With a massive database of hundreds of courses to choose from, it is difficult to be organized and get a good overview of what is available to you. In order to really get the most of your years at a university you should do proper research about which alternatives are available to you. At chalmers this is rather hard since there is no real overview of the courses available to help you fit together your elected courses with the rest of your studies. On top of that, there is no easy way to find, access and compare all the available courses that Chalmers offers. In the current situation the hard work falls upon the student to manage all elected courses and how they fit together in their general study plan.

The purpose of the application is to fill this gap and provide an intuitive and easy way to plan and fit in your elected courses into your studies at Chalmers. A student will be able to place courses in a study plan and get direct visual feedback about how it fits in the study plan. The student will be able to search and filter courses based on values such as study period, prerequisite courses, examiner and means of examination. Furthermore, the application will automatically tell the student about possible conflicts when they place a course in a specific slot. As students ourselves we see the need for this application and want to use it in order to better plan our years at Chalmers.

1.1 Definitions, acronyms, and abbreviations

CPS - CoursePlanningSystem, the class that exposes the functionality of the application.
StudyPlan - A place where you can save courses into different years that in turn contains two course slots for each of the four study periods.
Slot - A place in the study plan where a course can be placed.
Workspace - A place where the user can place courses for easy access to aid the workflow.
Course type - A course often belongs to a specific field(e.g. Math) and this is what course type describes.
StudyPeriod - Half of a semester and the way that Chalmers organizes its courses.
TKITE - The abbreviation for the software engineering program at chalmers.
UI - User interface, the graphical part of our program that the user interacts with.

IO - Input/Output, the package that takes care of inputting saved data into the model and saving the data when closing the application.
JavaDoc - A type of documentation and a way of describing the code in greater detail.
Git - A version control system, commonly used in programming.
GitHub - A web service which provides hosting for git projects.

2. Requirements

2.1 User stories

User story:

Story Identifier: CPS001
Story Name: Create GUI
Implemented: Yes by Mattias & Einar

Description:

As a programmer I want a basic GUI inorder to have a base to build off.

Confirmation:

Functional:

- Can I resize the window?
- Can I see the GUI?

User story:

Story Identifier: CPS002
Story Name: View Courses
Implemented: Yes by Oliver och Felix

Description:

As a student, I want to be able to see all the courses at Chalmers, in order to know what I have to choose from.

Confirmation:

Functional:

- Can I see a list of all available courses?

- Can I navigate through the list?
- Can I see course name, course code and number of study points for every course?

User story:

Story Identifier: CPS003
Story Name: Course Placement
Implemented: Yes by Joel och Felix

Description:

As a student, I want to place courses in a study plan so that I can visualise my studies.

Confirmation:

Functional:

- Can I see a visual representation of a study plan, with years, study periods etc?
- There is a grid with places to place courses that is sorted into Years and StudyPeriods with clear headings to signify this.
- Can I place a course object in the study plan?
- Can I move a course from one place in the study plan to another?
- Can I remove a course from the study plan?

Non-Functional:

- When an action is made it should only affect the place that is acted upon.

User story:

Story Identifier: CPS004
Story Name: Workspace
Implemented: Yes by Einar

Description:

As a student I want to have a workspace where I can place courses, so I don't have to look them up manually every time.

Confirmation:

Functional:

- Can I see a window where I can place courses temporarily?
- Can I add courses to the workspace?
- Can I remove courses from the workspace?

- Does every course appear as a visual object in the workspace?

Non-Functional:
- Do the courses stay there while I navigate the rest of the application?

User story:

Story Identifier: CPS005
Story Name: Search
Implemented: Yes by Oliver

Description:

As a student I want to be able to search for courses so that I can explore the courses easier

Confirmation:

Functional:
- Can I search by course name and get a correct result?
- Can I search by course code and get a correct result?
- Can I search by examiner and see all courses they teach?
- Can I search with an empty string and get all courses
- Can I filter by study period?
- Can I filter by study points?
- Can I combine terms and get a combined result?
- An empty search should show a result of all courses.
- Pressing enter should result in the same action as clicking the search button.

Non-Functional:
- The search should not be case-sensitive

User story:

Story Identifier: CPS006
Story Name: Detailed Information
Implemented: Yes by Mattias

Description:

As a student I want to be able to see detailed information regarding a specific course.

Confirmation:

Functional:

- Clicking a course should result in showing the information on the course in the application window.
- That information should include course name, required courses, course code, examinator, course description, a link to the course pm, examination means, study period, study points and language.
- Can I close down the window containing the detailed information?

User story:

Story Identifier: CPS007
Story Name: Saving
Implemented: Yes by Mattias

Description:

As a student I want to be able to save my study plans

Confirmation:

Functional:

- When the program gets restarted, studyPlan should contain and show the same information.
- When the program gets restarted, workspace should contain and show the same information.

User story:

Story Identifier: CPS008
Story Name: Indicate placement suitability
Implemented: Yes by Einar

Description:

As a student I want to know if the course placement I have made actually works and get an indication as to where I should place my courses, so I don’t have to look it up manually.

Confirmation:

Functional:

- Do I get an indication as to whether the course I am holding fits in the study plan slot or not. (Red = don't fit, green = fits)
- If the course is placed in a red slot, it should light up with an orange colour and indicate what the problem is, with either tooltip or a drop down message.
- Things that triggers non suitability includes study period and previous required courses
- If the course is placed in a suitable slot, it should not indicate anything.

User story:

Story Identifier: CPS009
Story Name: Switch study plans
Implemented: Yes by Joel

Description:

As a student I want to be able to switch between study plans so I can plan different paths

Confirmation:

Functional:

- I should be able to create multiple study plans.
- I can choose which study plan to display by pressing a button.
- The current displayed study plan should indicate that it is the currently shown study plan.
- When a new study plan is created it is empty.
- When I create a new study plan it should become the current displayed study plan.
- I can remove a study plan.

Non-Functional:

- Does the information on each study plan stay when I switch.

User story:

Story Identifier: CPS010
Story Name: Choosing TKITE
Implemented: No

Description:

As a student I want to be able to choose my program (TKITE) and its study plan so I don't need to put its courses in my study plan manually

Confirmation:

Functional:

- Can I choose my program and automatically build the correct study plan from its obligatory courses.
- If any space where the program automatically puts the courses is occupied, the course in the occupied slot should automatically be placed in the workspace area.
- Can I see if a course is obligatory or not, so that I dont remove it by mistake

----

User story:

Story Identifier: CPS011
Story Name: Course type
Implemented: Yes by Oliver

Description:

As a student I want to know what type a specific course is in order to make it easier when choosing courses.

Confirmation:

Functional:

- Can I see somewhere what field-specific type a course is?
- Courses should be able to have multiple types.

User story:

Story Identifier: CPS012
Story Name: Degree eligibility
Implemented: No

Description:

As a student I want to know if the course placement I have made is enough to give me my degree.

Confirmation:

Functional:

- Can I choose my specific program?
- Do I get an indication as to whether the study points and studyfield-specific courses are enough to give me a degree
- The user should receive an estimate feedback on whether they currently can get a degree or not.

User story:

Story Identifier: CPS013
Story Name: Examiner courses
Implemented: No

Description:

As a student I want to see all the courses a certain examiner teaches

Confirmation:

Functional:

- When a user performs a search with the name of an examiner, all his/her courses should appear in the course list.
- The user should be able to click on the examiner's name in a detailed view and have his/her courses appear in the course list.

User story:

Story Identifier: CPS014
Story Name: Lock placement and reset to workspace
Implemented: No

Description:

As a student I want to be able to reset my study plan and move all courses that are not locked to the workspace.

Confirmation:

Functional:

- Can I press a reset button so that all courses in my study plan are moved to the workspace except the locked ones.

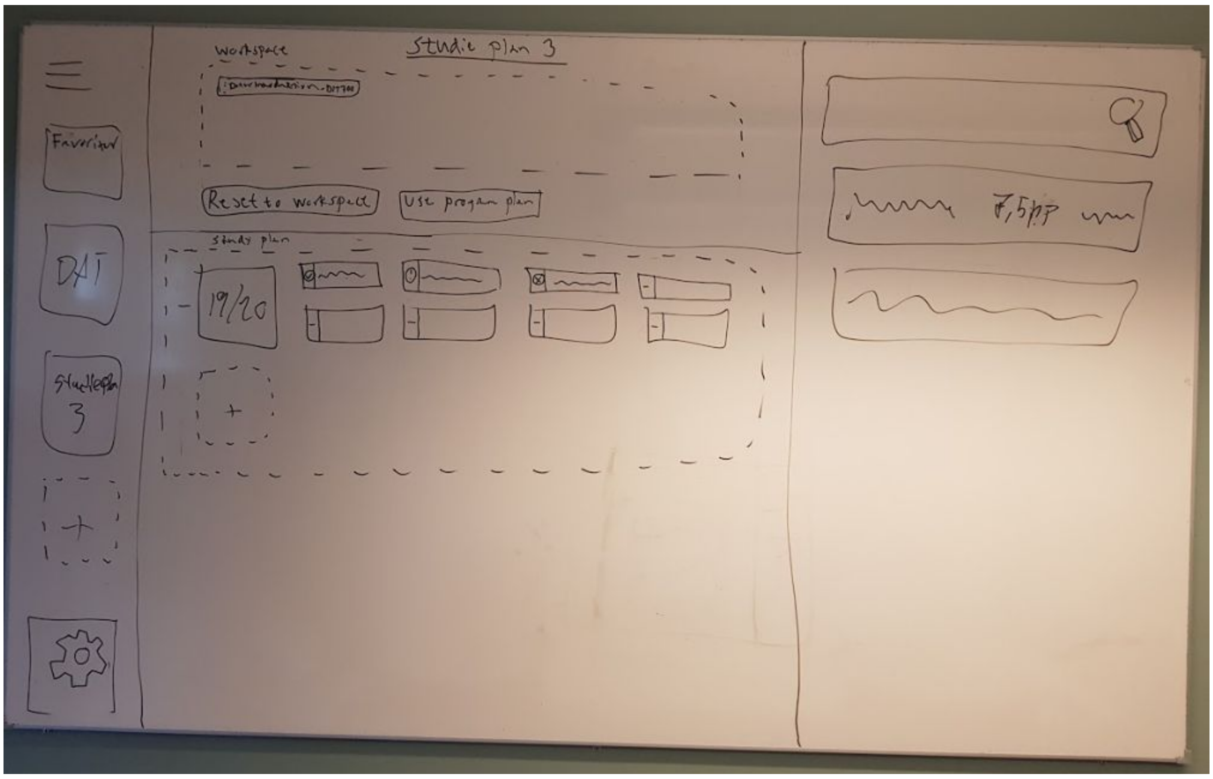
2.2 Definition of Done

For all user stories, the following criterias are the agreed upon standard which need to be fulfilled before it can be viewed as “done”.

- The code implementation needs to be reviewed by another person in the group whom has not been working on the user story.
- The promised functionality should be implemented and this should be verified by the reviewer.
- The code should follow a premade agreed upon code standard which involves naming conventions for variables and methods, JavaDoc for all public methods and comments for behaviour which is not obvious.
- If the code belongs to the model, it should be well tested and preferably achieve full line coverage.
- All tests should be successful when verified.
- It should be available on GitHub in the branch develop.

2.3 User interface

The main goal with the UI is to have everything fit in the same window, without the need to switch between screens. The reasoning behind that goal was to make the application simple and intuitive for the user and those two traits had the most impact when making decisions about the GUI. This can be seen in the way we present the user with information in the picture. Since switching between views is disruptive for the user we made a conscious effort to have most of the needed information in the same view as the user work.



The picture above shows the “home” screen where the user spend most of their time. The user can drag a course from the search panel on the right and drop it in either the workspace area or in the study plan area. The user can also move a course already in the workspace to the study plan and vice versa.

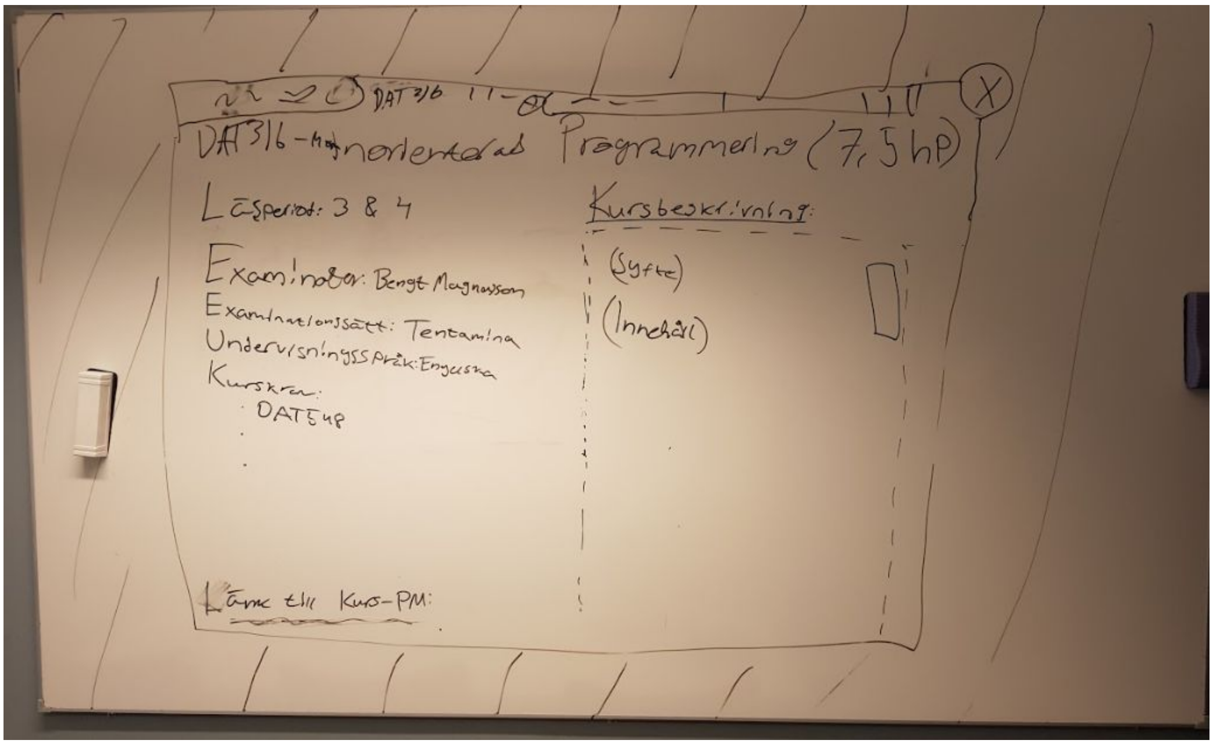
The right-most part of the application acts as a “library”. It should contain all available courses chalmers offers. This is where you find courses by searching with keywords. The result is a scrollbar with course components listed according to search relevance.

The middle part of the application consists of a Workspace and a StudyPlan. These aid each other a lot and are therefore close visually, making the drag distance short.

The workspace section also contains two buttons, “Reset workspace” and “Use programplan”. The reset-button removes courses from the study plan and places them in the Workspace automatically. The programplan-button loads a predefined study plan such as the TKITE program plan. Although useful, these features was not prioritised and thus not implemented.

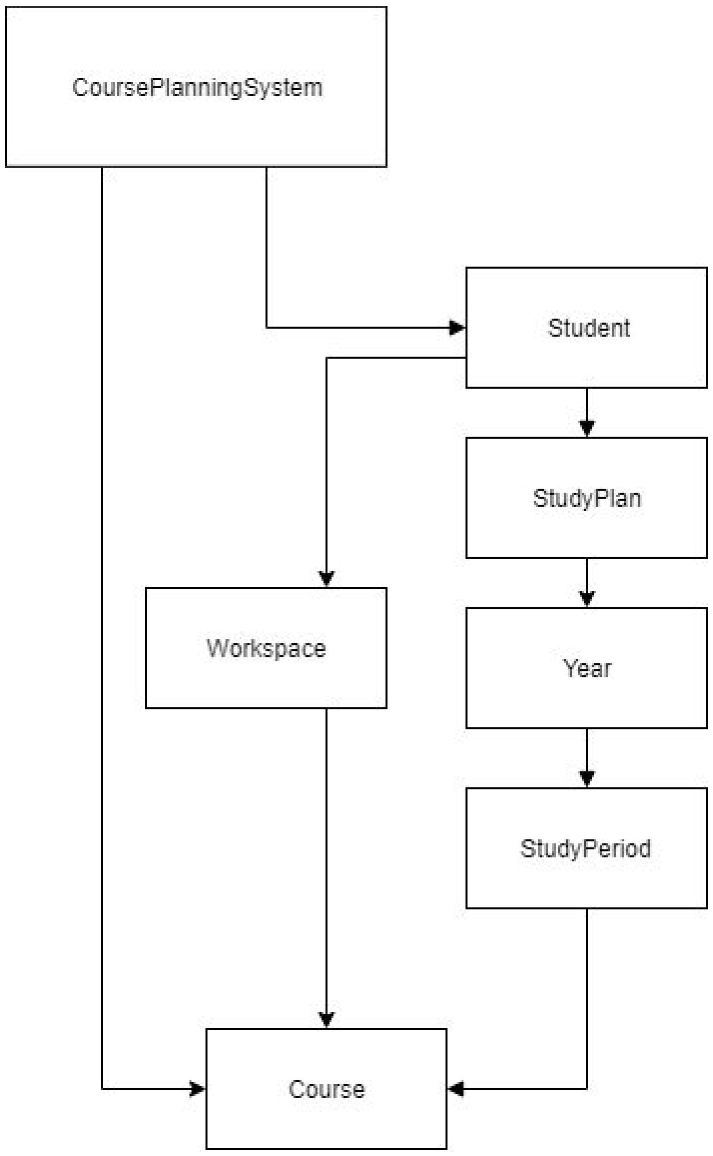
When a course is moving, the StudyPlan should visually show where it can be placed. It should also show visually if a course is currently placed incorrectly. Years should be removable and renamable. The rectangle with a cross represents adding a new year.

On the left is a menu where the user can choose different study plans to display, aswell as add new ones. They should be renamable. In the bottom of the side menu is a settings cog to change global settings such as language or theme. Certain parts of the GUI was not implemented in the final version.


DAT316
X
DAT316 - Objektorienterad Programmering (7,5 hp)
Läsperiod: 3 & 4
Examinator: Bengt Magnusson
Examinationssätt: Tentamina
Undervisningsspråk: Engelska
Kurskrav:
DAT548
Kursbeskrivning:
(Syfte)
(Innehåll)
Länk till Kurs-PM:

When a user clicks on a course the application shows more information. This skiss shows a popup, but we realized it would be more user-friendly and extensible for it to show up in a module tab.

3. Domain model



3.1 Class responsibilities

The domain model as a whole handles all the business logic of the application. The domain is divided into multiple classes with different responsibilities. Below is an explanation of these for each class.

CoursePlanningSystem (CPS)

CPS holds a student and all available courses a user can choose from. CPS exposes the functionality of the application to the public. It is responsible for filling the model with content provided from an external part.

Student
This class represents a student who wants to plan their studies for an academic degree. A student can hold multiple study plans and a single workspace. Its responsibilities are creating, removing and switching between study plans.

StudyPlan
StudyPlan can hold multiple Years and keeps track of them. Its responsibilities are adding and removing Years.

Workspace
Workspace is responsible for holding courses in the workspace area of the application.

Course
This class is an abstraction of an actual and specific course. It is responsible for storing the necessary information needed to represent an actual course. Such information is everything needed for a student to get an understanding of the course itself before deciding whether to apply for it or not. Some examples of information: Course name, course code, description, type of course, study period, examinator etc.

Year
This class is an abstraction of a students academic year. A bachelor student for instance has the scope of three years. A year has responsibility for keeping track of all four study periods within a year.

StudyPeriod
A student in the academics can attend two courses a given study period. Therefore a single class StudyPeriod has responsible to keep track of these two course slots. This class is also responsible for adding and removing courses from these slots.

4. References

Json.simple:
https://cliftonlabs.github.io/json-simple/
Maven:
https://cliftonlabs.github.io/json-simple/
Github:
https://github.com/
Junit:
https://junit.org/junit4/javadoc/latest/

System Design Document for Project Ocean

Einar Ingemarsson, Felix Bråberg, Mattias Retteli and Oliver Karmetun, Joel Postonen

10/10 - 2019
Version 1

1. Introduction

This document describes the project's system architecture, the design model, how the packages depend on each other, how data is loaded from and saved to a users computer. It also discusses topics such as access control, security issues and which tools were used to create the application.

This is an application that enables students to plan their courses taken at university. The GUI part is written using JavaFX and Java. The user is able to search for courses in the search box and drag interesting courses to a study plan. The user can remove courses from the study plan and move them around the different fields of the application. When the user is about to finish the session, the user can then save its progress before termination, or simply close the application and let it save automatically.

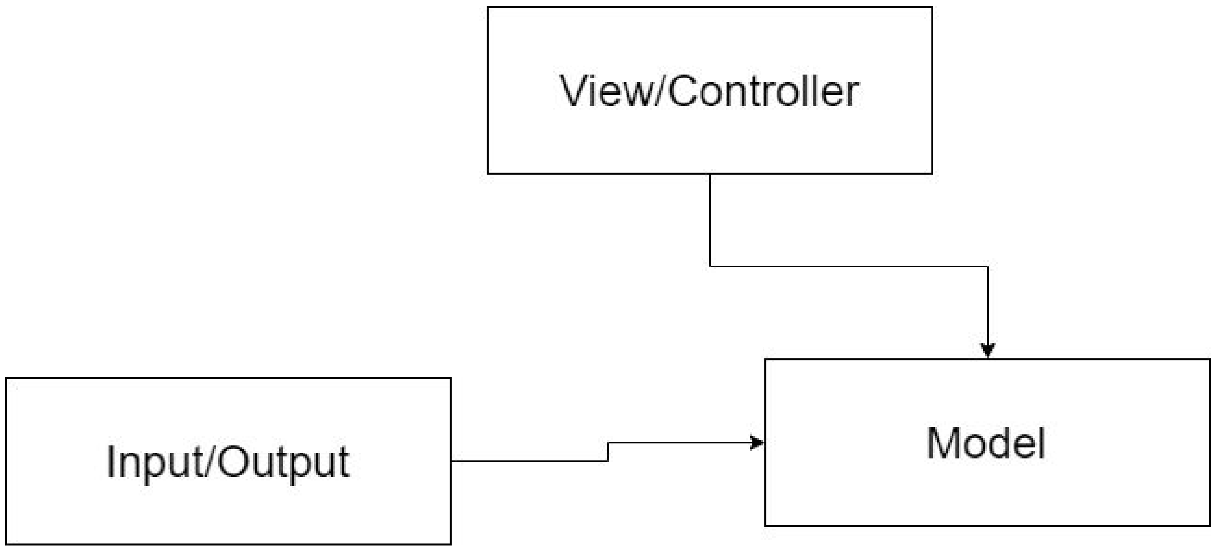
1.1 Definitions, acronyms, and abbreviations

CPS - CoursePlanningSystem, the class that exposes the functionality of the application.
StudyPlan - A place where you can save courses into different years that in turn contains two course slots for each of the four study periods.
Slot - A place in the study plan where a course can be placed.
Workspace - A place where the user can place courses for easy access to aid the workflow.
Course type - A course often belongs to a specific field(e.g. Math) and this is what course type describes.
StudyPeriod - Half of a semester and the way that Chalmers organizes its courses.
TKITE - The abbreviation for the software engineering program at chalmers.
UI - User interface, the graphical part of our program that the user interacts with.
IO - Input/Output, the package that takes care of inputting saved data into the model and saving the data when closing the application.
JavaDoc - A type of documentation and a way of describing the code in greater detail.
Git - A version control system, commonly used in programming.
GitHub - A web service which provides hosting for git projects.
Json - JavaScript Object Notation, is a language to store information i text.

2. System Architecture

2.1 Overview:

The application bases its architecture loosely on MVC, meaning it is split into three modules: model, view and controller. This is because the application uses JavaFX in order to handle UI and graphics the view and controller are so tightly coupled that they can be considered the same. In addition to these two modules, we also have a third one which handles IO for the data of the application. The view-controller is responsible for interacting with the user and sending calls to the model when changes are requested. It also listens to the model for any updates that might have happened and updates itself accordingly. The IO package's job is to provide the model with data at launch time and to save this data outside the application when the program exits, since the model itself does not contain any data when it is not running. The picture below shows the dependencies between the three packages.



2.2 Flow:

The application starts by loading in two JSON files. One is provided with the application and the other should be found inside a folder called ".CousePlanningSystem" in the user's home folder. The provided JSON file contains information about all the courses and is used to generate course objects for each course so the model can handle them. The other JSON file holds the information about how the data is manipulated in the model in order to save the changes made by the user. It does this by saving the state of each StudyPlan and the workspace. If this file does not exist, a question will appear, asking the user if they want to create a new StudyPlan. Pressing no will shut down the application and choosing yes will create the second JSON file at the location earlier described, which will be used to save user made changes when the application saves or shuts down.

After the application has started the ApplicationController is created which in turn creates all the other controllers. Each of the controller loads its corresponding JavaFX file and these in turn contain information about the GUI. The user can interact with the application in various ways by clicking and dragging different components or writing text in the search box. When clicking on a component the corresponding controller receives a JavaFX call. This call is distributed to the model according to its function. Here are the descriptions of what happens during common flows and interactions:

- **Search for courses**: if the user write “Mathematics” in the search box and hit enter or press the search button, a chain of method calls will take place. Firstly a method called “onActionSearch” in the controller class “SearchBrowserController” receives a call from its view. This method sends a call to executeSearch which passes the call to the model which in turn starts a search algorithm. A match is found based on the user input “Mathematics” and returns a list of all matched courses. The controller sends this list through a filter. A filter which sorts out courses based on study period and study points checked in the application. These boxes can be checked before the search starts. The controller creates the needed course components to show the search result.

- **Add a course to a study plan**: a user can move a course around by clicking and dragging it to another area of the application. An example of this is moving a course from the search result area to the study plan area. A method in CourseController gets called when a user drags a course in the application. The method tells the model to delete the course from the area if the area is Workspace or StudyPlan. During the drag sequence a callback named visualFeedback gets called in order to show the user which study period a course can be placed within. A green color appear if the course’s study period match or a red color if it does not. Once the course is dropped in a year’s slot the model assigns the course to the slot and calls on the update method to update all controllers accordingly.

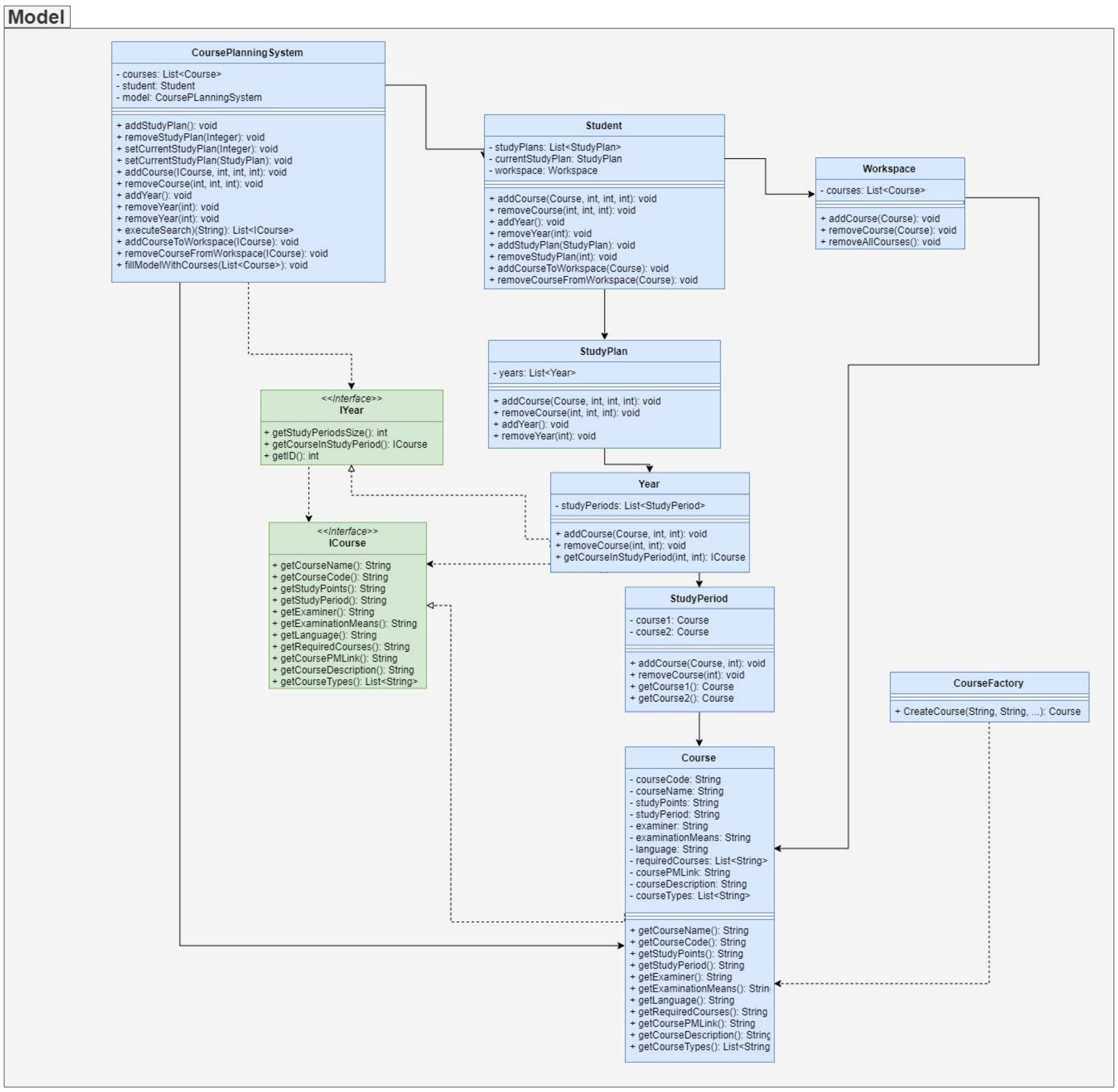
3. System Design

3.1 Design Model and Domain model relationship:

The design model is closely related to the domain model. All the classes in the domain model and their inner relationships also exist in the design model. The differences are that two interfaces, ICourse and IYear, and a class CourseFactory have been added into the design model. Each of the interfaces abstract the functionality of Course and Year. They provide an additional entry point into the model for our controllers. Without them all queries would have to go through CoursePlanningSystem. The CourseFactory separates the creation of a course from the actual course object for our IO package, making the dependency between them weaker.

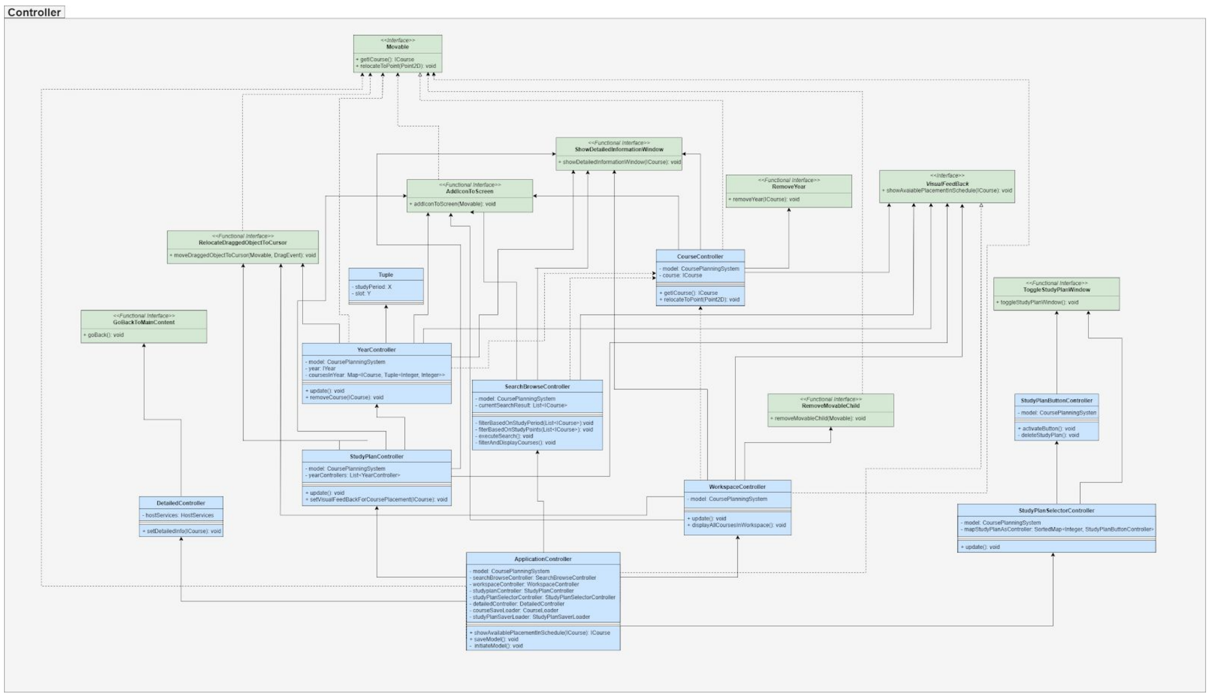
3.2 Packages

3.2.1 Model

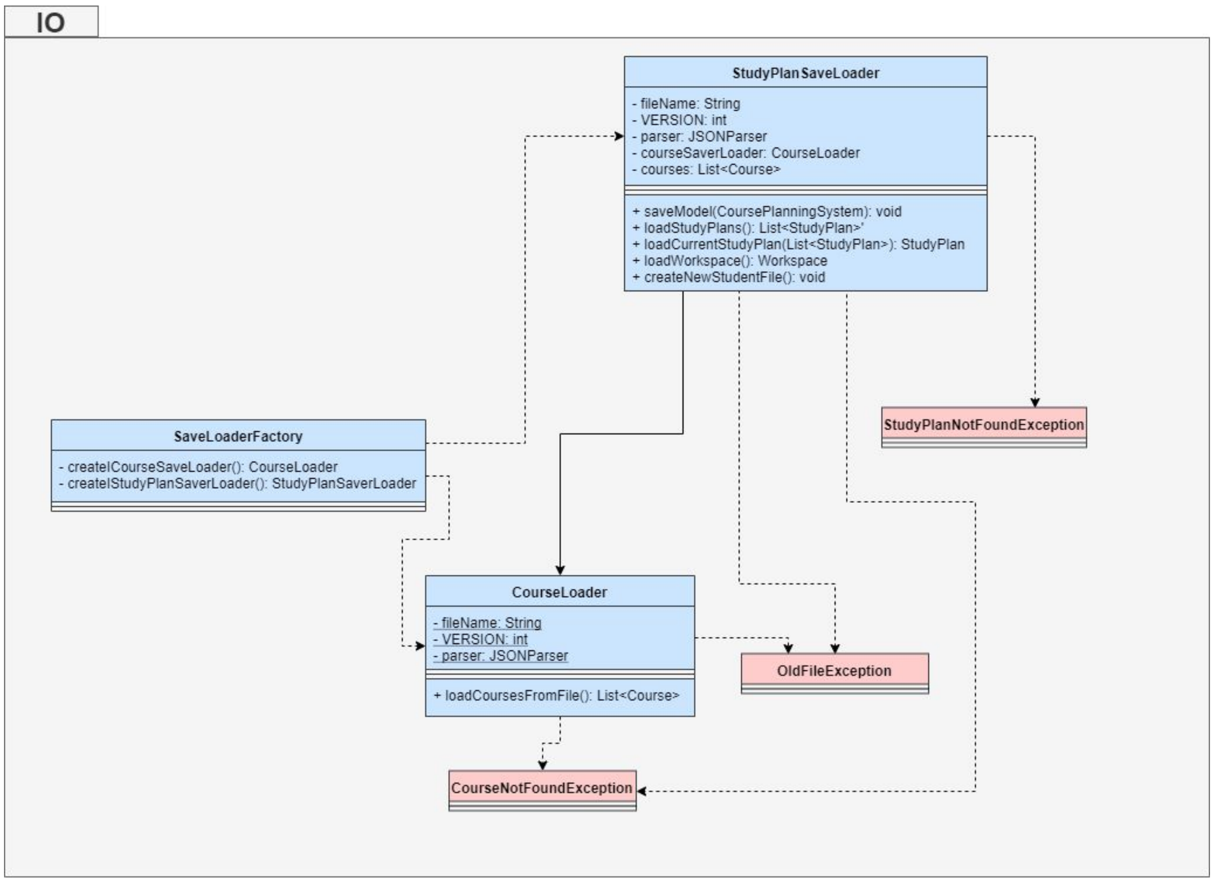


Model
CoursePlanningSystem
- courses: List<Course>
- student: Student
- model: CoursePLanningSystem
+ addStudyPlan(): void
+ removeStudyPlan(Integer): void
+ setCurrentStudyPlan(Integer): void
+ setCurrentStudyPlan(StudyPlan): void
+ addCourse(ICourse, int, int, int): void
+ removeCourse(int, int, int): void
+ addYear(): void
+ removeYear(int): void
+ removeYear(int): void
+ executeSearch)(String): List<ICourse>
+ addCourseToWorkspace(ICourse): void
+ removeCourseFromWorkspace(ICourse): void
+ fillModelWithCourses(List<Course>): void
Student
- studyPlans: List<StudyPlan>
- currentStudyPlan: StudyPlan
- workspace: Workspace
+ addCourse(Course, int, int, int): void
+ removeCourse(int, int, int): void
+ addYear(): void
+ removeYear(int): void
+ addStudyPlan(StudyPlan): void
+ removeStudyPlan(int): void
+ addCourseToWorkspace(Course): void
+ removeCourseFromWorkspace(Course): void
Workspace
- courses: List<Course>
+ addCourse(Course): void
+ removeCourse(Course): void
+ removeAllCourses(): void
StudyPlan
- years: List<Year>
+ addCourse(Course, int, int, int): void
+ removeCourse(int, int, int): void
+ addYear(): void
+ removeYear(int): void
<<Interface>>
IYear
+ getStudyPeriodsSize(): int
+ getCourseInStudyPeriod(): ICourse
+ getID(): int
Year
- studyPeriods: List<StudyPeriod>
+ addCourse(Course, int, int): void
+ removeCourse(int, int): void
+ getCourseInStudyPeriod(int, int): ICourse
<<Interface>>
ICourse
+ getCourseName(): String
+ getCourseCode(): String
+ getStudyPoints(): String
+ getStudyPeriod(): String
+ getExaminer(): String
+ getExaminationMeans(): String
+ getLanguage(): String
+ getRequiredCourses(): String
+ getCoursePMLink(): String
+ getCourseDescription(): String
+ getCourseTypes(): List<String>
StudyPeriod
- course1: Course
- course2: Course
+ addCourse(Course, int): void
+ removeCourse(int): void
+ getCourse1(): Course
+ getCourse2(): Course
CourseFactory
+ CreateCourse(String, String, ...): Course
Course
- courseCode: String
- courseName: String
- studyPoints: String
- studyPeriod: String
- examiner: String
- examinationMeans: String
- language: String
- requiredCourses: List<String>
- coursePMLink: String
- courseDescription: String
- courseTypes: List<String>
+ getCourseName(): String
+ getCourseCode(): String
+ getStudyPoints(): String
+ getStudyPeriod(): String
+ getExaminer(): String
+ getExaminationMeans(): Strin
+ getLanguage(): String
+ getRequiredCourses(): String
+ getCoursePMLink(): String
+ getCourseDescription(): Strinc
+ getCourseTypes(): List<String

3.2.2 Controller


Controller
<<Interface>>
Movable
+ getICourse(): ICourse
+ relocateToPoint(Point2D): void
<<Functional Interface>>
ShowDetailedInformationWindow
+ showDetailedInformationWindow(ICourse): void
<<Functional Interface>>
AddIconToScreen
+ addIconToScreen(Movable): void
<<Functional Interface>>
RemoveYear
+ removeYear(ICourse): void
<<Interface>>
VisualFeedBack
+ showAvaiablePlacementInSchedule(ICourse): void
<<Functional Interface>>
RelocateDraggedObjectToCursor
+ moveDraggedObjectToCursor(Movable, DragEvent): void
Tuple
- studyPeriod: X
- slot: Y
CourseController
- model: CoursePlanningSystem
- course: ICourse
+ getICourse(): ICourse
+ relocateToPoint(Point2D): void
<<Functional Interface>>
ToggleStudyPlanWindow
+ toggleStudyPlanWindow(): void
<<Functional Interface>>
GoBackToMainContent
+ goBack(): void
YearController
- model: CoursePlanningSystem
- year: IYear
- coursesInYear: Map<ICourse, Tuple<Integer, Integer>>
+ update(): void
+ removeCourse(ICourse): void
SearchBrowseController
- model: CoursePlanningSystem
- currentSearchResult: List<ICourse>
- filterBasedOnStudyPeriod(List<ICourse>):void
- fiterBasedOnStudyPoints(List<ICourse>): void
- executeSearch(): void
- filterAndDisplayCourses(): void
<<Functional Interface>>
RemoveMovableChild
+ removeMovableChild(Movable): void
StudyPlanButtonController
- model: CoursePlanningSysten
+ activateButton(): void
- deleteStudyPlan(): void
StudyPlanController
- model: CoursePlanningSystem
- yearControllers: List<YearController>
+ update(): void
+ setVisualFeedBackForCoursePlacement(ICourse): void
WorkspaceController
- model: CoursePlanningSystem
+ update(): void
+ displayAllCoursesInWorkspace(): void
DetailedController
- hostServices: HostServices
+ setDetailedInfo(ICourse): void
StudyPlanSelectorController
- model: CoursePlanningSystem
- mapStudyPlanAsController: SortedMap<Integer, StudyPlanButtonController>
+ update(): void
ApplicationController
- model: CoursePlanningSystem
- searchBrowseController: SearchBrowseController
- workspaceController: WorkspaceController
- studyplanController: StudyPlanController
- studyPlanSelectorController: StudyPlanSelectorController
- detailedController: DetailedController
- courseSaveLoader: CourseLoader
- studyPlanSaverLoader: StudyPlanSaverLoader
+ showAvailablePlacementInSchedule(ICourse): ICourse
+ saveModel(): void
- initiateModel(): void

3.2.3 IO

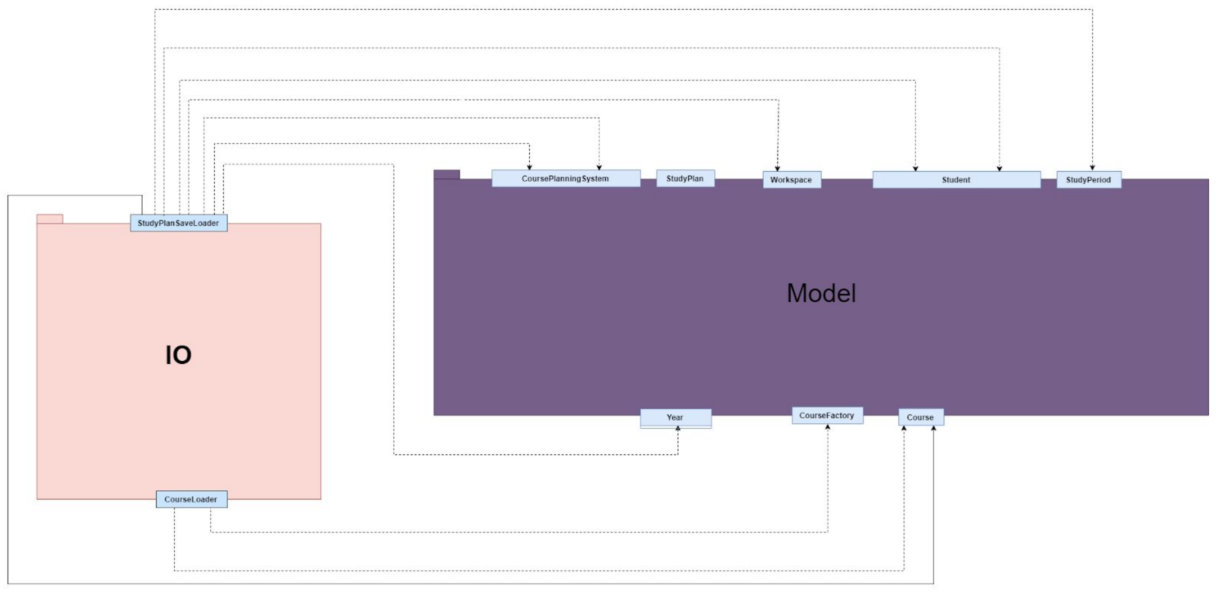

IO
StudyPlanSaveLoader
- fileName: String
- VERSION: int
- parser: JSONParser
- courseSaverLoader: CourseLoader
- courses: List<Course>
+ saveModel(CoursePlanningSystem): void
+ loadStudyPlans(): List<StudyPlan>'
+ loadCurrentStudyPlan(List<StudyPlan>): StudyPlan
+ loadWorkspace(): Workspace
+ createNewStudentFile(): void
SaveLoaderFactory
- createICourseSaveLoader(): CourseLoader
- createIStudyPlanSaverLoader(): StudyPlanSaverLoader
StudyPlanNotFoundException
CourseLoader
- fileName: String
- VERSION: int
- parser: JSONParser
+ loadCoursesFromFile(): List<Course>
OldFileException
CourseNotFoundException

3.3 UML-Class Diagram

The application is divided up according to Model-View-Controller. The difference being that the View and Controller is one in the same package here. This is due to the previously mentioned problem with JavaFX. The following sections show how the different packages are connected with each other.

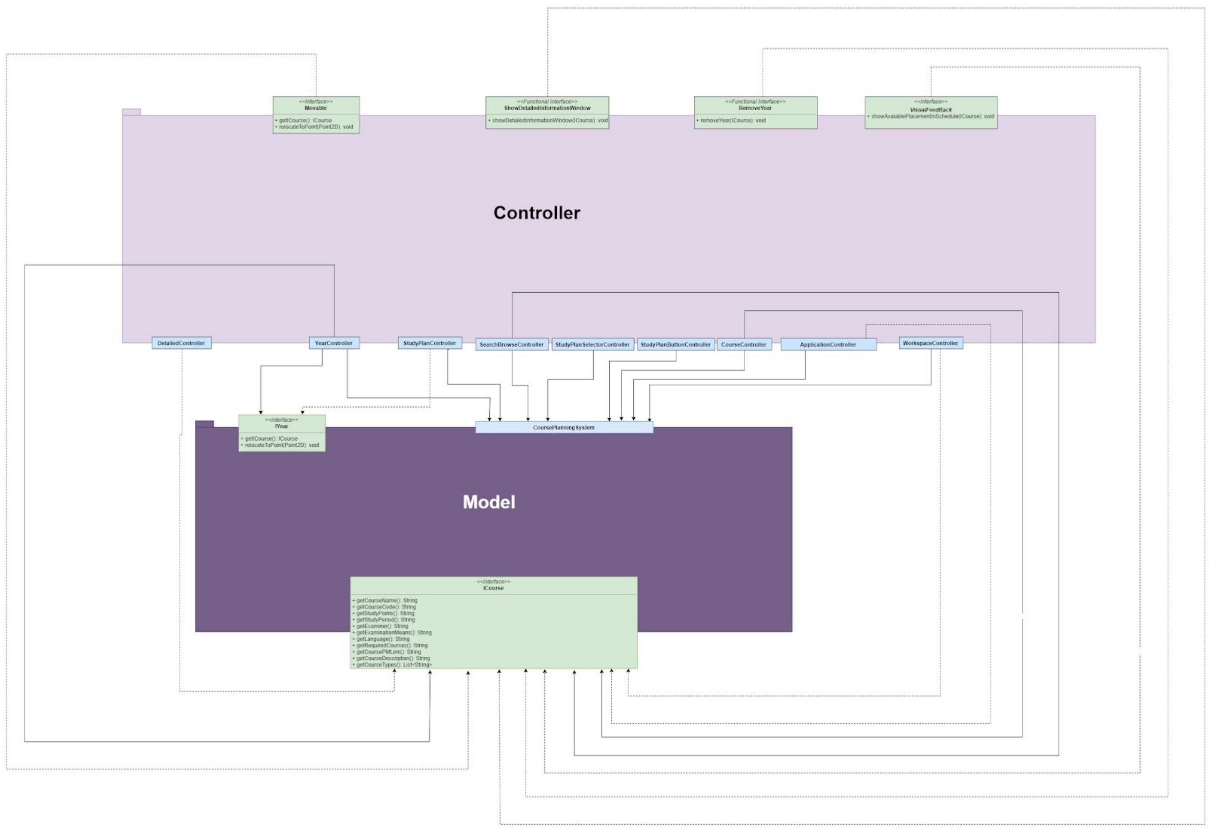
3.3.1 IO to Model

The IO-package uses all of the classes from within the model. This is mostly due to the fact that the IO-package creates the model based on inread data when the application starts.



3.3.2 Controller to Model

As mentioned in 3.1, the controller-package accesses the model through two interfaces and one class.


<<Interface>>
Movable
+ getICourse(): ICourse
+ relocateToPoint(Point2D): void
<<Functional Interface>>
ShowDetailedInformationWindow
+ showDetailedInformationWindow(ICourse): void
<<Functional Interface>>
RemoveYear
+ removeYear(ICourse): void
<<Interface>>
VisualFeedBack
+ showAvaiablePlacementInSchedule(ICourse): void
Controller
DetailedController
YearController
StudyPlanController
SearchBrowseController
StudyPlanSelectorController
StudyPlanButtonController
CourseController
ApplicationController
WorkspaceController
<<Interface>>
IYear
+ getICourse(): ICourse
+ relocateToPoint(Point2D): void
CoursePlanningSystem
Model
<<Interface>>
ICourse
+ getCourseName(): String
+ getCourseCode(): String
+ getStudyPoints(): String
+ getStudyPeriod(): String
+ getExaminer(): String
+ getExaminationMeans(): String
+ getLanguage(): String
+ getRequiredCourses(): String
+ getCoursePMLink(): String
+ getCourseDescription(): String
+ getCourseTypes(): List<String>

3.3.3 IO to Controller

There is no connection between the IO- and Controller-package

3.4 Design Patterns

Factory pattern - CourseFactory is a factory that is used by the IO-package
MVC - The application follows the Model-View-Controller design pattern as the model is completely separated from the rest of the application's aspects such as IO.
Module pattern - In addition to MVC we have also separated our IO package from the rest.
Chain of responsibility - For example, Course planning system has a method addCourse, that calls for
Singleton pattern - The CoursePlanningSystem class is a singleton

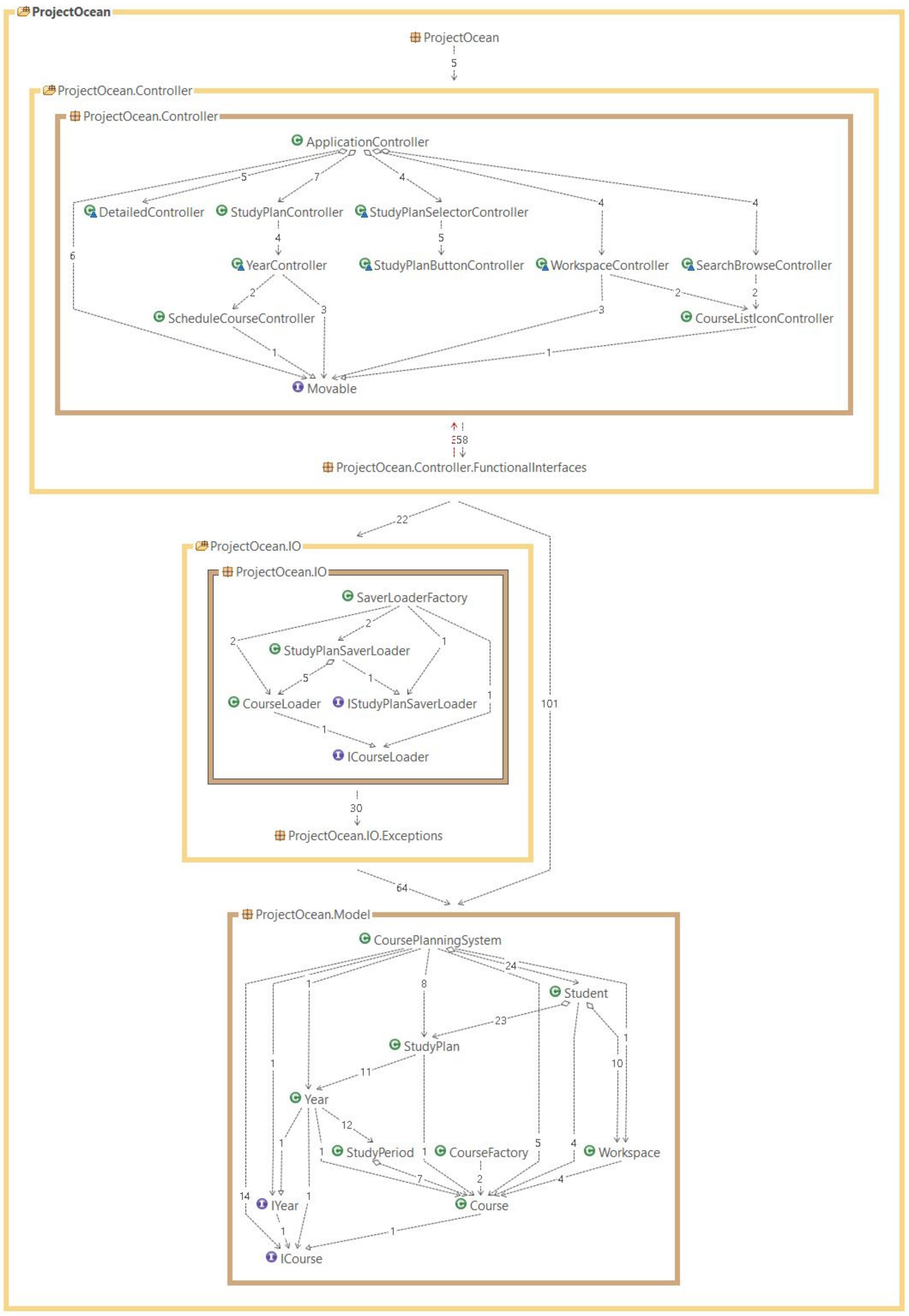
4. Persistent Data Management

When the application starts, the model is empty and in order to fill it we utilize the IO-package to load and save the content of the model to json files. During startup the IO-package loads all courses from a json file provided in the same folder as the application. The json file stores all courses and their attributes. Once the model contains all courses the application tries to load the saved model state from what the user saved the previous time. Once that is done, the user can plan their studies. When the user clicks save or presses the x in the top right corner, the IO-package starts the process of saving the data that the model is currently holding. It stores each course in a position within a year and every year is stored in its corresponding study plan. Each study plan is then stored in a json file together with the contents of the workspace and a pointer to the study plan that is currently shown.

Other resources are stored in the package resources and contains fxml-files, images and a css file. Furthermore, the program has the possibility for easy translation by the use of internationalized strings. There is a file containing the language specific strings and it is stored in the package resources. The file for the swedish translation is called Lang_sv.properties and contains all the translatable strings in Swedish. Currently the only language is swedish but the process of adding more languages is very easy. To add another language, a new file should be created with the proper translated strings.

5. Quality

The application was developed using test-driven development and our goal was to have tests created for each public method prior to writing them. Unfortunately, this was not always the case. However, as a requirement before pushing a user story implementation to Github, JUnit tests had to be written and passed for each new public method. Of course, all previous tests had to pass as well. The tests are located in their own package "tests".



The Static Analyzer tool, STAN, has been used in order to inspect dependencies in the application. STAN shows that there are three circular dependencies in the controller package. Apart from that all packages follows high cohesion low coupling. But most importantly it shows that the model is not dependent on anything.

Furthermore Intellij's built in code analyzer has also been used to further optimize the code. It has aided us in finding unused imports and helped us define a stricter scope to classes and methods. Moreover it has highlighted compiler issues and redundant code.

Issues:

- One of the issues with our program is the resizability of the window. While the program works well in full screen, if the window becomes too small, some parts of it becomes quite unusable. This is because we encountered a problem with ScrollPanes in JavaFX that makes them grow indefinitely with their children if they are located in the wrong container. We tried solving this but could not find any solution in time, so we resorted to fix the specific ScrollPanes size. The fixed size means that at some resolutions it will not look that good in fullscreen.
- We have a dependency from IO to Course in model. This was supposed to be fixed with our CourseFactory and ICourse interface, but during implementation Course was mistakenly used directly. We unfortunately did not have time to correct this. If we had implemented it correctly, it would have lead to a more robust and extensible application.
- We don't show a tooltip if a course is placed on a red slot, instead it only becomes orange, as we forgot to implement this and when we realized, there was not enough time. This was a part of the "Indicate placement suitability" user story. This resulted in a small decrease in usability but no losses in the code implementation.
- We are aware that our controller package is not really optimal. It is very messy with a lot of interfaces and long parameter arguments. This would have been the focus on our next refactor. We partly blame JavaFX but in combination with they way we choose to structure our GUI code.
- In CourseController we currently use a switch case to know where the object was before you started moving it. This could certainly have been improved in some way.
- When the application is showing a detailed view we can, when clicked on add or remove a study plan show both the detailed window and the study plan at the same time. This problem was found too late to be able to solve. However, in future development iterations it would be more optimal to either only show the study plan or force the user to close the detailed window before interacting with study plans.

5.1 Access control and security.

The only security risks involve someone getting the users study plan as it could count as personal data. But the only way someone can get access to the study plan is by having access to the computer, and by that point if someone can see your study plan it's the least of your problems.

6. References

Json.simple:
https://cliftonlabs.github.io/json-simple/
Maven:
https://cliftonlabs.github.io/json-simple/
Github:
https://github.com/
Junit:
https://junit.org/junit4/javadoc/latest/
STAN:
http://stan4j.com

Peer review for group 21

Do the design and implementation follow design principles?

-Does the project use a consistent coding style?
All in all we think that the coding style is very consistent, good job! At some places however, a lot of empty lines are used. An example is at the end of methods, at some places there are 2 empty lines are used, in others 1 and in the rest none. The same goes for beginning of methods, stick to one option, preferably no unnecessary empty lines.

Also one strange thing that was found was in the Report class. At around line 180, is the “userExists = true” statement supposed to be outside of the if block?

There’s a method called getFeedBack in the User class that just returns 0.

-Is the code reusable?
There are some methods that are quite massive and not easily reused. This could be addressed by breaking down bigger methods into smaller ones. This would potentially make the smaller methods more reusable by other parts of the program.

-Is it easy to maintain?
The application seems to be relatively easy to maintain as there are only a few number of dependencies between the different classes.

-Can we easily add/remove functionality?
Certain parts are easy to add to and remove from, but others are nearly impossible to scale. VehicleTimeTable for example uses only hard-coded strings and a switch with a case for each station. From the RAD we understand that you intend to use Västtrafiks API, and if that means this class would be removed or drastically changed then its fine.

-Are design patterns used?
The only apparent design pattern is the Model-View-Controller(MVC) pattern. The application divides its classes and components in to a Model-package and a Controller-package. You have an interface TravelInformation which currently is unused but an indicator that you might use dependency inversion principle in the future.

Is the code documented?
There is a lot of javadoc and most of it is written very well, but at some places it can seem inconsistent with the rest. For example, in most places you state the type that will be returned, but at getNegativeFeedback Report, you simply state “returns the negative feedback”, with no clue as to how this is represented. In other places there

seems to have been less effort put in into wording, such as “This method is to create and add vehicles obj to the list of vehicles“ at addVehicles in Station. Might be worth going through it all and make it consistent.

Are proper names used?
The names of variables and methods are descriptive and intuitive. Really good job! There are a few places where things are spelled incorrectly and destination seems to have 4 different spelling throughout the program. (destnination, destention, destnation, destination)

- Is the design modular? Are there any unnecessary dependencies?
- Does the code use proper abstractions?

There are a number of methods that are a bit too complicated and should be broken down into smaller more concise methods. This to make it easier to understand how the methods in question work.

Is the code well tested?
At the moment the line coverage is 24%, which is unimpressive since we are doing test driven development. This coverage should be higher. Many tests are commented and some of them are failing. Private methods are trying to be tested, but they may be tested indirectly through public methods instead. A few classes is not tested at all.

Are there any security problems, are there any performance issues?
The application gives a null pointer-exception when the search box is empty and when the user searches for a station that does not exist. Other than that the application performs as you would expect, with a fast response time and no loading time between scene switching.

Is the code easy to understand? Does it have an MVC structure, and is the model isolated from the other parts?
The model is correctly isolated from the view and controller. All calls to the model goes through the model aggregate object Connector. Although the model is separated from the view and controller the view and controller are not separated from each other. Both the view and controller are strongly linked that inturn makes it harder to swap out the graphical library if, in the future, it’s needed to replace with another library.

Can the design or code be improved? Are there better solutions?
Setting the label to “back button works” seems like a waste of time since you wouldn’t see the message if the button actually did work. (Which is does not, even though it tells me it does.)

We would advise to use a version of scenebuilder which lets you avoid the error messages that clutters the terminal.

We also found some dead imports :

- AnchorPane in main.
- SearchContorl and ArrayList in Connecter
- EventHandler in SearchControl

We are also a bit confused as to what Grupp21 means in the context of the domain model.