# Technical Report on the Data Reference Model v3.6 of the AIStudyBuddy

Sven Judel, Tobias Johnen
Learning Technologies Research Group, RWTH Aachen University
{judel,johnen}@cs.rwth-aachen.de

## 1 Introduction

The AIStudyBuddy project implements modern AI technologies to support students in their individual study planning and reflection, as well as monitoring cohorts and making data-supported decisions about study plan designs. To achieve these aims, various data on study program models and study life cycle data needed to be collected and processed. As these data are managed by different systems in different formats by the project's partner universities, a data reference model was created.

This report presents this model and reasons its structure. It is aimed at supporting employees of universities who need to transform their local data into this model and analysts by describing each attribute's purpose and content. Additionally, techinical specifications and enums for selected attributes, are given in an Open Science Framework project at [7].

## 2 Contributors

Within the joint project, different people contributed to the data reference model. Their names are given below, ordered by their last name:

Katharina Batz
Ruhr University Bochum

Karin Brieger
University of Wuppertal

Joel Emanuel Fuchs
University of Wuppertal

Hayyan Helal
RWTH Aachen University

Tobias Johnen
RWTH Aachen University

Sven Judel
RWTH Aachen University

Christian Metzger
Ruhr University Bochum

Dorian Naujokat
University of Wuppertal

Peter Pultke
University of Wuppertal

Christian Rennert
RWTH Aachen University

Sonja Sokolovic
University of Wuppertal

Miriam Wagner
RWTH Aachen University

Jürgen Wittsiepe
Ruhr University Bochum

## 3 Purpose of the Data Reference Model

The three universities, joining in to the AIStudyBuddy project, use different systems to manage their students' data. These systems use different models to store data, such that a direct exchange of data dumps from the different systems was not suitable nor a merge for joint analyses possible. Instead, a common model was created in which each partner should transform their local data into, to exchange them, and later store them together in a data warehouse. To serve the needs of curriculum analytics and

enable rule-based AI to check study plans and make recommendations to students, the models consist of study life cycle data as well as models of the universities' study programs.

# 4 Model Description

The data reference model (DRM) was created in an iterative way. At the beginning of the project, modeling study programs and study life cycle data (LCD) was split up between working groups to work in parallel and get first results earlier.

Different intermediate models were created and used by other working groups within the project, while a dedicated working group targeted a final alignment and merge of the two models. In the following, the model in its final version 3.6.0 is presented. It allows modeling of all focused study programs from each university as well as the LCD that occur in them. The DRM is modeled in an object-centric approach to be stored within a MongoDB. However, due to the fact that all source systems for LCD were relational, this part of the model, containing the *Personal Data*, *Enrollment Data* and *Achievement Data* (Figure 1 bottom), is kept relational, although it is still stored in a MongoDB.

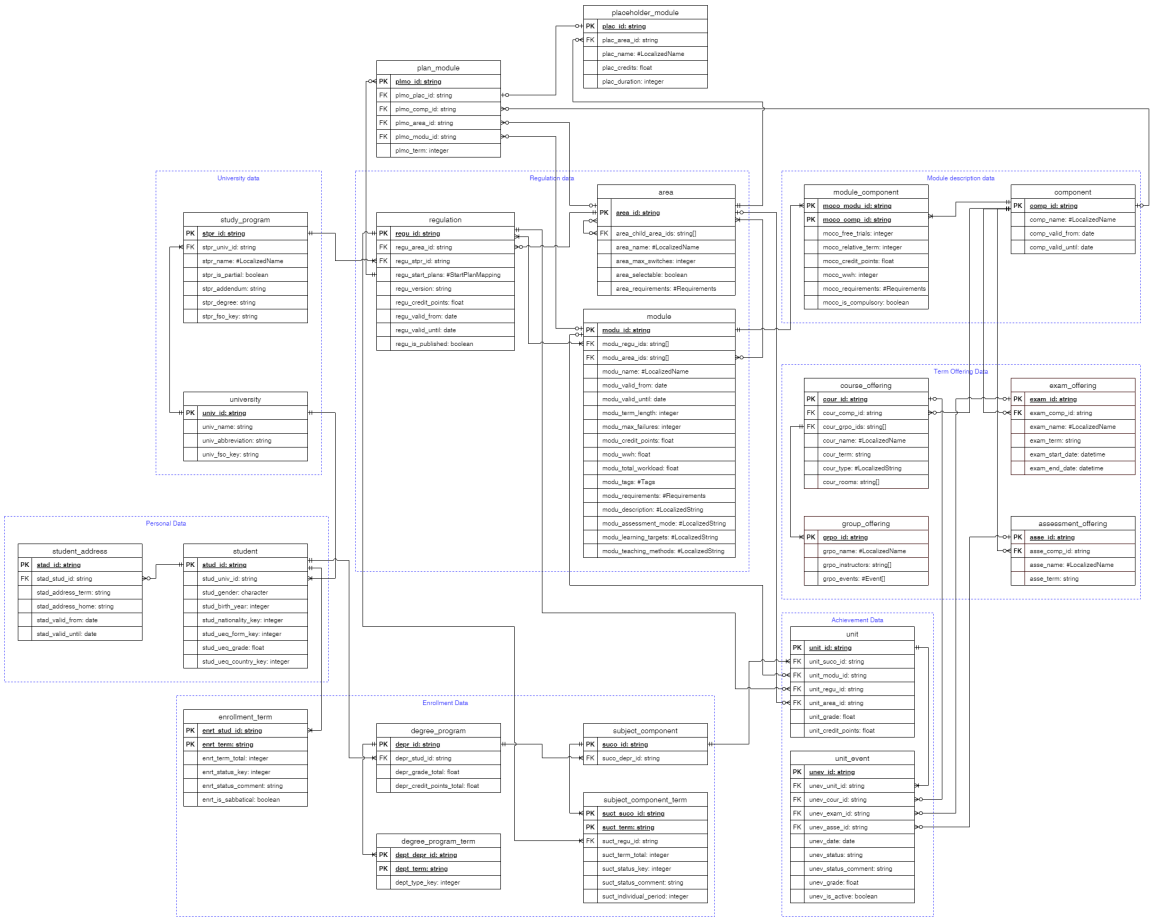


Figure 1: The complete data reference model in version 3.6

## 4.1 Preliminary Remarks

Some decisions about the model's design impact different aspects of it and cannot be introduced in a section dedicated to only one part or entity of the model. Therefore, these decisions are given here:

- The model was created within a research project, collecting a broad range of analysis interests and needs for insights. In this context, data protection requirements for the use of the data to operate

the application are met. If necessary, participating universities must create the legal requirements within their own legislative competencies.

- Although all partner universities structure their study year in semesters, the model uses *term* as a general notion to enable the usage of semesters, trimesters, quarters and other structures, if given.
- For each entity, a 4-letter abbreviation is set and used as the prefix for each of its attributes. This ensures uniqueness of each attribute and eases communication in contrast to having an `id` attribute for each entity. The abbreviations are also used within foreign key attributes. E.g., the `stud_univ_id` attribute is part of the entity *student* and references the `univ_id` of the entity *university*.
- All dates are formatted either as *YYYY-MM-DD* or just *YYYY-MM* as it was sufficient for the project's scope and concrete datetimes were sometimes not retrievable from the source systems.
- For some attributes, possible values are defined within an enum (see [7]). As this model was developed within the context of German universities, these values are given in German only.

## 4.2 Data Sources

LCD can be extracted from the campus management systems, examination management systems and student information systems of the universities. The structures that define the legal framework for the implementation of a study program are usually only available within the business logic of the respective system. This makes automatic extraction difficult or impossible. Therefore, examination regulations and module handbooks, which are often written in natural language, must be transformed manually. To do so, a special editor for the study program data was created.

In the following, each semantic group and its entities, as well as the separated university are described in detail.

## 4.3 University

The *university* is the only entity which data is not derived from the universities' systems or the study program editor. Instead, its content must be entered by an administrator of the data warehouse. This consists of the university's name, an abbreviation and an fso key, referencing the university's *Hochschulnummern*, used by the Federal Statistical Office (FSO), see [3]. It is possible to use the fso key as the ID of the university. However, there are cases in which the fso key can change (e.g., split of Universität Koblenz - Landau [2]). To ease or overcome migrations, such semantic connections of IDs and entity attributes should be avoided. Internally, the `univ_id` is also used for configurations of other software components and, e.g., linked to authentication tokens.

## 4.4 Study Program Modeling

The entities of the study program modeling serve as the foundation for the project. It represents the study programs that can be planned in the StudyBuddy application and the basic rules for the rule based AI. Further, the study life cycle data, that is analyzed and which results are displayed in BuddyAnalytics, has references to the entities that are presented below.

### 4.4.1 Study Program

A study program is the root for the complete model and consists of an identifier (`stpr_id`) and a reference to the university it is offered at (`stpr_univ_id`). Although study programs like computer science are offered at various universities, they differ in multiple ways, including regulations and offered modules, such that each study program for each university has to be modeled separately. However, the fso key (`stpr_fso_key`) is given to detect similar study programs. In Germany, these keys are defined for

different levels (universities, states, country). To unify these definitions and overcome having different study programs assigned the same key, an own list of allowed keys was created by the working group, which is given in [7]. The name of a study program (`stpr_name`) is given as a *LocalizedName* (see Section 4.6) to have the localized names for each supported language, currently English and German. The `stpr_is_partial` attribute states if a study program is part of a bigger study program, e.g., combinatorial study programs at BUW[1] or the Teacher Education Programs at RWTH[2]. Some study programs might have an addendum to their degree or name, e.g., Teacher Education Programs (*Lehramt fuer Gymnasium/Gesamtschule*) that can be stated in the `stpr_addendum`. The earned degree when completing the study program is given in the `stpr_degree` and a list of possible values stated in [7].

### 4.4.2 Regulations

Examination regulations are modeled in a separate entity, as for each study program, multiple regulations can apply at the same time. Further, historic versions would be lost if regulations would just be an attribute of the study program entity. Therefore, a regulation consists of an identifier (`regu_id`) and a reference on the study program it applies for (`regu_stpr_id`). As version identifiers might overlap between study programs at the same university, it is not used as the identifier of an entity object but set in the attribute `regu_version`. The number of credit points students need to earn is set in the `regu_credit_points`. To represent the validity, the attributes `regu_valid_from` and `regu_valid_until` are provided. A study program references a *root area* (`regu_area_id`) that contains all it's subareas and modules. This entity is described in detail in the next subsubsection. To model different default study plans, the `regu_start_plans` holds a *StartPlanMapping* (see Section 4.6) which plan modules are presented in Section 4.4.5. Finally, the `regu_is_published` states if a regulation model is finished. This attribute is purely for internal management to not show regulations via any client but the study program editor, that are still edited and, i.e., miss modules or other configurations.

### 4.4.3 Areas

Areas provide structure within a study program regulation by grouping modules and other areas. This structure can be found in different study programs and might also be used in rules (e.g., in RWTH's Master programme in Computer Science, students have to achieve at least 12 ECTS within the area of theoretical computer science). Each area has an identifier (`area_id`) and a localized name (`area_name`). Utilizing the advantages of a document-oriented database, child areas are set within the single attribute `area_child_area_ids` as a list of IDs. The lists of child areas of two different areas do not have to be disjoint, one area can be the child of multiple others. If an area does not hold any mandatory modules, that students have to take, it can be flagged as selectable (`area_selectable`). Linked to selectable modules is the number of allowed switches (`area_max_switches`) between the selected and deselected state of an area. Finally, potential requirements, linked to an area, can be set as the respective complex data type (see Section 4.6) in the `area_requirements`.

### 4.4.4 Modules

Modules also consist of an identifier (`modu_id`), a localized name (`modu_name`) and a validity time span (`modu_valid_from` and `modu_valid_until`). In contrast to areas, having their child areas as their own attribute, modules reference their parent areas (`modu_area_ids`). This slightly inconsistent modeling is due to already established models within different work packages that contributed to the data reference model. Changing these models would have resulted in extensive reworks, while the functionality of the DRM is not harmed by keeping it. Therefore, this inconsistency was kept. Additionally,

[1] `https://www.uni-wuppertal.de/de/studium/studiengaenge/detail/ma-kombinatorischer-studiengang-kombi/`, last access: 21.03.2025

[2] `https://www.lbz.rwth-aachen.de/cms/LBZ/Studium/~qqwo/Rund-ums-Lehramtsstudium/?lidx=1`, last access: 15.04.2025

the list of regulations, the module is part of, is given in the module itself (modu_regu_ids). These IDs could be computed by going up the nested areas, but this operation can become extensive such that these values are set explicitly. For which area and regulation a student took a module will be stated in the *unit* entity, described in Section 4.5.5. Further, the number of terms a module endures is set in the modu_term_length and the maximum number of failed exam attempts in modu_max_failures. If there is no limitation on the number of failures, the attribute should be set to *999*. This value is considered big enough to cover a reasonable number of failed attempts during the time a module is valid. The number of credit points students earn when passing the module is set in modu_credit_points. The related weekly working hours (this name is used to overcome term related notions like *Semesterwochenstunden*) are set in the modu_wwh and the workload in hours in the modu_total_workload. Tags and Requirements (see Section 4.6) can be set in modu_tags and modu_requirements. Finally, four textual elements from the module handbook can be set, all as *LocalizedString*s: The module description (modu_description), the assessment mode (modu_assessment_mode), the learning targets (modu_learning_targets) and the teaching methods (modu_teaching_methods). Although not utilized within the project's duration, text mining can be applied on these attributes to further investigate a module's content or potential links of students' course preferences or grades and, e.g., assessment modes. Keeping this information up to date creates additional effort, universities must manage. As this is not required by the end of the project, these attributes are optional and therefore nullable.

### 4.4.5 Plan Modules

While the *module* entity represents concrete modules and the areas and regulations in which they can be taken by students, the placements of some of these modules in the default study plan of a regulation (if given) is set by a *plan_module*. This entity maps a module, module component or placeholder to a specific term. It consists of an ID plmo_id and a reference to the specific *area* in the attribute plmo_area_id. There should either be a reference to a specific module through plmo_modu_id or to a placeholder plmo_plac_id. This mutual exclusion requires one of these attributes to be null. In case a specific component is placed differently than the moco_relative_term suggests, the plmo_comp_id reference can be set. Component references also always require plmo_modu_id to be set. Besides the references, the *plan_module* saves the term in which the module or component is planned in the attribute plmo_term. It is stored as an integer, which represents the offset from starting the plan to taking the corresponding *plan_module* item.

### 4.4.6 Placeholder Modules

In cases where the study program allows for a choice between modules or has a set amount of credits in a specific area with variable amounts of modules, placeholder modules are required to display default study programs with the correct amount of credits and workload, as well as giving students a better overview over the number of modules in a term. The *placeholder_module* entity consists of an id plac_id and a required reference plac_area_id to the *area* entity. The other attributes are a name plac_name, the amount of credits plac_credits and the duration plac_duration.

### 4.4.7 Module Description Data

Every module consists of different components. For example, the module "Programming" in a computer science study program might have the components "Exam", "Lecture" and "Exercises". If the module should be reused in different study programs (with different instances in the *module* entity), some way of checking if the components are the same is required. This can be used, e.g., to compare grades achieved in the same module between different study programs and the modules that were taken before. To account for that, two entities for components exist. The *component* entity is used to have the same components for every single regulation. It consists of an ID (comp_id), a name (comp_name) and the time range (comp_valid_from and comp_valid_until), in which the component is valid.

To connect modules to components, the entity *module_component* is implemented. Multiple *module_component* instances can be connected to the same *component*. The main purpose is to store the specific rules a component has for a module in a regulation. This allows the rules for components to differ, depending on which exact module the student takes. As every *module_component* is connected to exactly one *component* and exactly one *module*, its primary key is composed of the references `moco_modu_id` and `moco_comp_id`. In case of exams, `moco_free_trials` store the maximum number of trials for the component. `moco_relative_term` stores the offset from start of the module to start of the component, which is relevant for modules that span over multiple terms. `moco_credit_points` are the credit points the specific component contributes to the module. `moco_wwh` are the weekly working hours required for the component and `moco_is_compulsory` flags a component as required or not required for passing the related module. `moco_requirements` store all other rules about the component (see Section 4.6).

## 4.5 Students' Life Cycle Data

Life cycle data (LCD) represents the data students create while studying. This includes their term-wise enrollments for study programs and single modules, exam registrations and received grades.

### 4.5.1 Students' Personal Data

Students' personal data is stored in two entities: *student* and *student_address*. A student consists of an identifier (`stud_id`), a reference to the university he or she is enrolled in (`stud_univ_id`), a key for the gender (`stud_gender`, e.g. *f* for female), the year of birth (`stud_birth_year`) and a nationality key (`stud_nationality_key`). Regarding the student's university entry qualification (ueq) a key for the form (`stud_ueq_form_key`, e.g., *Gymnasium* or *Berufsoberschule*), the grade (`stud_ueq_grade`) and a key for the country it was received in (`stud_ueq_country_key`) have to be provided. The country key is an integer from 1 to 3, mapping a student's nationality on either *Germany* (1), *EU and UK* (2) or *Other* (3) [7], creating a compromise of the target users' desire for insights and data protection.

Theoretically, students can be enrolled at multiple universities at once or do a consecutive master at another university. In practice, detecting multiple enrollments or changing the university cannot be done automatically and would require a student to confirm his or her identity in the received data dumps of each university. Further, these cases are considered to occur so rarely that an analysis of such cases does not create meaningful results that can not be used to provide feedback to students. Therefore, a separate entity to model students' enrollments was omitted. If intended for future extensions of the model's scope, such an entity could be reintroduced. In this case, all references on the `stud_id` in the entities *enrollment_term* and *degree_program* have to be adjusted to the enrollment ID.

While the attribute values within the *student* entity documents are expected to stay put, a student's address might change during the time of study. Therefore, the addresses are modeled in the separate entity *student_address*. It consists of an identifier (`stad_id`) and a reference to the student (`stad_stud_id`) whose address is modeled. Two address strings, consisting of the country key and the first three digits of the postal code, for the term and home address can be set as well as two dates, stating the time period that an address is valid in. This data is included as the question arose whether the distance of a student to a non-distance university has an impact on study progress. For some universities, students can state two addresses to refer to the addresses on which they stay during a term and outside of it. Therefore, the two different address attributes are included. Finally, both attributes, to mark the valid time span of an address, have to have values. The currently valid time span can be identified by the `stad_valid_until` attribute being set to the default value *9999-12*.

### 4.5.2 Enrollments

The state of a student's enrollment is modeled term-based within the *enrollment_term* entity. This allows tracking terms with a regular enrollment, sabbaticals and terms without an enrollment. The primary key consists of the combination of the student ID, the enrollment is tracked for (`enrt_stud_id`) and the encoding of the term (`enrt_term`). Besides sabbaticals, there might be multiple other states of non-regular enrollments, e.g., terms abroad, but due to data protection, all these states are mapped on *sabbatical* and represented in the boolean attribute `enrt_is_sabbatical`. Besides that, an enrollment has a status of predefined values, e.g., *Haupthörer*in* or *Zweithörer*in*, represented by defined keys (see [7]). If desired, a comment on this status can be set in `enrt_status_comment`. Finally, the total amount of terms a student was enrolled at a university, including the one the entry is referencing (meaning the value can not be 0 or lower), is given as `enrt_term_total`. This is done as the local management systems of all partners were able to provide this value, enabling to use it as a potential value to check for missing data.

### 4.5.3 Degree Program

The enrollment for a study program is modeled via the *degree_program* and *subject_component* entities and their related term-wise ones. A degree is linked to a student (`depr_stud_id`) and stores the current total grade (`depr_grade_total`) and credit points (`depr_credit_points_total`). These two values are given, similar to the `enrt_term_total`, because of their availability in the source systems and their potential complex calculation.

The type of degree is also tracked term-wise in the *degree_program_term* entity. For these types, stored in the `dept_type_key`, a key list is given in [7]. It consists of the regular *Erststudium* and *Zweitstudium* (see [1, 4, 5, 6]) as well as more specific ones (*Konsekutives Masterstudium*, *Aufbaustudium*, *Promotionsstudium*). Depending on a university's internal model, some of the specific values might not be given explicitly but grouped in another one. This needs to be considered when interpreting analysis results later in time. Similar to the *enrollment_term*, the ID of the degree program (`dept_depr_id`) and the term representation (`dept_term`) are used as the primary key.

### 4.5.4 Subject Component

Linking an enrollment and the study program is managed by the *subject_component* entity. While the enrollment and related data is managed term-wise in the *subject_component_term* entity, the *subject_component* serves purely as a link between the term-wise data, the degree program and the related units from the achievement data (see Section 4.5.5). Therefore, it only consists of its own ID (`suco_id`) and the reference of the related degree program (`suco_depr_id`). This link is especially necessary when more than one subject is related to a degree program, e.g., for combinatorical study programs that consist of two or more partial study programs (see Section 4.4.1). While each partial program is modeled at its own element in the *study_program* entity, the combined study of them is represented by having a subject component for each and linking them to the same degree program.

Following that, the *subject_component_term* provides the details on the study program enrollment. Instead of referencing an ID of the *study_program*, the `suct_regu_id` references a regulation, as this might change during the study. An inspection of the partners' management systems shows that if students change the regulation, data related to the achievements is reassigned to the new regulation. This means that historical data is hard to detect (e.g., by using exam dates and a lot of study program specific context knowledge) or completely lost. Transforming local data into this data reference model and keeping the latest version locally for comparison enables the detection of such changes and also keeping the historic data, which also benefits some intended analysis within the project. Next, the `suct_term_total` stores the number of terms enrolled for the related (partial) study program, including the modeled term. The `suct_status_key` tracks the detailed enrollment state, including values for various exmatriculation reasons or completing a study program [7]. If desired, a comment on the state can be set in the

suct_status_comment. Finally, if a student is assigned an individual period, deviating from the official standard period for a study program, it can be set in the suct_individual_period.

### 4.5.5 Achievement Data

Achievement data, referring to achieved grades and credits, are split up into two entities, *unit* and *unit_event*. A *unit* represents an achievement on the level of a module. Therefore, every unit has a reference to the *module* entity with the foreign key unit_modu_id. Besides its own primary key unit_id, it also references the *subject_component* to which the achievement belongs (unit_suco_id). The reference to the *regulation* (unit_regu_id) is needed, as modules can be part of multiple regulations, but it is also required to know the exact regulation. It is not possible to use the *subject_component_term* reference to the regulation, as modules can have a duration spanning over multiple terms, making it hard and sometimes impossible to find out the exact term to use when a student changed the regulation during the duration of the module. It also includes an optional reference to the *area* entity (unit_area_id), as modules sometimes are part of multiple areas in a regulation. The reference is optional, because some universities do not have the information until it is calculated for the graduation of a student. The grade for the module is stored in unit_grade and the awarded credit points in unit_credit_points. The later attribute is necessary, as the credit points awarded might differ from the general credit points, stored in the *module* entity, e.g., because of overshooting the credit point boundaries of the regulation.

A *unit_event* represents a single event that occurs in the context of a single unit, but on the level of components or offerings, e.g., registering for or passing an exam. It consists of an ID unev_id and foreign keys to its unit as well as the possible term offering data (see Section 4.5.6) it is linked to. The foreign keys unev_cour_id, unev_exam_id and unev_asse_id are all optional, but usually, exactly one of them should be set. If none of them is set, no connection to the upper part is possible, and probably the unit's unit_modu_id will also be null. The date on which the event occurred is stored in unev_date. The kind of event is stored in unev_status. This attribute can hold arbitrary values but to ease the creation of analyses, some values, e.g. "Bestanden" for passing an exam, are predefined (see [7]). The unev_status_comment attribute can be used to add more context to the event. The grade is stored in the attribute unev_grade. Finally, the attribute unev_is_active is set to true if the given unit event and its grade is the currently active one for a unit. This can change, e.g., after an exam review and a regrading. Such events do not override a unit event but only outdate them by setting their is_active attribute to false and creating a new one which becomes the active one. However, the unit_grade within the related unit event is overridden. This differentiates the unit events from classical event logs which single entries do not change over time. The concept in the DRM arose from the way the investigated universities' management systems represent changing grades internally.

### 4.5.6 Term Offering Data

Term offering data refers to entities, where new data is required for every term. For example, exams and courses have new dates every term and therefore, need to be modeled in a way where the core module and component data does not need to be updated. As courses and exams are held for specific components and not the module in its entirety, all offerings are connected to components (see Section 4.4.7). There are four different entities to differentiate the kind of offering. The *exam_offering* entity consists of an ID (exam_id) and the foreign key to the component table (exam_comp_id). The name of the exam is stored in exam_name, the term in which the exam is hold in exam_term and the exact start and end times in exam_start_date and exam_end_date as datetimes.

The assessment_offering entity is used to model assignments that can also lead to credits being awarded, but are worked on by students at home and not on a fixed date. Therefore, the entity also consists of an ID asse_id, a reference to a component asse_comp_id, a name asse_name and the corresponding term asse_term.

Course data is modeled by the entities *course_offering* and *group_offering*. Two entities are needed, as, e.g. universities use course data to group seminars on different topics together. For example, the

module ”Seminar Computer Science” can have different, topic-related offerings (e.g. ”Advanced Topics in Cryptography” or ”Selected Topics in Technology Enhanced Learning”). The representation will consist of a single *course_offering*, for the module, and multiple *group_offerings*, one for each topic. For modules with just a single course, there will be exactly one group offering. The *course_offering* entity consists of an ID `cour_id`, a reference to a component `cour_comp_id` as well as a list of references to *group_offering* instances (`cour_grpo_ids`). It also consists of a name `cour_name`, the term `cour_term`, the type of the course (`cour_type`, e.g. lecture or exercise) and a list of rooms `cour_rooms`. The *group_offering* includes an ID `grpo_id`, a name `grpo_name`, a list of (general, not event related) instructors `grpo_instructors` and a list of events (see Section 4.6) occurring for the offering (`grpo_events`).

## 4.6 Complex Data Types

While most attributes are basic data types (integer, float, string and boolean), some require matching patterns (e.g., dates as strings in the format of *YYYY-MM* or *YYYY-MM-DD*), others even require data structures. These structures are provided by the *complex data types* that are described in this subsection. Each data type is a document-like structure itself with keys and values of different basic or other complex data types.

### 4.6.1 `LocalizedString`

A *LocalizedString* represents the translation of a field’s content into different languages. The current version requests a German and English translation, represented by the fields `de` and `en`.

```
{
  "de": "(string) German translation",
  "en": "(string) English translation"
}
```

*Example:* The translation of the name of the study program Computer Science Bachelor would look like this:

```
{
  "de": "Informatik Bachelor",
  "en": "Computer Science Bachelor"
}
```

### 4.6.2 `LocalizedName`

The *LocalizedName* provides a name translation not only in different languages (using *LocalizedStrings*) but also enables the provision of an optional `short_name`, e.g. if a module name has an established or official abbreviation.

```
{
  "complete_name": "(LocalizedString) The complete name in English
     and German",
  "short_name": "(LocalizedString) The optional short name in English
      and German"
}
```

*Example:* The translation of the name of the study program Computer Science Bachelor would look like this:

```
{
  "complete_name": {
    "de": "Informatik Bachelor",
    "en": "Computer Science Bachelor"
  },
  "short_name": {
    "de": "Info B.Sc.",
    "en": "CS B. Sc."
  }
}
```

#### 4.6.3 `Event`

The *Event* type represents events that can happen within the academic context. Its only required field is the date, containing the datetime of the event. All other fields, listed below, are optional.

```
{
  "date": "(datetime) The datetime of the Event",
  "duration": "(int) Optional: The duration of the Event in minutes",
  "room": "(string) Optional: The room the Event took place in",
  "instructor": "(string) Optional: The name of the instructor of the
      Event",
  "hybrid": "(bool) Optional: States if an Event took place both in
     person and online",
  "compulsory": "(bool) Optional: States if the Event was mandatory
     for students to attend",
  "recording": "(bool) Optional: States if the Event was recorded"
}
```

*Example:* The first event of a typical lecture in the winter semester 2024 would look something like this:

```
{
  "date": "2024-10-14 12:00",
  "duration": "120",
  "room": "FL.00.01",
  "instructor": "Univ.-Prof. Dr.-Ing. Tobias Meisen",
  "hybrid": "false",
  "compulsory": "false",
  "recording": "true"
}
```

#### 4.6.4 `Requirements`

*Requirements* refer to a module, a module component, or an area and contain the various constraints to respect to start and finish it. Further, it lists various flags, like a study goal it is related to, can be given and a natural language description of constraints that cannot be modeled by the used constraint representation. E.g., a literature research course, required for a seminar, done by the university's library that hands out printed documents to certify successful participation.

```
{
  "start": "(string) The constraint to respect before being allowed
     to start with a module, a module component, or an area",
  "finish": "(string) The constraint to respect to complete a module,
      a module component, or an area successfully",
  "flags": "(list[string]) List of flags related to the requirement",
  "other": "(string) A natural language description of special
     requirements"
}
```

*Example:*

```
{
  "start": "{'Mathe'} & {$[CS]>=120}",
  "finish": "",
  "flags": ["allow_auto_completion", "area_by_student"],
  "other": "Theoretical knowledge is required"
}
```

This example states that before starting with the module that has these requirements, the module *Mathe* should have been done and at least 120 credit points in the area *CS* should have been collected. Further, the example assumes that the Campus Management System cannot provide success results of this module; therefore, students are allowed to mark modules as completed by themselves, such that further planning takes it into consideration. Another relevant flag is *area_by_student*, where the student must choose the area of modules with multiple areas. If this flag is not used, the area will be chosen by the solver. Finally, the `other` field allows for adding requirements in natural language, if it does not fit in the model. These won't be taken into consideration by the solver, but allow students to be at least aware of them.

#### 4.6.5 `Tags`

The *Tags* data type represents a list of tags, categorized by `types`, `languages` and `other`. They enable referencing a group of modules. `types` tags refer to the type of a module (course, seminar, ...) and the `languages` tags to the languages a module is offered in. `other` tags can be used for additional tags, e.g., *math-related*.

```
{
  "types": "(list[string]) Tags defining the types of a module",
  "languages": "(list[string]) Tags listing the languages a module is
      offered in",
  "other": "(list[string]) Further tags not related to a module's
     type or language"
}
```

*Example:* The module of a seminar within the area of theoretical computer science, held in English, can have the following tags:

```
{
  "types": ["Seminar"],
  "languages": ["English"],
  "other": ["Theoretical"]
}
```

### 4.6.6 `StartPlan`

While all previously presented data types provide a fixed set of fields, the *StartPlan* can have arbitrary keys.

A *StartPlan* object represents one or multiple possible study plans, representing a certain study aim (as the key) with a tuple of a translation as a *LocalizedString* and the associated *plan_modules* (see Section 4.4.5) as the value.

```
{
  "(string) Key for the study aim": [
    "(LocalizedString) German and English representation of the study
       aim",
    "(list[string]) List of associated plan module IDs"
  ]
}
```

*Example:* The definition of a full-time study program plan, next to one for part-time, can be represented by:

```
{
  "fulltime": [
    {"de": "Vollzeit", "en": "Full-time"},
    [/* List of associated plan_module IDs */]
  ],
  "parttime": [... ]
}
```

### 4.6.7 `StartPlanMapping`

The *StartPlanMapping* defines a mapping of term part keys on *StartPlans*. Term part keys refer to the instances a term can have. E.g., *summer* and *winter* for semesters and *autumn*, *winter* and *spring* for trimester.

```
{
  "(string) Term part key": "(StartPlan) The related start plan"
}
```

*Example:* The definition of different start plans for winter and summer semester as well as full-time and part-time students can look like this:

```
{
  "summer": {
    "fulltime": [
      {"de": "Vollzeit", "en": "Full-time"},
      [/* List of associated plan_module IDs */]
    ],
    "parttime": [
      {"de": "Teilzeit", "en": "Part-time"},
      [/* List of associated plan_module IDs */]
    ]
  },
  "winter": {... }
}
```

# References


[1] SGV § 4 Vergabe von Studienplätzen für ein Zweitstudium — RECHT.NRW.DE, 2019. https://recht.nrw.de/lmi/owa/br_bes_detail?sg=0&menu=0&bes_id=41405&anw_nr=2&aufgehoben=N&det_id=529402, Last access: 01.04.2025.

[2] Stellungnahme zur Entscheidung des Ministerrats zur zukünftigen Struktur der Universität Koblenz-Landau, 2019. https://www.uni-koblenz-landau.de/de/aktuell/archiv-2019/stellungnahme-zur-entscheidung-des-ministerrats-zur-zukuenftigen-struktur-der-universitaet-koblenz-landau/index.html, Last access: 06.02.2025.

[3] Schlüsselverzeichnisse für die Studenten- und Prüfungsstatistik, Promovierendenstatistik und Gasthörerstatistik WS 2023/2024 und SS 2024, 2023. https://www.it.nrw/system/files/media/document/file/nrw_schlusselverzeichnis_definitionenkatalog_studierende.xlsx, Last access: 06.02.2025.

[4] Second Degree Applicants, 2025. https://studium.ruhr-uni-bochum.de/en/second-degree-applicants, Last access: 01.04.2025.

[5] Special Admission Process - RWTH AACHEN UNIVERSITY - English, 2025. https://www.rwth-aachen.de/cms/root/studium/vor-dem-studium/bewerbung-um-einen-studienplatz/~djkz/sonderantraege/?lidx=1, Last access: 01.04.2025.

[6] Zweitstudium, 2025. https://www.studierendensekretariat.uni-wuppertal.de/de/bewerbung-und-einschreibung/bewerbung/zweitstudium/, Last access: 01.04.2025.

[7] AIStudyBuddy Data Model Working Group. AIStudyBuddy Data Reference Model v3.6, 2025. https://doi.org/10.17605/OSF.IO/YVN8C, Last access: 17.04.2025.