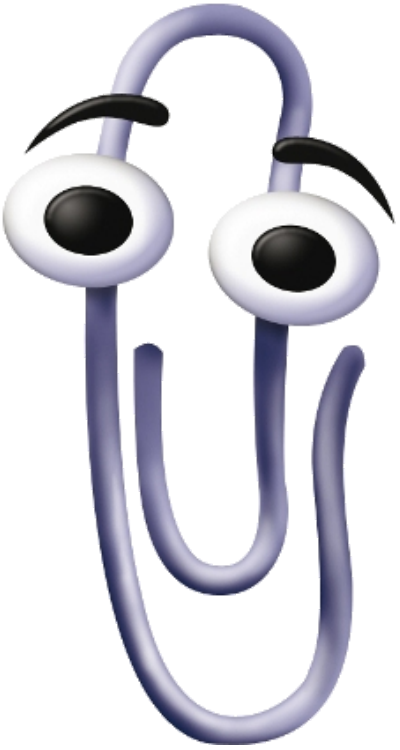
16.06.22
Version 1.0

SWT Projekt

Individueller interaktiver Studienplaner

Design-Dokument von
Mike Daudrich, Christian Kunkel, Lara Reitz, Pablo Schneider



Versionshistorie

Version	Datum	Änderung	Autor:in
0.1	03.06.22	Bausteinsicht erster Entwurf hinzugefügt	Chris, Lara, Mike, Pablo
0.2	07.06.22	Einführung / Ziele hinzugefügt	Chris, Lara, Mike, Pablo
0.3	10.06.22	Bausteinsicht verfeinert	Chris, Lara, Mike, Pablo
0.4	12.06.22	Laufzeitsicht erste Entwürfe hinzugefügt	Chris, Lara, Mike, Pablo
0.5	14.06.22	Bausteinsicht finalisiert	Chris, Lara, Mike, Pablo
1.0	16.06.22	Laufzeitsichten finalisiert, Beschreibungstexte hinzugefügt, Gliederung hinzugefügt	Chris, Lara, Mike, Pablo

Gliederung

1 Einführung und Ziele

Das Ziel der Anwendung ist es, Studierende bei der Planung ihres Studiums zu unterstützen.

Eine genaue Ausarbeitung der Einführung und Ziele findet man im Dokument Anforderungsspezifikation, Version 1.0, Seite 1 f.

2 Bausteinsicht

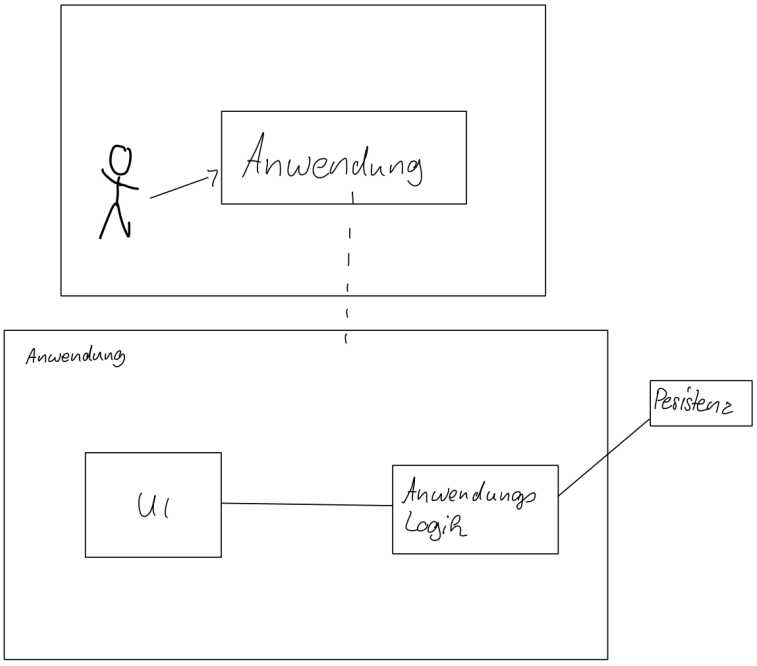


Abbildung 1: Grobe Ansicht des Systems

Die Anwendung kann von Studierenden genutzt werden. Studierende haben dabei jedoch nur Zugriff auf die UI (Benutzeroberfläche). Diese wird in ihrer Funktionalität durch die dahinterliegende Anwendungslogik gestützt. Die Anwendungslogik erhält notwendige Informationen per Zugriff auf die Persistenzschicht, welche für den Nutzen des Systems lediglich aus einer .json-Datei mit dem darin enthaltenen Studienplan besteht (dieser enthält Module, Abhängigkeiten, etc.).

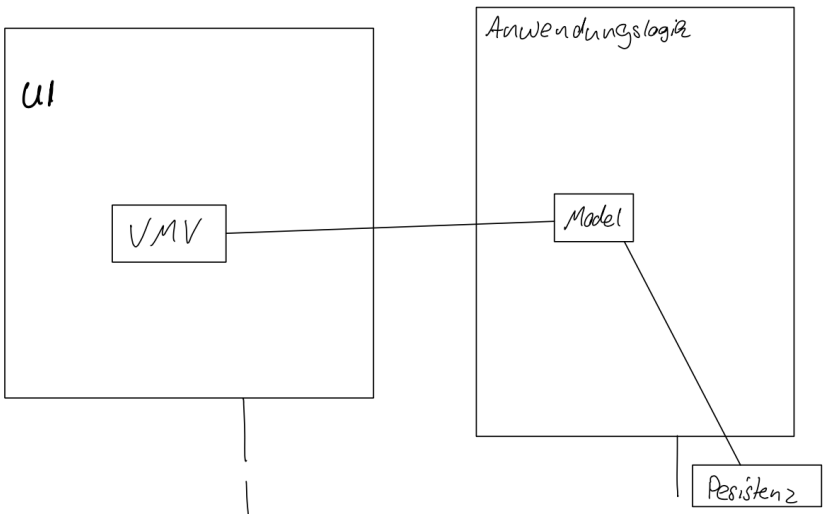


Abbildung 2: Inhalt der UI und Anwendungslogik

Die UI ist als View ViewModel aufgebaut, welcher auf das Model der Anwendungslogik zugreift. Das Model wiederum kann auf die Persistenzschicht für das Speichern und Laden von Daten zugreifen.

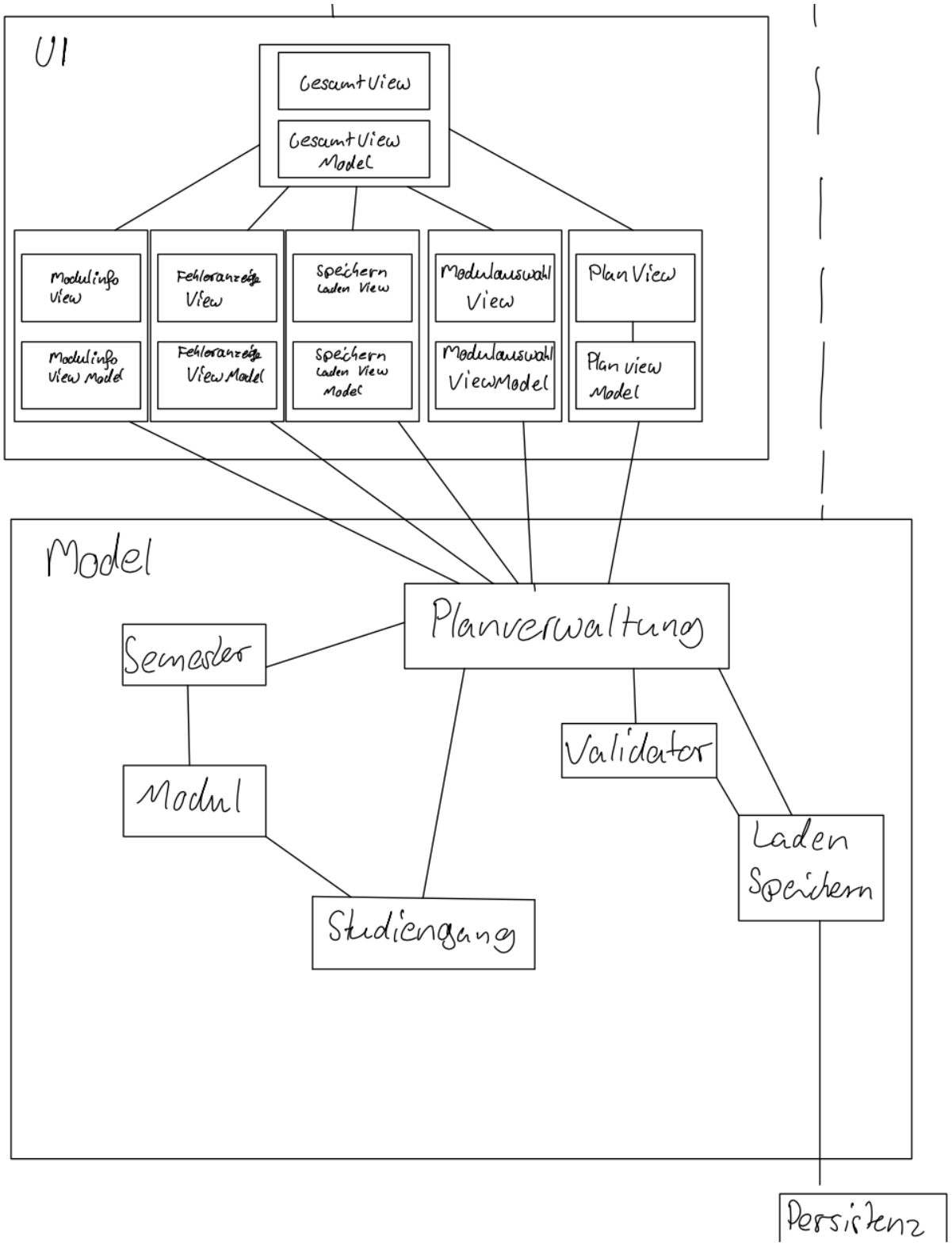


Abbildung 3: Detailansicht zur UI und zum Modell

Die UI besteht aus mehreren kleineren Views mit jeweils einem ViewModel, welche in einer GesamtView mit ViewModel verwaltet und als Scene zusammengebaut werden werden. Die Einzelnene ViewModel kommunizieren mit der Planverwaltungs-Klasse des Model, welcher alle Komponenten des Models zusammenführt. Dieser baut auf dem Dämonen Model (Anforderungsspezifikationen-Dokument Abbildung 2) auf.

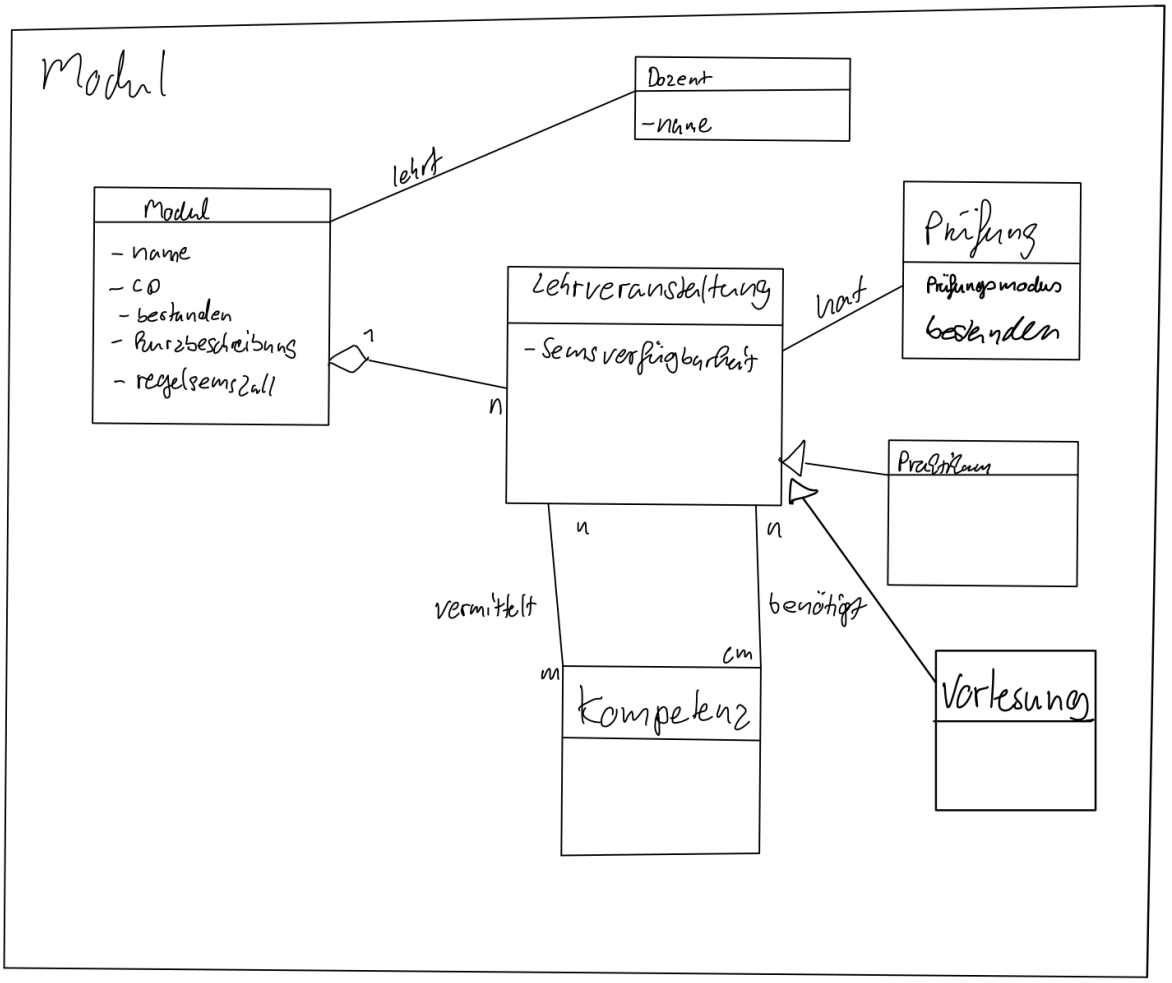


Abbildung 4: Detailansicht zum Modul

Das Modul-Baustein besteht aus einer Modul-Klasse. Ein Modul hat mehrere Lehrveranstaltungen, z.B. Praktikum und Vorlesung. Lehrveranstaltungen benötigen und vermitteln Kompetenzen und haben gegebenenfalls auch noch eine Prüfung.

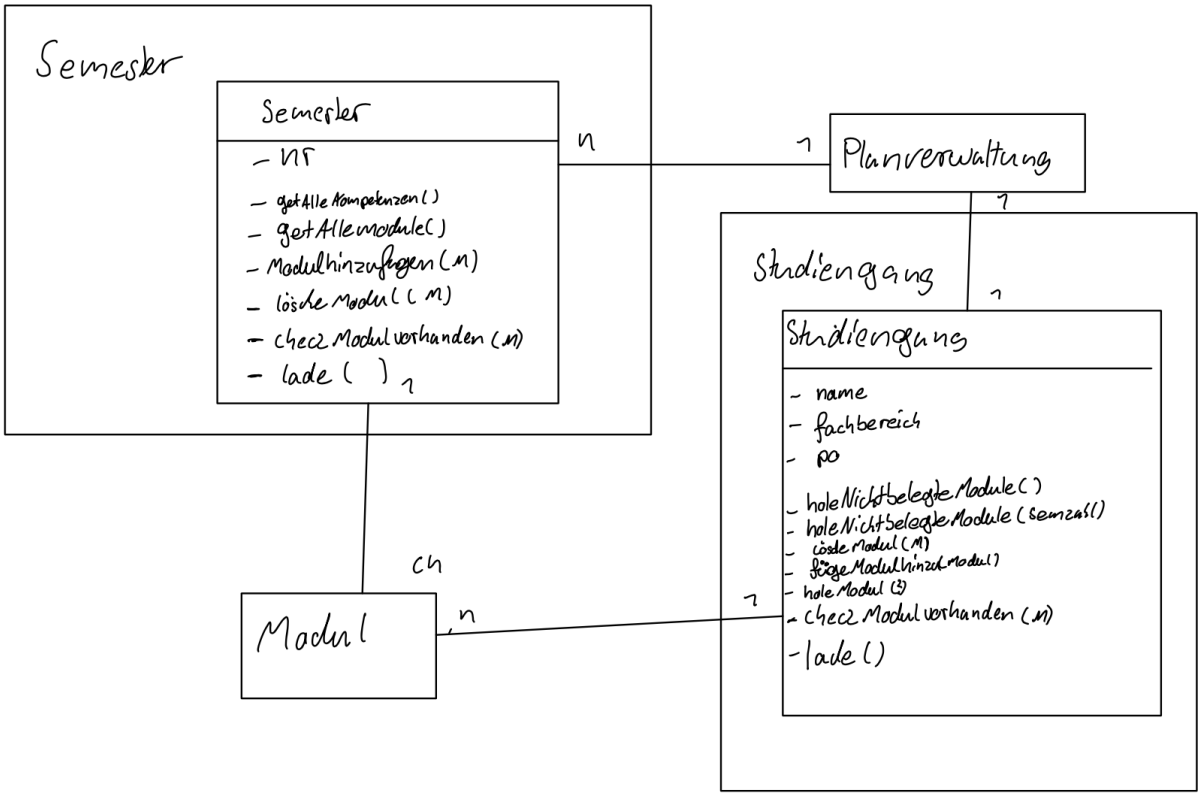


Abbildung 5: Detailansicht zum Semester und Studiengang

Das Wesentliche dieser Anwendung ist es Module zu verwalten. Einmal in einem Studienplan und einmal in einer Auswahl von Modulen, die der User noch planen soll. Der StudienPlan wird in der Planverwaltung verwaltet und verwaltet Module, die Semestern zugewiesen. Neben denen verwaltet die Planverwaltung einen Studiengang. In diesem sind Informationen über den Studiengang und die Module, die noch ausgewählt werden können. Für die Module, Semester und den Studiengang sind hier keine Getter und Setter extra angegeben; sie werden aber im finalen Programm sein. Hier sind nur die wesentlichsten Funktionalitäten angegeben, um auf die jeweiligen Listen zuzugreifen.

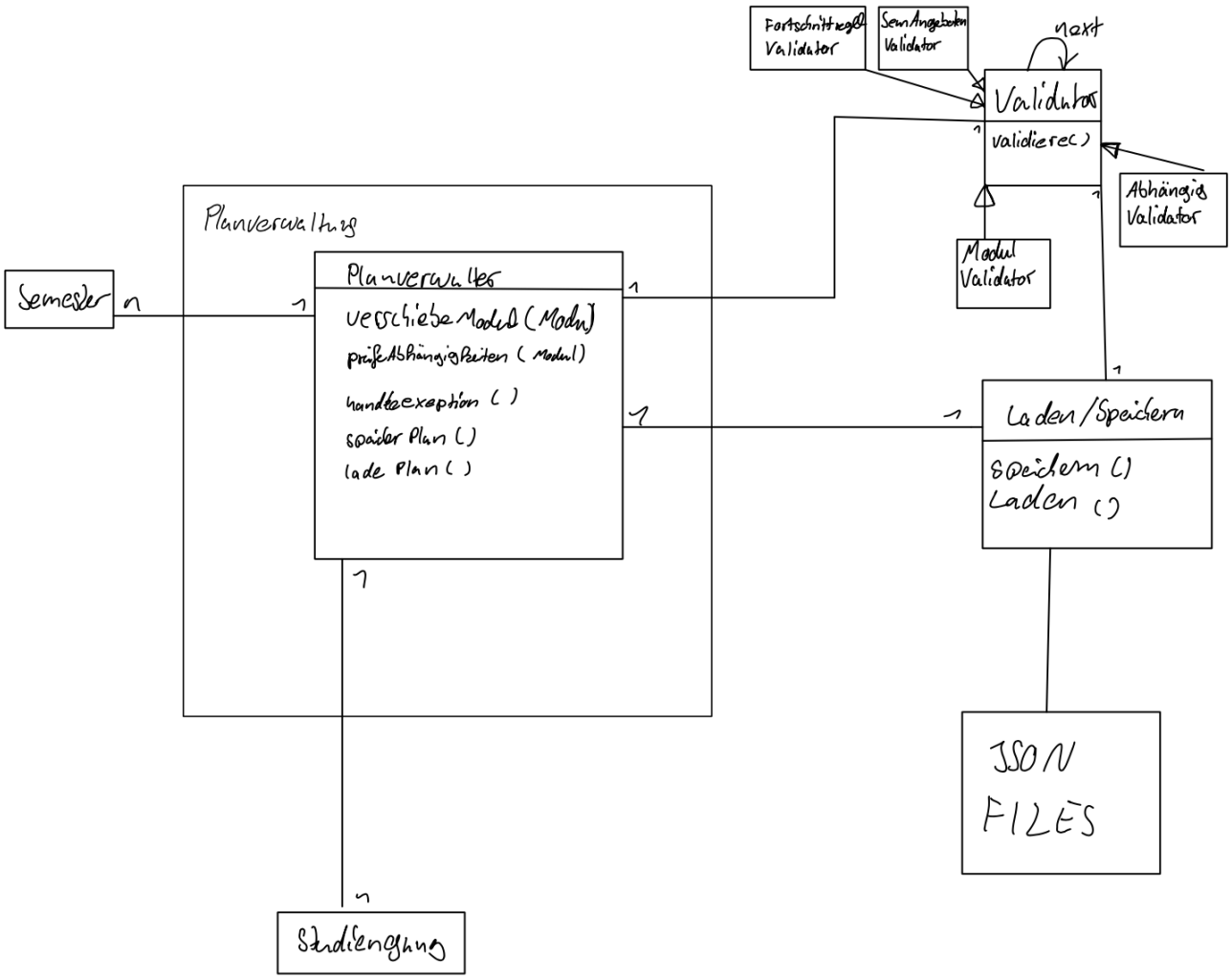


Abbildung 6: Detailansicht zur Planverwaltung und Validator

Der PlanverWalter soll, wie oben erklärt, den StudienPlan und die im Studiengang auswählbare Module verwaltet werden. Hierzu sollen verschiedene Funktionalitäten ermöglicht werden. Darunter das Verschieben von einzelnen Modulen, wenn keine Regeln verletzt werden. Unter den Regeln ist die Fortschrittsregel und die Semesterverfügbarkeit. Hier soll auch das Laden und Speichern ermöglicht werden. Da hier alle Daten verwaltet werden, sollen die, die die die die Gui braucht, mit Java Beans oder JavaFX Beans zugreifbar gemacht werden.

Das Parsen und Auswerten der Json Files sind in einer separaten Klasse ausgelagert.

3 Laufzeitsicht

3.1 Verschieben von Modulen erfolgreich :D Lara Reitz

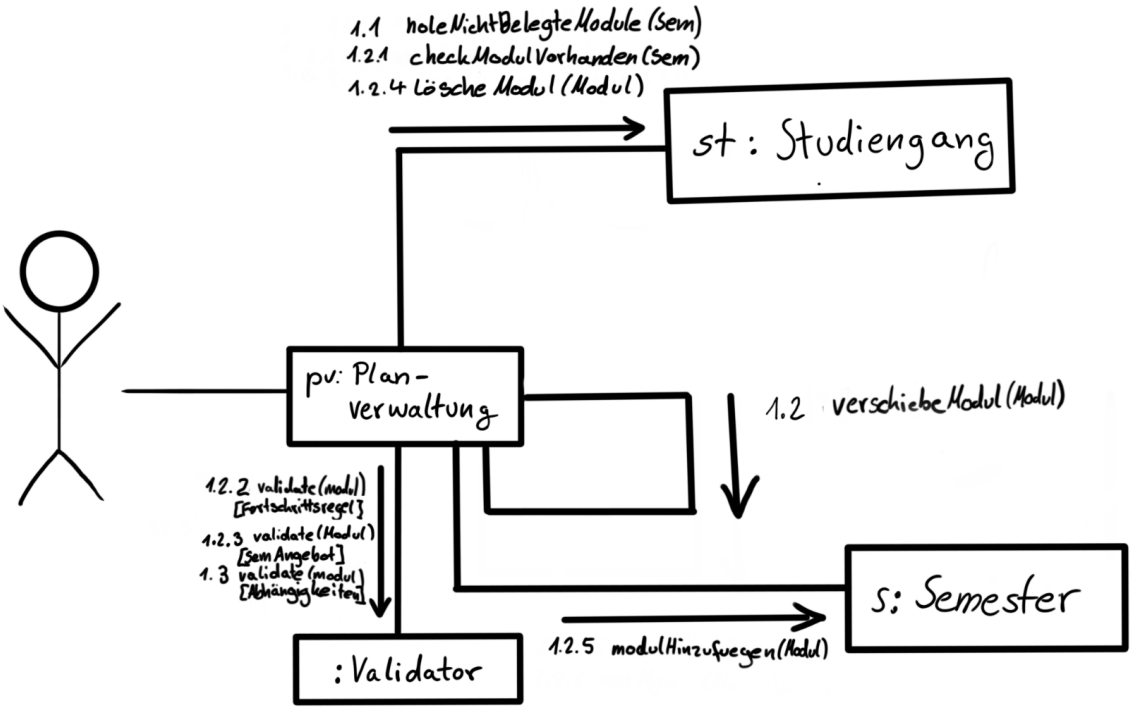


Abbildung 7: Kommunikationsdiagramm zum erfolgreichen Verschieben eines Moduls

Der Student möchte ein Modul aus einem spezifischen Semester auswählen. Hierzu wählt er im Menü auf der rechten Seite des Programms ein Semester aus. Durch das Anklicken des gewünschten Semesters wird im System eine Filterung der Module nach Semester ausgelöst. Alle bisher noch nicht belegten Module aus diesem Semester werden dann in der Modulansicht aufgelistet. Syntaktisch erfolgt dies über die Methode 1.1 holeNichtBelegteModule(Semester), die innerhalb der Planverwaltung aufgerufen wird, und dann die entsprechenden Semester aus dem Studiengang filtert. Nachdem dies stattgefunden hat, wird der Verschiebevorgang eingeleitet, dieser wird über die Methode 1.2 verschiebeModul(Modul) ausgelöst. Innerhalb dieser Methode werden mehrere Zwischenschritte durchgeführt. Zu Beginn wird im Studiengang geprüft, ob das vom Studenten ausgewählte Modul existiert (1.2.1 checkModulVorhanden(Sem)). Daraufhin wird das Modul und dessen zukünftige Position hinsichtlich der Fortschrittsregelung (1.2.2 validate(Modul) + :FortschrittsregelValidator) validiert, sowie ob das Modul in diesem Semester überhaupt angeboten werden kann (1.2.3 validate(Module) + :SemesterNichtAngebotenValidator). Sind alle Überprüfungen erfolgreich vonstatten gegangen, so wird das Modul aus dem Studiengang gelöscht (1.2.4 löscheModul(Modul))

und dem Semester in dem Planer hinzugefügt (1.2.5 modulHinzufuegen(Modul)). Zum Schluss wird das Modul in dem Semester noch ein letztes Mal auf seine Abhängigkeiten hin validiert (1.3 validate(Modul) + AbhängigkeitenValidator) und ggf. eine Warnmeldung ausgegeben, falls die Validation fehlgeschlagen ist.

3.2 Verschieben von Modul schlägt fehl : (Pablo Schneider

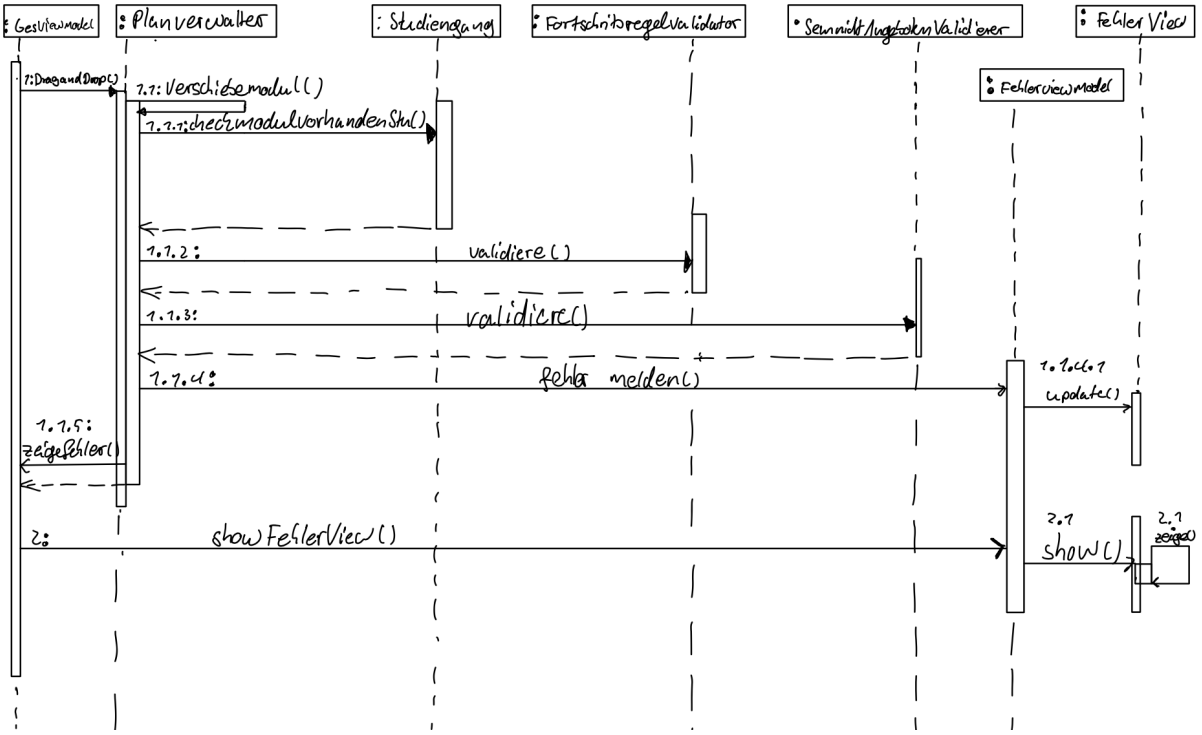


Abbildung 8: Sequenzdiagramm zum fehlerhaften Verschieben eines Moduls

Der Student will nun auch noch ein anderes Modul in den Planer verschieben.
Das geschieht durch die in dem GesamtViewModel vorhandene 1: dragAndDrop-Methode, welches im Model die 1.1: VerschiebeModul-Methode des Planverwalters anstößt. Dort wird erstmal sicherheitshalber überprüft, ob das Modul noch in der Studiengangsklasse vorhanden ist (1.1.1). Die Studiengangsklasse verwaltet die noch nicht ausgewählten Module. Wenn das Modul vorhanden ist, wird der 1.1.2: FortschrittsValidator (Welcher überprüft, ob bei der Verschiebung des Moduls die Fortschrittsregel verletzt wird) und danach der 1.1.3: SemesterNichtAngebotenValidator (Welcher überprüft, ob das Modul in dem Semester angeboten wird.) aufgerufen.
Wenn ein Validator nicht sein ok gibt, was für das Beispiel nun der Fall ist, wird das Modul nicht verschoben und der Fehler wird nach oben in die View geschickt(1.1.4, 1.1.4.1), sodass der Fehler dem Benutzer angezeigt werden kann.
Es sollte sich dann eine “FehlerView” oben mittig im Fenster der Anwendung bemerkbar machen(1.1.5 - 2.1). (siehe Abbildung 8, Anforderungsspezifikationen).

3.3 Laden eines Plans Christian Kunkel

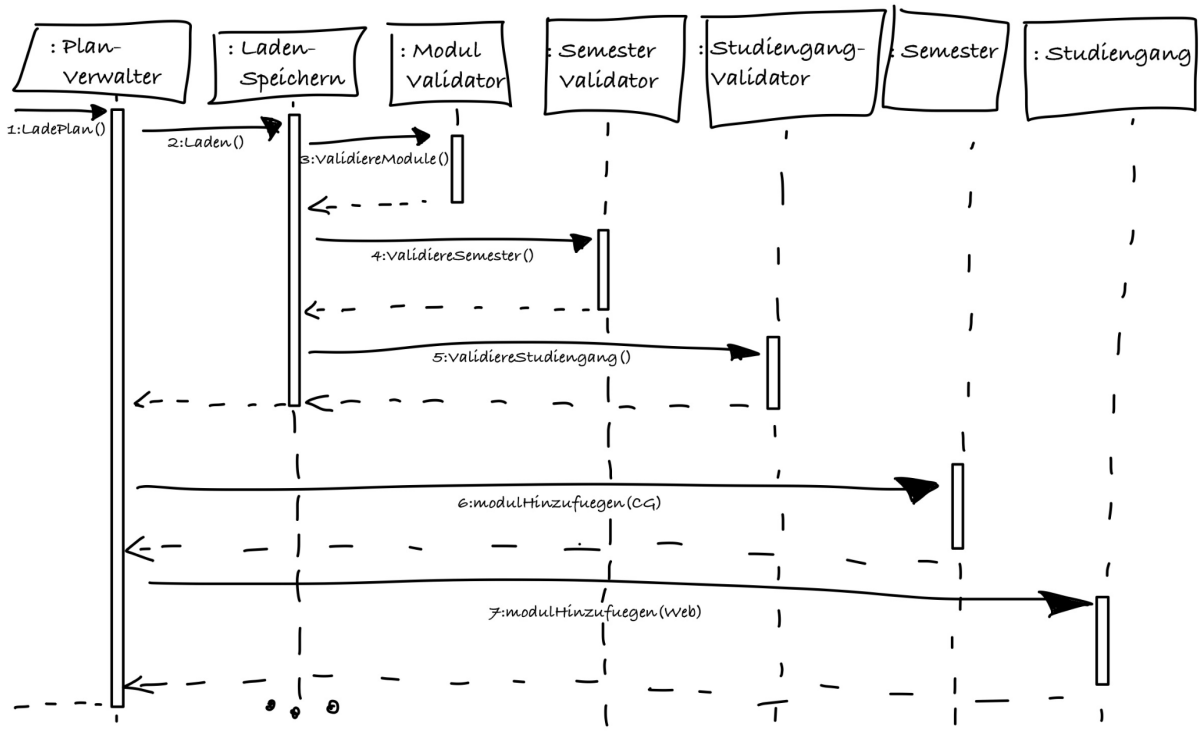


Abbildung 9: Sequenz-Diagramm zum Laden eines Studienplans

Der Student will einen Studiengang für seine Planung laden. Über den Button “neuer Plan” wie in Abbildung 3 der Anforderungsspezifikation gezeigt, wählt er ein File aus, um dieses zu öffnen. Dadurch wird die Methode LadePlan() im SpeicherLadenViewModel angestoßen, welche wiederum die Laden() Methode aufruft. In dieser werden zuerst die Module auf Vollständigkeit und Syntax überprüft. Das Ganze wird auch für die Semester und den Studiengang gemacht. Damit kann sichergestellt werden, dass unser JSON gültig ist. In der Klasse LadenSpeichern wird das JSON Objekt, welches aus Key/Value paaren besteht, in ein Dictionary geparst. Dieses wird daraufhin benutzt, um die Liste der bestandenen Module in Semester und die Liste der nicht bestandenen Module in Studiengang zu füllen. Sobald diese gefüllt sind, können die PlanView sowie die ModulAuswahlView über ihr Model aktualisiert werden, um die Module auch im Planer anzuzeigen.

3.4 **Details anzeigen** Mike Daudrich

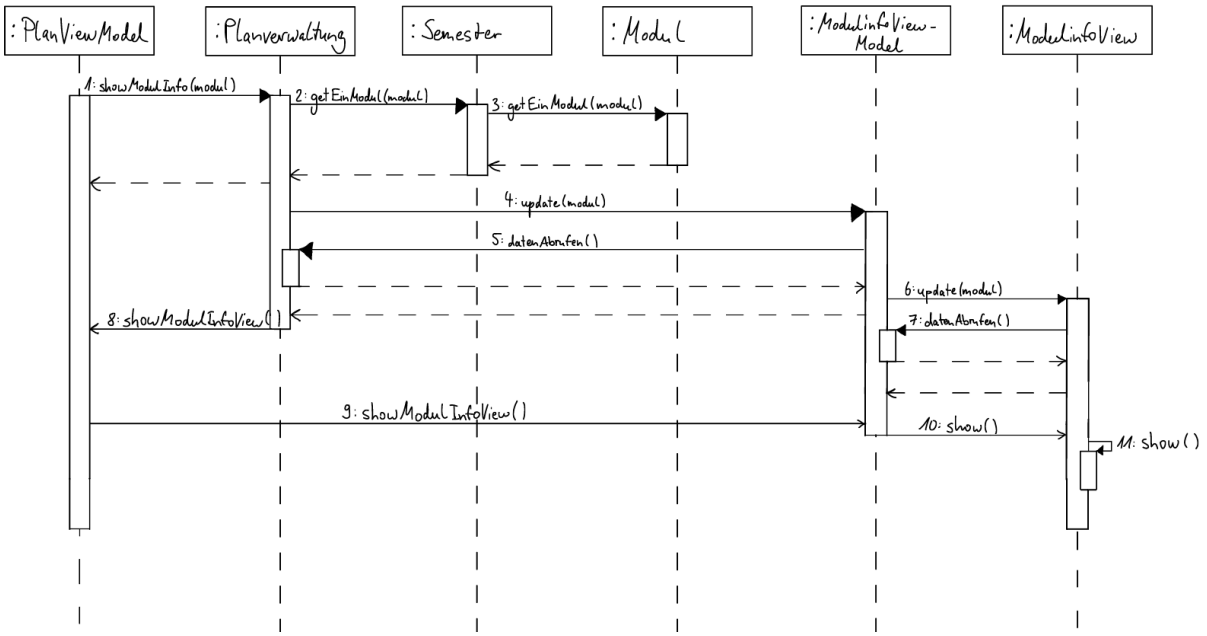


Abbildung 10: Sequenzdiagramm zur Detailanzeige

Der Studi möchte sich die Details zu einem Modul anzeigen lassen. Dazu klickt er ein Modul an. Das PlanViewModel ruft dabei die showModulInfo() Methode mit dem Parameter modul auf, welcher dann durchreichende getEinModul(modul) Methoden aufruft (siehe Abbildung 9). Diese übergeben dann die Informationen vom Modul zurück an die Planverwaltung. Das ModulinfoViewModel wird mit den Informationen zum aktuellen Modul geupdated, was dann wiederum die entsprechende View updated. Zum Schluss wird die ModulinfoView aufgerufen und damit dem Studi die Informationen zum Modul angezeigt (siehe Abbildung 4, Anforderungsspezifikationen).

3.5 Abhängigkeiten anzeigen Mike Daudrich

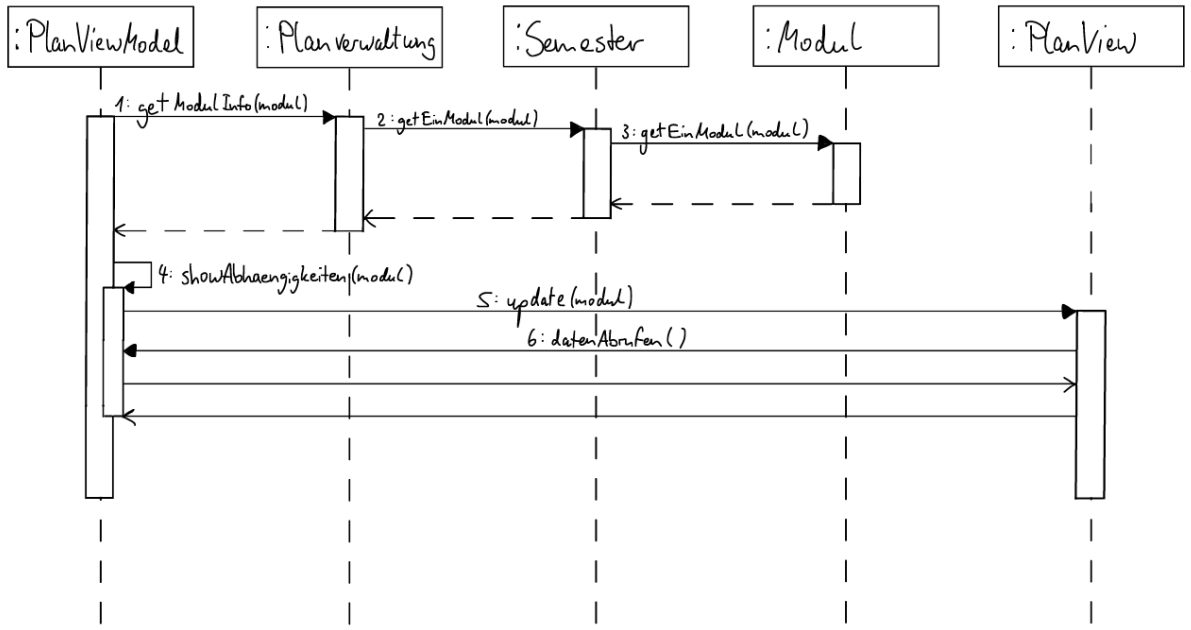


Abbildung 11: Sequenzdiagramm zur Anzeige der Abhängigkeiten von Modulen

Der Studi möchte sich Abhängigkeiten zwischen Modulen anzeigen lassen. Dafür klickt er auf ein Modul. Dabei holt sich das PlanViewModel durch die Klassen durch die Informationen zum angesprochenen Modul (siehe Abbildung 10). Es ruft seine eigene Methode showAbhaengigkeiten auf, welche dann wiederum die View updated. Damit werden die Module farbig hervorgehoben, von denen das ausgewählte Modul abhängig ist.

3.6 Als bestanden markieren Mike Daudrich

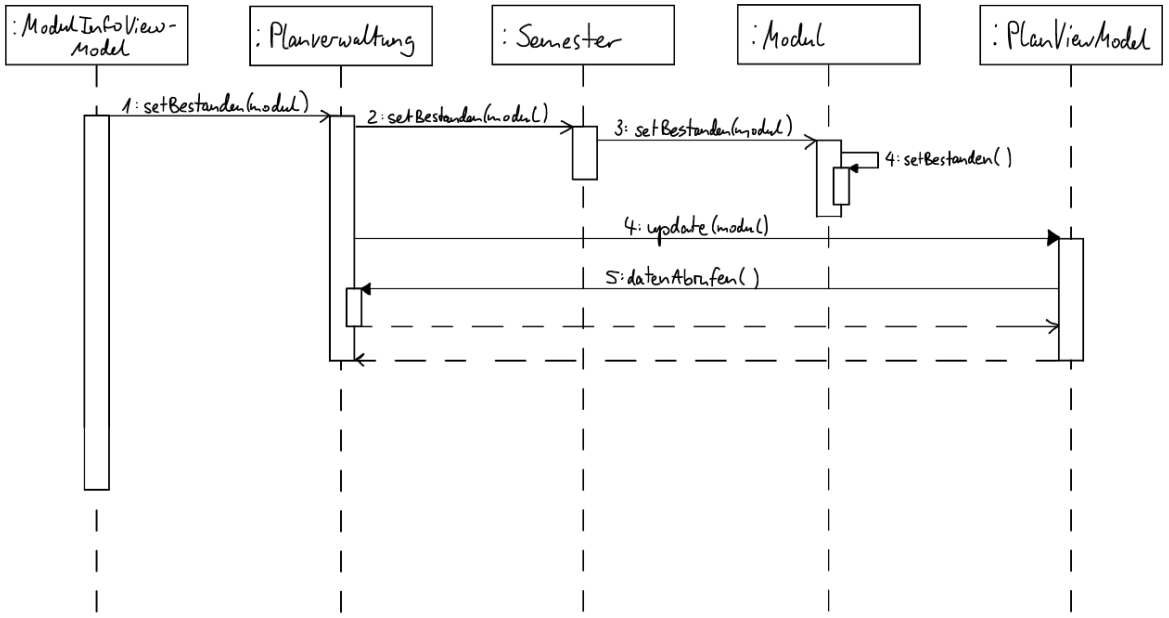


Abbildung 12: Sequenzdiagramm zur Markierung von bestandenen Modulen

Der Studi möchte ein Modul als bestanden markieren. Dafür setzt er in der Detailansicht bei “Bestanden” ein Häkchen (siehe Abbildung 11). Das ModulinfoViewModel löst damit eine Reihe an setBestanden(modul) Methoden aus, welche letztendlich beim Modul eben dieses als bestanden markieren. Die Planverwaltung updated dann auch direkt das PlanViewModel, sodass dann in der PlanView auch das Modul farbig markiert werden würde (siehe Abbildung 7, Anforderungsspezifikationen), sobald der Studi aus der Detailansicht zu der allgemeinen Ansicht wechselt.