# Modeling Financial, Project and Staff Management

## A Case Report from the MaCoCo Project

Arkadii Gerasimov[a], Peter Letmathe[b], Judith Michael[a], Lukas Netz[*,a], Bernhard Rumpe[a]

[a] RWTH Aachen University, Software Engineering, Aachen, Germany
[b] RWTH Aachen University, Management Accounting, Aachen, Germany

Abstract. *To obtain more financial freedom, universities and especially their chairs and institutes have to establish a well functioning and reliable financial management and accounting system. Currently, chairs have different technical solutions for these systems, each of which must react individually to external changes and require a high effort to adapt their reports. Thus, they rely on either commercial accounting software, which is not tailored to their specific needs, or standard spreadsheet software making use of complex sheets and cross-references which are error-prone and hard to adapt. We have used domain models and code synthesis methods to create an enterprise information system. This paper shows how models reflect user requirements, evolve with changing requirements and how they impact an agile, model-driven engineering process. The resulting system simplifies the planning of financial management and accounting by university chairs.*

Keywords. Agile Methodology • Generative Methods • Higher Education Reform • Management Cockpit for Chair Controlling • Model-Based Software Engineering

Communicated by Henderik A. Proper, Giancarlo Guizzardi. Received 2022-04-09. Accepted on 2023-08-02.

## 1 Introduction

*Motivation and relevance.* Universities, like any industry sector, must drive the digital transformation of their processes. These processes include teaching, research, acquisition of third-party funding, and administration. Since the 1990s, German universities have been facing additional challenges: They face increased competition due to the modular bachelor and master system leading to a need for increased efficiency (Küpper 2007). Further, the German government began to decentralize management accounting by delegating it to the universities themselves. As a result, in order to obtain more financial freedoms, universities have to establish a well-functioning management accounting system as well as implement financial reporting. The last aspect in particular leads to new challenges for universities and their stakeholders, which include the central university administration, various faculties as well as the chairs and research institutes. The universities' central administration depends on obtaining aggregated financial data from all belonging facilities, which is used to allocate financial resources appropriately. Further, they are obliged to provide annual reports including profit and loss accounts as well as balance sheets. Thus, a reliable software system is a necessity. Most German universities choose to utilize adapted Enterprise Resource Planning (ERP) systems, such as SAP (About SAP. 2022). In contrast, the management accounting of university chairs and research institutes entails the development of plans for long and short-term spending

* Corresponding author.
E-mail. netz@se-rwth.de
Note: This work is funded by RWTH Aachen University.

of, on the one hand, the financial resources provided by the central administration and, on the other hand, the necessary acquisition of third-party funding. Especially the accounting of third-party funds underlies numerous regulations in regard to their settlements and spending.

The processes and tools necessary to cope with chair requirements are not included in universities' ERP systems as these are only designed to fulfill governmental requirements and the needs of the central university administration. Consequently, chairs and institutes rely on (1) commercial accounting software, which is not tailored to their specific needs, (2) standard calculation tools such as Excel which includes complex sheets and cross-references which are error-prone and hard to adapt for new employees, or (3) larger chairs have developed their own software tools which have to be adapted to new requirements on an ongoing basis, and (some of which) are too technologically outdated to be used as a basis for all chairs. To simplify the planning of financial management and accounting for chairs, a software solution is needed. This article sheds light on the required software engineering methodologies needed to develop a solution that supports the planning, revision, and governance of management and accounting.

Addressing these challenges, this paper is guided by the following research question:

- *What concepts are relevant to reflect the management domain of organizational units of universities?*

*Contribution.* The main contributions of this article are (1) the description of the process from requirements elicitation to iteratively changing and growing domain models, (2) the domain models themselves, and (3) the system generated from them. Within the MaCoCo project at RWTH Aachen University, we are developing a system for the chairs' financial management accounting and planning since 2016. We apply a Model-Driven Software Engineering (MDSE) approach using the language workbench and code generation framework MontiCore (Krahn et al. 2010, Hölldobler and Rumpe 2017). This allows us to reduce the effort of creating and changing handwritten code by using models and code synthesis for continuous regeneration. The used domain models are created with UML/P (Rumpe 2016) inspired Domain-Specific Languages (DSLs), which is a language family better tailored for programming. Moreover, we use agile development methods to handle changing requirements. This case report is authored by the main developers of the MaCoCo application.

*Structure.* The next section motivates the need for chair financial management accounting and planning systems and presents the current financial structure of RWTH Aachen University. Sect. 3 introduces the MaCoCo project, relevant stakeholders, and the development process. Sect. 4 presents the DSLs used in MaCoCo and the generator used to create the resulting system. Sect. 5 describes the domain models as well as the process towards these models and design decisions. Sect. 6 describes the resulting application and Sect. 7 discusses the return on modelling effort and lessons learned. The last section concludes.

## 2 Motivation

Universities and especially chairs are facing challenges regarding the digital transformation of their financial management (Brdesee 2021; Küpper 2007). In the following, we show what influences them and provide insights into the personnel and funding structure of the German university that raises the most third-party funding.

### 2.1 Challenges for Universities

Küpper summarizes the innovations and challenges German universities have been facing since the beginning of the 1990s as the "second German higher education reform" (Küpper 2007, p. 82). He identifies three central objectives driving the changes, namely: (1) strengthening the efficiency of German universities, (2) increasing the competition between universities and (3) decentralizing its economic governance and controlling systems (Küpper 2009, p. 54).

Various changes have been implemented to meet these challenges. These include the introduction of global budgets in order to grant more autonomy to the universities (Ambrosy et al. 1997, p. 204). By cutting back the basic governmental funding and fostering the acquisition of third-party funding within the framework of the "Pact for Research and Innovation" university chairs should in addition be motivated to raise their research funds (Hornbostel 2001, p. 527 ff.).

Furthermore, the courses of study were adapted to international standards by introducing the Bachelor's and Master's systems. The modular structure of the system provides universities the opportunity to individually determine the focus of their courses of study and thus, personalize them. In combination with the introduction of tuition fees, competition between individual universities is stimulated (Müller-Bromley 2011, p. 1f).

In order to be able to meet the new challenges and international competition universities must innovate and acquire suitable instruments, in particular systems for information processing and management accounting (Küpper 2007, p. 82). Information systems which support the planning, revision, and governance of management processes and further offer tools for cost accounting are considered most necessary (Ambrosy et al. 1997, p. 204). Moreover, the higher education autonomy act of the state North Rhine-Westphalia (HFG)[1] grants universities free choice of the underlying accounting standards. Universities can either retain the traditional German fiscal accounting standards called cameralistic standards, or implement the business accounting standards of private enterprises. In return for utilizing business accounting standards which grant more freedom and flexibility, universities have to manage their budgets based on integrated management and accounting, as well as planning and implementing their economic management (HFG §5 Sec. 2). This ensures that the continuous fulfillment of their tasks is manageable.

Research on management accounting in German universities mainly concerned itself with two relationships and governance levels: Firstly, the relation between states and universities. Secondly, the relationship between universities and their faculties. In both cases research focused on the distribution of monetary resources either from the government to the universities or from the universities' board to their faculties (Küpper 2009, p.51). Some studies further focus on the distribution of the faculty's monetary resources to their belonging chairs. Whereas concepts for the management accounting of universities and the above-mentioned allocation problems have been developed since 2007, only little attention has been paid to the chairs and research institutes that acquire the desired third-party funding and execute research projects. This research gap is worth addressing as these entities and their needs can be compared to small and medium-sized enterprises considering the number of employees and financial resources invested in them. Furthermore, as they play a significant role in the universities' total liquidity due to their project acquisition, they play a crucial role in education and research.

## 2.2 Financial Structure of RWTH Aachen University

RWTH Aachen University (Rheinisch-Westfälische Technische Hochschule Aachen) was founded in 1870 and comprises 8 faculties plus the faculty for medicine which is linked to the university's medical center. In winter semester 2020/2021[2] , 47.269 students are enrolled. The students choose from 170 courses of study offered by 553 professors. RWTH Aachen University employed 10.156 annual full-time equivalent staff including 553 professors, 4.734 academic and technical staff paid with governmental resources, 537 trainees, 3.826 staff financed by third-party

[1] Hochschulfreiheitsgesetz (HFG) des Landes NRW from 31. Oktober 2006. https://recht.nrw.de/lmi/owa/br_vbl_detail_text?anw_nr=6&vd_id=1460&vd_back=N

[2] Figures and Facts provided by RWTH Aachen University: https://www.rwth-aachen.de/cms/root/Die-RWTH/Profil/~enw/Daten-Fakten/lidx/1

funding and 506 from special funding. Additionally, 3.092 students and research assistants are employed. Considering the total budget of the university, including the administration departments, but excluding the affiliated institutes, 404 million euros from a total of 1049 million euros were acquired via third- party funding. This leads to a third-party funding ratio of 38.51 %. According to the latest report of the Federal Office of Statistics [3], RWTH Aachen University acquired in comparison to other German universities the most third-party resources in total (331 million euros) and per professorship (985.000 euro).

RWTH Aachen University integrated the ERP System SAP in order to replace the former cameralistic management accounting system HIS. The SARA project, which deals with the transition from the HIS system to the SAP system, began in 2012. After the commissioning of the software in January 2015, it became clear, that there is a gap between the needs of the university's administration and those of the chairs and research institutes. An exploratory study comprising interviews with chairs confirmed this and led to the start of the MaCoCo project.

## 3 The MaCoCo Project

The MaCoCo (Management Cockpit for University Chair Management and Controlling) project started in 2016 with the goal to develop an Enterprise Information System (EIS) for the planning, revision, and governance of management processes as well as the cost accounting of small and medium-sized chairs at RWTH Aachen University. It is a joint project by the chair of Management Accounting from the Faculty of Economics and the chair of Software Engineering from Informatics.

In the first step, the developed software solution was tailored to the needs of small and middle- sized chairs at the university. The phrase *small and middle-sized* does not relate to a certain amount of funding, staff, or accounts (there are no technical restrictions on this level) but to their organizational structure. In contrast, large chairs often have further administrative structures, are more workflow-oriented, and have other needs regarding their financial affairs. They already use systems for accounting and sometimes even workflow systems similar to companies in the private sector. Beyond small and middle-sized, the first large chairs are currently also in the process of transitioning to using MaCoCo which results in a set of additional requirements, e. g., to cover hierarchical structures.

The capabilities and skills to be developed in the EIS are represented by the universities staff. Thus, their expertise and competency need to be continuously included in the development and testing processes. Consequently, an agile software development paradigm was chosen, which strongly involves future users in the conceptualization process.

### 3.1 User Groups

Following a user-centered approach, two groups of people were included in the development process of MaCoCo: (1) A group of *lead users* gave input and helped developing concepts and system functions. (2) The *steering committee* supervised and evaluated the concepts and project progress.

**Lead Users.** The faculties of the university differ not only regarding their research but also in terms of financial instruments and accounting modalities needed. The resulting diverse functionalities require a broad spectrum of lead users covering different perspectives and needs. Therefore, the circle of users consists of representatives from the eight faculties for whom MaCoCo is designed. Further, small to medium-sized chairs and institutes as well as deaneries are selected. Taking the strategic and operational planning as well as the administrative level into account entails the inclusion of people occupying administrative, secretary positions or those in leadership, such as professors or deans. In order to efficiently involve users in the development process, we kept the group size small. To ensure that we had involved a broad range of perspectives, we made sure that

[3] Statistisches Bundesamt Destatis, Figures from 2019: https://www.destatis.de/DE/Presse/Pressemitteilungen/2021/09/PD21_418_213.html

users were spread across different chairs and institutes of varying sizes, faculties, and roles at the institutes.

Moreover, this group can change over the years: When functionalities are completed and new functionalities are developed, new perspectives are needed, for which we attract other roles and chairs, e. g., not only small and medium-sized chairs but also large ones.

Additionally, as MaCoCo is being developed in-house and the created application is also used by the developing chairs, feedback from our own colleagues is also constantly flowing in.

**Steering Committee.** The MaCoCo steering committee is commissioned to monitor the project from diverse perspectives. Each member is entitled to represent values, rules or interests of a group of people. From a financial point of view, the project is supervised by the universities chancellor's representative and the head of the Department for finances. The values and rights of the employees are protected by the *two personnel councils* and the head of the Department for personnel. In cooperation with the *data protection officer* the collected personnel data is reviewed in regard to data security and its permitted analysis and display methods. Further, the information collected by the system during the usage of MaCoCo should not allow tracking activities of a certain user or to evaluate his or her work performance. Considering the fact that MaCoCo is hosted on the server infrastructure of the IT Center and has an aspired interface to the ERP system SAP used by the university administration both, the ITC and SARA project (internal project, which adapts SAP for the university), are represented in the steering committee. Additionally, the head of the department "Organisation and IT", the managing director and dean from the faculty for mathematics, physics, computer science and natural science, a representative and a professor from different computer science chairs are part of the steering committee. To advocate the interests of the lead users, two permanent representatives of the circle of lead users are invited to join the semiannual meetings of the steering committee.

## 3.2 The Software Engineering Process

Within MaCoCo (Gerasimov et al. 2020a), we follow an agile, model-driven software engineering process with an intensive, iterative requirements elicitation process. We have adapted the Scrum (Schwaber 1997) method, an agile, lightweight, and iterative framework to manage product development (Figure 1).

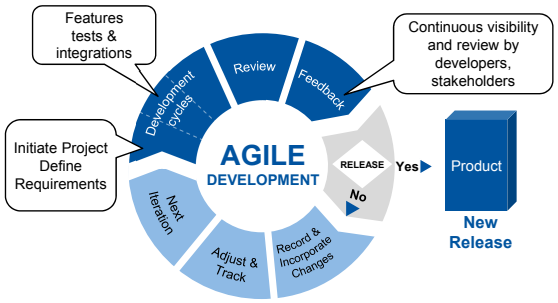



*Figure 1: Agile software development method*

In an initial workshop with all lead users, we collected, evaluated, discussed, clustered, and prioritized general problems, requirements, and functions. We have focused on financial and staff management as they form the basic features of the system. The compendium of these requirements for the product is represented in the *product backlog*. Before each Scrum iteration, the team selects a manageable portion of the product backlog and adds it to the *sprint backlog*. This portion is intended to be completed within one iteration cycle (*sprint*).

After selection, we added details to the requirements for an upcoming sprint and designed the corresponding parts of the system. This included detailed use cases with user interface prototypes and the relevant data structure within the first years of the project. Single users were interviewed to discuss our concepts for user interfaces and how they could interact with the system. Hereafter, these concepts were discussed, validated, and adapted with other users. More complex functionalities needed a longer preparation time, which resulted in a process where the preparation of the functionality details was finalized in one to two sprints

after which their implementation was started in the following sprints.

Once the concepts were stable enough, they were iteratively implemented. This includes, modeling the data structure and user interfaces (domain models) as well as handwritten additions, or more rarely, additions or adaptations of the generator. We are testing the developed product changes incrementally within the implementation and perform manual user interface tests before showing the product to users. The user feedback is collected and a version of the product is either released or kept unpublished for further development. When the target quality is reached, the new functionality can be released. If not, we collect requested changes and create a plan to incorporate the adaptations into our software. The next iteration of the agile process starts by implementing planned changes in multiple development cycles.

Within the development team, we are organizing *bi-weekly meetings* to discuss current issues of the ongoing sprint and to keep track of the overall progress. Code reviews, test-driven development, automated test runs, and quality assessment metrics support the development team.

To assess the long-term needs which go beyond the basic functionalities and to develop the product vision, an exploratory study utilizing expert interviews has been conducted. In this study mainly heads of institutes, professors and deans have been questioned regarding project management and personnel planning. Gaps in the system were identified and discussed, alongside the extent to which the existing SAP system supports the required accounting and planning routines. The participants have expressed wishes for functionalities like importing data, connection to the central SAP system, and time sheets for project management functionalities. For some specific features, we conducted surveys among key users and held several workshops. Moreover, we organize monthly user meet-ups to get direct feedback.

## 4 Models and Languages of the Engineering Process

MDSE is a development approach, which uses models as primary artifacts, from which software is produced using a code generator or other tooling (Hölldobler et al. 2019). The models used in MDSE can be defined in DSLs. Being primary development artifacts, they are treated the same way as code written in a general-purpose language: They are handled using the same IDE as the application code, included in version control, etc. DSLs enable the involvement of domain experts with little experience in the software development process, as the models provide a domain-specific abstracted view on parts of the application (Hesse and Mayr 2008; Stachowiak 1973). The models are not intended to be defined by the user of the application, but to assist the developer in implementing the software.

The models are created during the analysis and design phases or throughout the development of the application. On basis of these models, the complete software is planned, analyzed, and developed. They are used as a specification or guideline for a developer. Additionally, the models are used as documentation for parts of a system, e. g., its behavior and data structure, as they form an abstract description of the intended design. A detailed breakdown between generated and handwritten code can be found in Table 1.

### 4.1 Domain-Specific Languages

A multitude of different DSLs are used to create the MaCoCo application. They are used to describe the exact domain and ease the communication between a stakeholder and a developer (Domain-specific conceptual modeling: Concepts, methods and tools 2016; Völter et al. 2013). DSLs can be developed to describe a generic domain, such as data structures in general, or a very specific aspect of an application like permission management. Defining new DSLs is a challenging and time-consuming task (Frank 2013, Karsai et al. 2009, Michael and Mayr 2015), therefore DSLs should be reused if possible. To tackle the enhanced

tooling challenge and to ensure DSL interoperability (France and Rumpe 2007), we rely on a single language workbench to either define new or reuse existing DSLs. In the following, we introduce the used languages: CD4A for defining data structures, OCL/P for defining data input validation, and GUI-DSL for defining Graphical User Interfaces (GUIs), where each language is defined using the MontiCore language workbench.

CD4A

```
1 package de.macoco;
2 classdiagram Example {
3   class Person {
4     String name;
5     int age;
6   }
7 }
```

*Listing 1: Example for CD4A representing class Person with attributes name and age.*

**CD4A (Class Diagram for Analysis)** is a textual DSL, which enables the definition of UML class diagrams intended for analyses. They are based on UML class diagram (Object Management Group 2017) features and have a Java-like syntax (Rumpe 2016). The DSL supports all common elements of UML class diagrams such as enums, interfaces uni- or bidirectional associations with cardinalities and inheritance. We are using CD4A models to define the domain-specific data structure of the MaCoCo application.

Listing 1 shows the structure of the CD4A language. Line 1 specifies the package name and is similarly handled to the package structure in Java. The keyword `classdiagram` denotes the start, and the name has to match the filename. A class diagram can have multiple classes, interfaces, associations, etc. One such example definition is the `Person` class in line 3. The class has 2 attributes `name` (4) and `age` (5). Visibility modifiers such as `public` can be omitted and a model can be left underspecified. The generator then decides how to handle such cases.

**OCL/P (Object Constraint Language for Programming)** is a Java-like adaption of the OMG OCL language (Object Management Group 2014). The goal of OCL/P is to provide additional restrictions for other models by defining constraints (Rumpe 2016). Within MaCoCo, we use OCL/P to add constraints to classes and attributes in CD4A models. Listing 2 is an example of an OCL/P model which adds a constraint to the CD4A model in Listing 1.

OCL/P..

```
1 ocl ExampleConstraints {
2   context Person inv isAgeNotNegative:
3     age >= 0;
4 }
```

*Listing 2: Example for an OCL/P model based on the class diagram in Listing 1*

Each invariant can have a name (`isAgeNotNegative` line 1) and a context in which it operates. In combination with the CD4A language, it is possible to restrict attributes of classes in a given model. Line 3 references the `age` attribute of the `Person` class. In the example, the age is restricted to non-negative. OCL/P automatically checks for type properties and valid expressions.

**GUI-DSL (Graphical User Interface Domain-Specific Language)** is a domain-specific language for describing graphical user interfaces (Gerasimov et al. 2020b, 2021) in a textual form. GUI-DSL models are used in the MaCoCo project to define web pages and their connection to the underlying data structure. A basic example of a textual model and its corresponding graphical representation within the generated application can be found in Fig. 2.

A GUI-DSL model consists of a web page signature and a body. In Fig. 2, a web page named `Staff` is declared using the keyword `webpage` (line 1), the only parameter included is staff information, which is further used as input for a data table. The parameter itself has an implicit connection to the data structure. The connection is defined using additional class diagrams, in which user data interface classes are created. These can reference domain classes and may contain additional attributes, whose values are loaded using custom hand-written logic.

GuiDSL

```
1 webpage Staff(all StaffOverview
      staffList){
2   card {
3     head {
4       label "Staff Overview"
5     }
6     body {
7       datatable {
8         rows <staffList {
9           column "Name", name
10      column "Birthday", birthday
11 }}}}}
```

Staff Overview

| Name | Birthday |
|---|---|
| Bob | 01.02.1994 |
| Alice | 12.12.1991 |

*Figure 2: Example GUI model*

A major part of a GUIs definition is a description of its layout. The example demonstrates the usage of the built-in card UI component (line 2) with a header showing the "Staff Overview" title (lines 3-5), and a body displaying a table filled with staff information (lines 6-11). Each graphical component is defined in the language including its configuration, e. g., the text of the label, columns of a table, or if a table is editable. In the example, each column (lines 9-10) has a name, which is displayed in the table header, and an attribute name, whose value is displayed in each row.

Events and their handlers are either completely generated or the functionality is implemented manually. The generator produces code describing common functionalities, such as navigation, copying content into the clipboard, as well as default CRUD operations. In more complex cases where provided default functionality does not satisfy requirements, it can be replaced by handwritten extensions.

### 4.2 Generator

The DSLs (CD4A, OCL/P, GUI-DSL) are used to define the model input for the generator (see Fig. 3)(Adam et al. 2018). Models are parsed by a DSL-specific parser that is generated by MontiCore (Hölldobler et al. 2021). The parser converts the textual models into Abstract Syntax Trees (ASTs). This abstract representation can be transformed and extended. In the case of the domain models, each class in the model is transformed into multiple Java classes, (e. g., object builder), and during transformation the OCL AST is used as an additional input to add data validation logic into the generated classes. Once fully transformed, the ASTs are passed on to a template engine for each AST type. Freemarker templates (Freemarker 2022) are used to create target code in the specified programming language. The data structure is generated in Java for the back end and in TypeScript for the front end. The user interfaces are generated as a combination of HTML and TypeScript code. Common code that is independent of the domain models, i. e. the Run Time Environment (RTE) is provided by the framework and not generated each time.

## 5 Domain-Specific Models

The data structure of the MaCoCo project (Gerasimov et al. 2022)[4] is the primary artifact for the generative MDSE approach. It is extended for every use case and contains more than 100 classes at the time of writing.

In the following, we will look at parts of the data structure and discuss their effect on the application. The model consists of four sub-domains.

- **Financial Management**: Classes describing the financial management of a chair including accounts, and their budgets and bookings. This includes public budgets and third-party funding.
- **Staff Management**: Classes used for managing the staff such as persons, their contracts, and salary bookings.

[4] The class diagram of the German implementation of MaCoCo is available as Zenodo artifact.

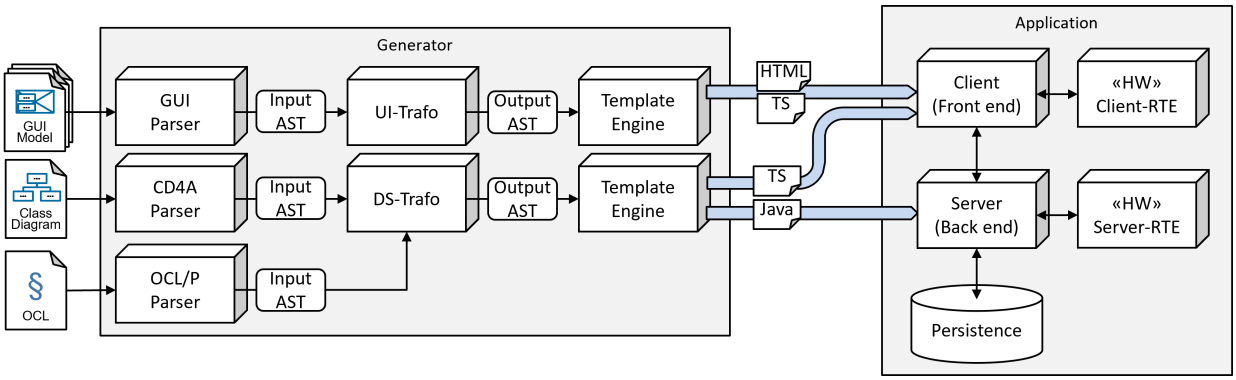



*Figure 3: Simplified MontiGem Generator Architecture. Showing the transformation of textual models into abstract representation (AST) that is further transformed and used as input for different Template Engines, that convert the AST into source code.*

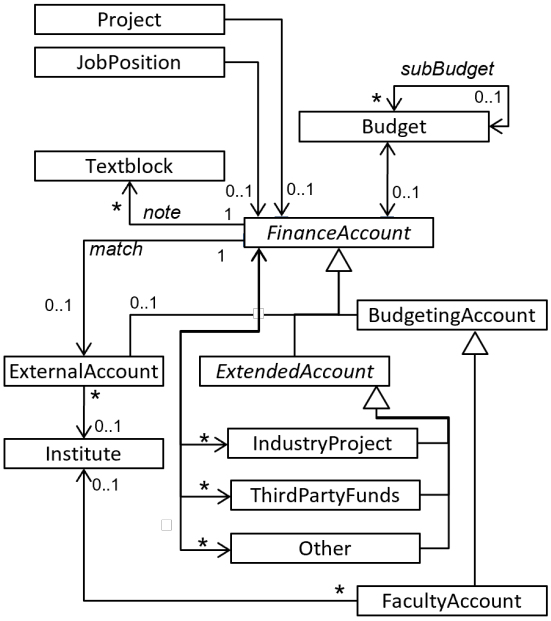
- **Project Management**: Classes describing projects, work packages, what persons are assigned to them, and how much effort they spent on projects.
- **Application Settings**: Classes describing the individual user interface configurations and system-wide settings.

## 5.1 Financial Management

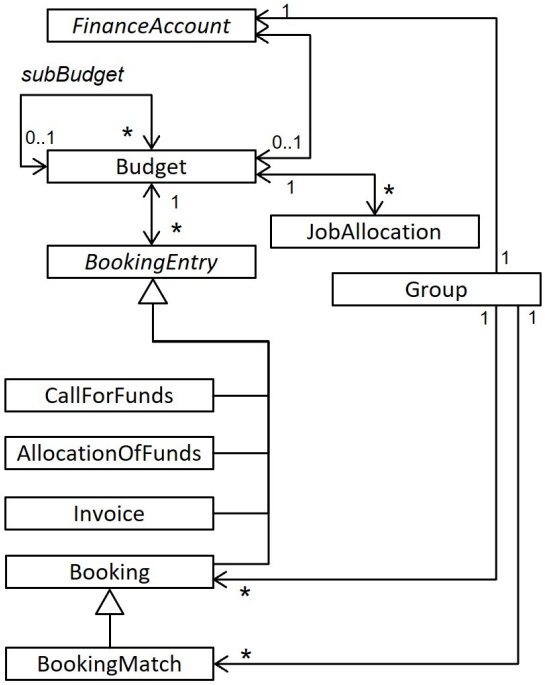
The financial data structure of the MaCoCo project (Fig. 4) is organized around the `FinanceAccount` class (Fig. 4a), where every account can have a `Budget`, which in turn can have multiple sub-budgets. Quite early in the development process, the ability to attach `Notes` to accounts was requested, as chairs need the ability to attach unstructured information in a lightweight manner. At the same time, the account was further specified and subsequently divided into three subclasses that extend `ExtendedAccount`. `FinanceAccount` defines basic account information, whereas `ExtendedAccount` contains further details. The inheriting three classes `IndustyProject`, `ThirdPartyFunds`, and `Other` implement the primary use cases for accounts that appear in the financial management of institutes. Finally, classes for two additional purposes were added: Synchronization with SAP and communication between faculty and institute. Accounts that are imported from SAP are stored in a separate subclass from `FinanceAccount`: `ExternalAccount` to prevent overwriting of existing information in the first step, but allow for comparison in the second step.

The last account type that was added to MaCoCo is `FacultyAccount`. The corresponding use case refers to the requirement that university institutes have to spend the money they receive on a specific account type until the end of a year. To ensure this, the deanery of one faculty observes these accounts and reminds the institutes to do so in a self-defined communication process which we had to implement within the application. The class contains additional information to manage this communication between an institute and the deanery. A `FacultyAccount` is linked to an institute to provide context to faculty users, as one faculty typically communicates with multiple institutes.

Within the MaCoCo project all financial flows are described with booking objects. The data structure around the `Booking` class is shown in Fig. 4b. Similar to the `FinanceAccount` class a booking is defined by a lightweight abstract class `BookingEntry` which is extended by the `Booking` class. Every booking has to be part of either a budget or a sub-budget, which is linked to a class (connecting the Class Diagram in Fig. 4a with Fig. 4b). Multiple extensions of the basic `BookingEntry` class are needed to meet the




*(a) Account-Classes*




*(b) Booking-Classes*

*Figure 4: Class Diagram for Finances*

different requirements of the application. In order to facilitate account synchronization between local accounts and SAP down to a booking level, the `Booking` class was extended to store additional information for the synchronization process. Bookings and accounts that are received from an external source and matched to internal data are tracked within the `Group` class.

The financial data structure is linked to the other sub-domains of the MaCoCo project using several classes: An account can be directly linked to a `Project` Sect. 5.3, enabling referencing from the project to corresponding finances. In order to keep object loading efficient, the association is only unidirectional from a project to an account, as in most use cases the account is relevant in the context of a project, but not otherwise.

The financial data structure is also linked to the staff management Sect. 5.2 via the account and budgets. An account can be defined as a source of financing for a `JobPosition`. Similarly, the budgeting of a `JobAllocation` is linked to a budget, providing a target for the assignment of staff bookings.

## 5.2 Staff Management

The infrastructure for staff management (Fig. 5) is centralized around the `Person` class describing a staff member. The essential information about staff members includes their personal information, such as name and birthday, as well as employment type and further details such as specific employment forms, contracts, and their relation to the financial counterpart.

The `EmploymentType` class is an intermediate structure that indicates the current employment status, showing that an employee is, e. g., a scientific staff member or a research assistant. The `EmploymentForm` contains more detailed information about concrete employment including the time period when a staff member has a specific employment status, e. g., a research assistant from 2021-01-01 to 2021-31-12, their pay level during that time period, and their salary.

An actual physical contract is reflected in the system via the `Contract` class. A contract is

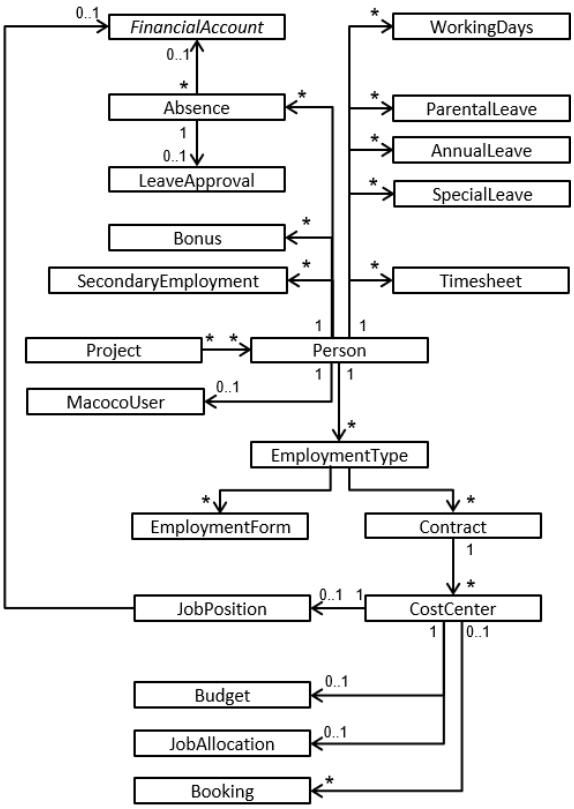



*Figure 5: Class Diagram for Staff*

closely related to an employment form. It is active during a certain period of time, where the time interval needs to satisfy constraints set by an employment form. A contract serves as a connection point to the financial sub-domain since it is used to define the source of the employee's salary. There are several types of such connections, that are defined via the `CostCenter` class. The finances can be allocated in different forms, for example: by automatic creation of bookings, by specifying a budget for withdrawals, or by referencing the employee's post via the `JobPosition` class.

The described structure covers the core concepts which were used in staff management for a long time. With the growth of the user base of the MaCoCo project, new requirements have shaped new areas of resource management and new connections towards existing areas. The staff management has become more complex, as a `Person` class can now become associated with the `MacocoUser` class, indicating a connection between the actual user of the system and a staff member whose data is managed in the system. This change has a notable presence in the data access management infrastructure, as personal data has become accessible to the corresponding user regardless of the access rights granted to that user.

The need for additional functionalities as a result of growing user demands led to further expansion of the staff management infrastructure. Classes like `ParentalLeave`, `SpecialLeave`, and `Absence` were added to describe different types of leaves such as vacations and sick leaves taken by a person. The total annual amount of vacation days allocated for a person is handled in the `AnnualLeave` class. This information provides the data basis to handle, e. g., vacation application processes, but can also be included in timesheets for third-party funded projects.

New information is continuously added to the system. The most recent changes include the addition of `SecondaryEmployment` and `Bonus` classes handling supplementary income sources and salary bonuses.

## 5.3 Project Management

In the context of a university chair, a project describes an individual or collaborative undertaking that is carefully planned to achieve a particular (research) aim. Therefore, the `Project` class can be linked to multiple `Persons` that work on several `Workpackages`. Depending on the funding of the project, a `Person` may be required to keep a timesheet, that contains several `TimeSheetEntries`, defining each worked time interval. Each time interval is linked to a `Workpackage`, providing a transparent overview of who worked when on what part of which project. The note feature used on the accounts in Fig. 4a was also requested for projects, resulting in the association from `Project` to `TextBlock`. Finances are always an important aspect of any project, therefore an account can be linked to any project, providing access to financial information in budgets and bookings related to this project.

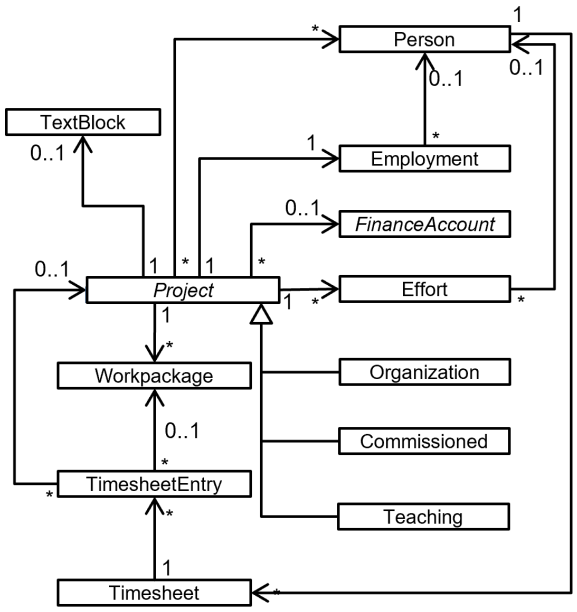



*Figure 6: Class Diagram for Project management*

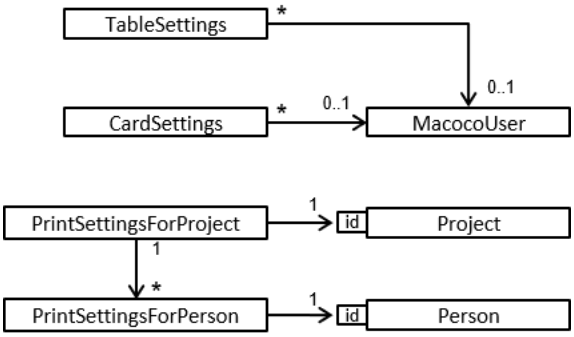



*Figure 7: Class Diagram for Settings*

## 5.4 Application Settings

With the growing complexity of the graphical interface, persisting user configuration has become a necessity. Each user has their own preferences for the information they would like to view, this has triggered a few changes in the project. The tables have become more configurable, e. g., features like sorting, filtering, and grouping of table entries were introduced. Specific view parts, called cards, were adapted to hide information on button clicks. Such configurations were required to be persistent between login sessions, which resulted in the creation of the classes `CardSettings` and `TableSettings` within the domain model Fig. 7.

The settings are saved on a per user basis, hence the association to a `MacocoUser` class. The settings also contain information about the location of a specific table or card, such as the page where the configuration is set. The structure of persisted settings information was initially viewed to be simple as it does not require any processing alongside the business logic of the project. This is why it was decided to save each setting as a simple JSON string. While this approach works well for very basic configurations, such as cards being expanded or not, more complex table settings have become less robust to change with the growth of their structure. When we change a table, e. g., the addition, removal, or modification of a table column, the saved data becomes inconsistent with the new settings structure and requires additional effort in the migration of JSON strings, when compared to more fine-grained object-to-object relations. Moreover, new functionality continues to add complexity to our user interface configuration, for example, the printing of project and personal information has been implemented and as a result, we had to expand the setting structures to include structures `PrintSettingsForProject` and `PrintSettingsForPerson`.

## 5.5 Other Domain-Specific Models

Class diagrams have proven to be a great tool for describing a data structure and are heavily utilized in the project. However, other aspects such as complex application logic or user interface are not covered as well using the class diagrams and are tackled using different tools.

**GUI-DSL.** Graphical user interfaces are one of the most important aspects of our application and require efficient handling from developers. In the MaCoCo project, GUI models are utilized to describe a large part of web pages covering all application domains presented in this work and more Fig. 8. The connection to the domains is apparent: central entities of each domain, such as account and project, spawn a multitude of views

that divide the management of the domain into smaller activities. For example, staff management is broken down into several sections of user interfaces, each tackling different aspects, such as browsing and updating employee information, handling financial aspects, and reporting workload. However, specific classes and interfaces of the data structure cannot be mapped directly onto the user interface, since each view combines different interrelated information.

Functionalities that are loosely dependent on a domain, such as create, read, update, and delete (CRUD) operations, have a generated default implementation. They are, however, sometimes extended using handwritten code to handle specifics of a domain, for example, extending a contract leads to the creation of additional bookings. In other cases, functionality is moved to a library, e. g., the filtering widget which handles both common queries and domain-specific parts had been refactored to a domain-independent library component. The domain-specific parts of the widget remain in the model.

**OCL/P.** The MaCoCo project is heavily oriented towards data management and consequently has to provide a validation mechanism for the data. Whenever a user adds or updates information in the system, it needs to be checked against legal and organizational standards, which require the corresponding implementation in the code. For this purpose, OCL/P models are utilized. They supplement the data structure with additional information. The context of an OCL/P invariant is always a domain class. The example in Fig. 9 shows how a simple OCL/P constraint is defined for a class `Person`, and how it affects the user interface when an incorrect value is entered in the field corresponding to the `age` attribute. Such OCL/P statements provide a simple way of controlling user input in MaCoCo. It ensures that a restriction is obeyed in different parts of the project since the validation procedure is automatically integrated by the generator both in the user interface and the application logic. Since OCL/P is used for input validation, the statements are paired with corresponding error messages.

## 6 Results

The generated MaCoCo source code (up to March 2022) is based on 73 domain classes that are defined in the class diagram. Further classes are defined in 62 view models. The application has 74 pages that are based on the same amount of GUI models and less than 10 further handwritten pages.

The generator produces 500.573 Line of Code (LOC) that are extended with a further 170.505 handwritten LOC (Table 1). MaCoCo uses the MontiCore TOP Mechanism (Grönniger et al. 2006; Hölldobler and Rumpe 2017), which utilizes inheritance to override or extend the generated code. This keeps generated and handwritten codes separated while integrating them into the product. The TOP Mechanism allows for an iterative development process, as changes or additions are still in place during continuous regeneration. The greatest amount of code is written and generated in Java, however, for the HTML code, a greater ratio is generated. This can be explained by the fact that the HTML code used is rather static and difficult to extend. MaCoCo contains 7435 LOC of handwritten SCSS code in addition to inline CSS included in HTML LOC.

| | gen | hw | $\sum$ | ratio |
|---|---|---|---|---|
| Java | 325.709 | 102.442 | 428.151 | 76.07% |
| TS | 140.525 | 62.335 | 202.860 | 69.27% |
| HTML | 34.339 | 5.728 | 40.067 | 85.70% |
| Models | 0 | 14.251 | 14.251 | 0% |
| Total | 500.573 | 184.756 | 685.329 | 74.59% |

*Table 1: LOC of the MaCoCo platform ('gen' = generated, 'hw' = handwritten)*

We can define the user interfaces via models very precisely and also extend the generator to meet our requirements. However, there is still 30.73% handwritten code to adjust the generated TypeScript code. The generator cannot provide generic solutions for every specific use case that require a high degree of optimization to satisfy the high standards set by the users. Thus, every page consists of generated base code that is extended and optimized with handwritten code.

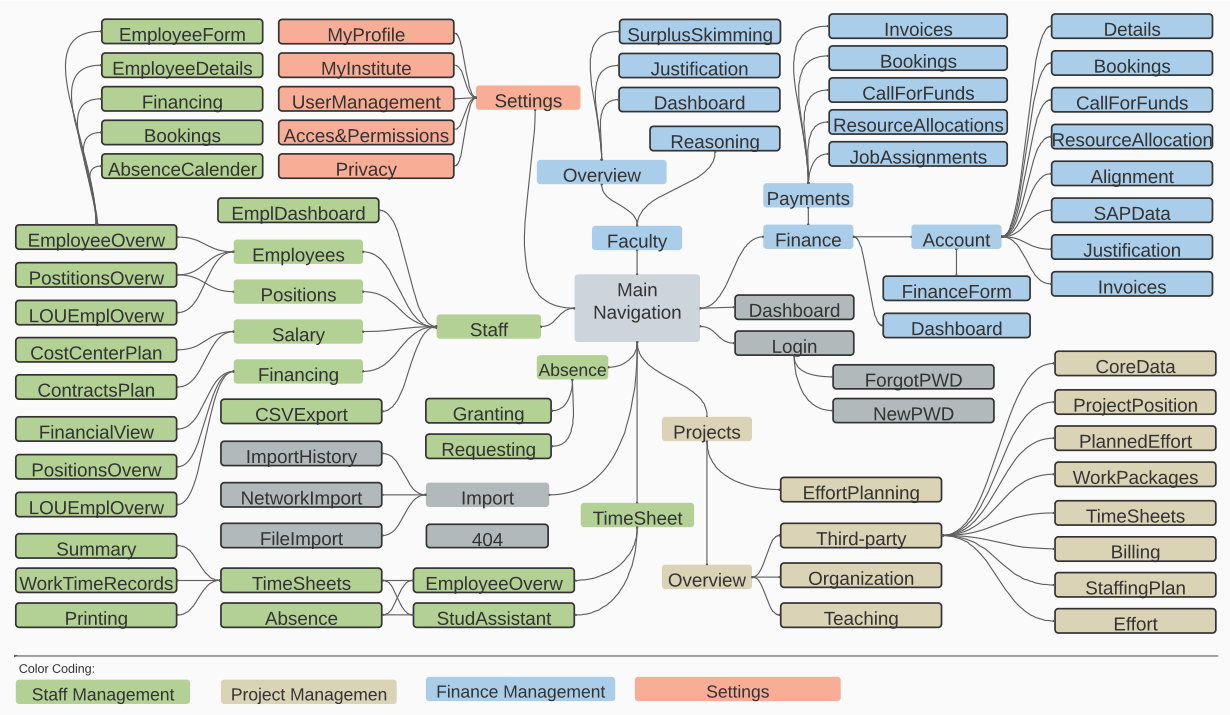



*Figure 8: Currently Generated Pages for the Different Segments in MaCoCo.*

Fig. 10 shows a screenshot of the current state of MaCoCo (Version 2.7.3_20220310). The page is used to manage staff planning. In this view, an employee (A) with the name abbreviation 'BEABI' is mapped to a cost center (B) (in this case a global budget). The dates from the scope of employment are displayed in (C). Next to it, we can inspect the planned person-month (D) for the upcoming years. In order to improve the workflow, filtering options (E) were added that allow the user to filter results by employee name, cost center, and type of employment. Additionally, the viewed time horizon can be adjusted (F). Finally, the table provides the same generic settings (G), as any other table in MaCoCo, allowing the user to group, sort, hide and rearrange columns and also export the contents.

OCL/P

```
1 ocl ExampleConstraints {
2   context Person inv isAgeNotNegative:
3     age >= 0;
4 }
```

Bob

Age

-10

Age has to be positive

*Figure 9: Example OCL/P Model for age validation*

At the time of writing, MaCoCo was provided to 188 chairs and institutes of RWTH Aachen university, making up a third of all available professorships and serving 1403 users. On an average workday, 157 users login to the web application. In its current state, MaCoCo is primarily designed to support the office management in their accounting activities, however, the testing process for MaCoCo timesheet capabilities is still ongoing. Once the feature is rolled out, the number of daily users is expected to increase dramatically, as not only administrative staff would use the platform, but also student assistants and doctoral candidates.

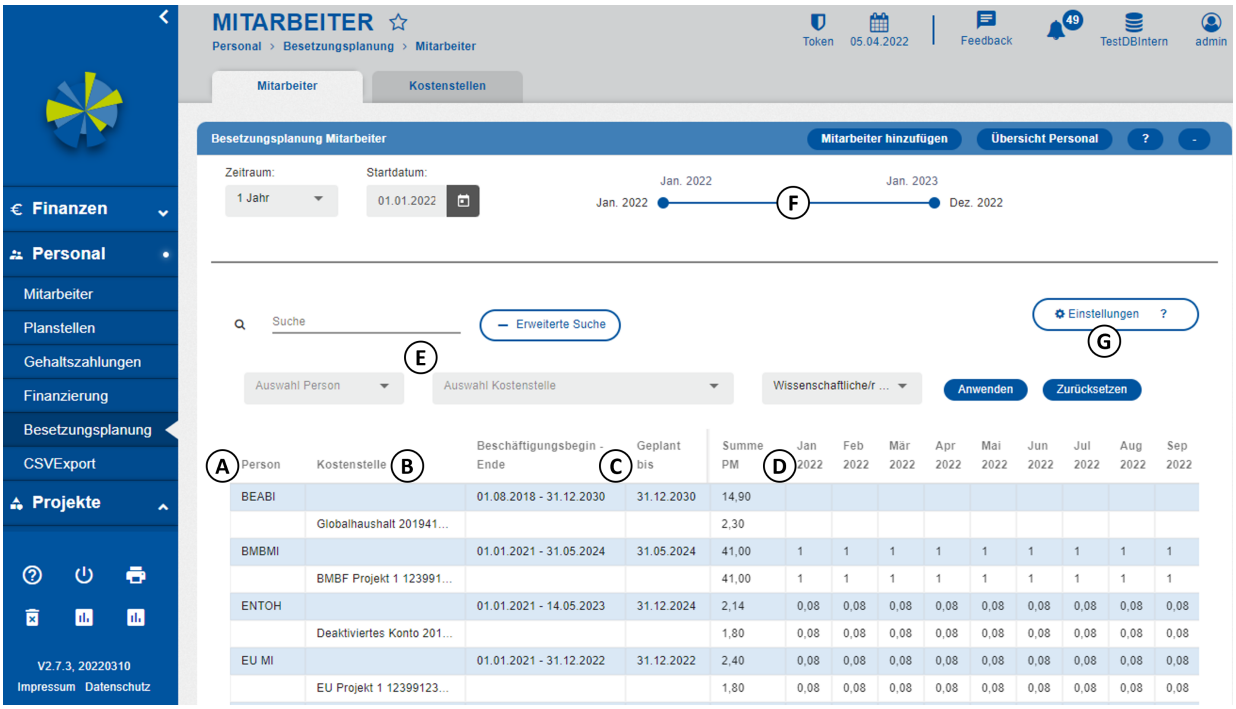



*Figure 10: Screenshot of the page for the staff planning (Fig. 8, bottom right)*

## 7 Discussion and Lessons Learned

Using a model-driven approach within a project brings advantages as well as disadvantages to the development process (Akdur et al. 2018). In the following, we discuss the most important lessons learned from developing full-size real-world applications and the return of modeling effort for the developers and end users. For lessons learned about the parallel development of an application and the used generator see (Adam et al. 2020).

**Retrofitting generative aspects leads to constant generator refinement and occasional new DSL engineering.** The MaCoCo project started as a greenfield approach with a completely handwritten prototype. Within the following months, we iteratively replaced parts of the application with generated code (Drave et al. 2021). When incorporating generative methods, the developers are free to choose what parts to replace with generated code. With a growing code base, we recognized new patterns that led to a systematical replacement of handwritten code via generated parts within multiple iterations. This also includes the creation of new DSLs and corresponding generators. For example, generic structures emerged in the user interface implementation and validation logic, which could not be easily described using class diagrams and were therefore tackled with models of new languages instead. See (Drave et al. 2021) for more details on the retrofitting process.

**Using a model-driven approach reduces the problem implementation gap and enables quick adaptation of generated parts for the application.** A major challenge in software development is understanding requirements and mapping them to the implementation (France and Rumpe 2007). Although the developer is an expert in computer science, he might not have any expertise in the domain the application is used for, and still has to define how the system is to be used. Defining models alleviates this problem by (a) providing a single point of truth and a basis for communication between domain experts and developers. While models are a basis for communication between domain experts and developers, this does not mean that they must be easily understood

by domain users on their own. Our experiences have shown that the textual CD4A models are normally understandable by experts in the business management domain after a short time. However, **software developers are needed for further explanations and especially to define more complex additions to textual models.** The second aspect where defining models alleviates the problem-implementation gap is by (b) enabling quick adaptations for changes that occur in the requirements if they are covered within the models, e. g., adding attributes to a class, or defining associations between sub-domains. **If changes are related to the handwritten code**, e. g., a new calculation in the business logic, **these changes require as much effort as in non-generative project implementations**.

**The learning curve for a MDSE project is steep but can be mitigated via practical training.** A system that is implemented using a model-driven approach utilizes one or several modeling languages, introducing additional effort for new employees trying to come into the project. In the university context, frequent rotation in a development team is not uncommon, as students and Ph.D. candidates are only working for a limited time until they graduate. MaCoCo being a university project and having its implementation based on multiple modeling languages has certain drawbacks and benefits. Student assistants that have to get started with the platform need to learn an unfamiliar modeling language and the concept of model-driven development. However, a practical environment helps new employees to adapt quickly (Ciccozzi et al. 2018). Once they get familiar with the system, they have little problems extending the implementation with their own code, as they can simply extend the models. As most of the code is generated, the few extension points for handwritten code are easily located. Moreover, we engage new developers to review other handwritten implementations to improve their own code design.

**A generative approach enables fast integrations of system-wide adaptations.** The source code in MaCoCo can be divided into two groups: Domain dependent and generic code. Generic code, such as utility classes is easily adapted and affects all generated code that relies on it. As the domain-dependent code is generated, new solutions and optimizations can be integrated throughout the entire code base with little or moderate effort. In several stages of the project, generic changes were added to the generated code, e. g., changes to the caching, security, and access control on domain objects. These changes were integrated into the generator and are therefore generated not only in the existing target code but also for any future feature implementations.

**Generating security and access control ensures consistency in their implementation.** Access control should be applied with a holistic approach to the system in order to minimize gaps and inconsistencies that can lead to unwanted access to restricted data (Clavel et al. 2008). We use a generator to apply permission checks and to control access to every object in the database. The ability to see both the data structure and the rules for data access that a model provides enables a clear specification with a single point of reference on how security and privacy concepts are realized in the application.

**A model-driven approach provides testing support.** Having a rapidly growing system, it is important to test all functionalities thoroughly. The effort spent towards testing increases as the business logic and relations between different parts of the application becomes more complex. Considering the fact that major parts of the system are described using models, the tests can also be derived using the same models (Mussa et al. 2009). A good example in the MaCoCo case is the usage of OCL/P models for validation of user input. The validation logic is mostly generated from the OCL/P constraints and in order to test it, one could potentially **derive test data from the constraints**, as they effectively define different cases, in which information can be considered valid or invalid (Sartaj et al. 2019). For example, if a person's age has to be non-negative, one would use a negative, positive, and zero value as test cases, which can be inferred from a corresponding OCL/P constraint.

Since the models define a basis for most of the functionalities of the system, additional utilities can be generated for testing purposes, such as stubs and mocks. It is fairly simple to produce a husk of an actual system component if the component is to be generated anyway. For example, GUI models can be used to create mock-up web pages to simplify testing the interaction between a client and a server. By modeling the scenarios further, test cases can be generated completely (Bünder and Kuchen 2019).

Another interesting aspect is the automatic creation of tests, that fail unless implemented by a developer. When used in a context of continuous integration, such **generated tests support ensuring that no functionality inadvertently or purposefully remains untested** for a long time. Such an approach requires careful planning to avoid unnecessary alarms, which can occur if a test is generated for a trivial function not requiring any tests.

**A model-driven approach provides migration support.** During the development of an information system, changes in the data structure are unavoidable and when combined with an agile approach, which implies release cycles, the process of data migration becomes necessary (Brodie and Stonebraker 1995; Hick and Hainaut 2003). However, if the data structure is completely modeled, the changes in the models can be observed to assist with the migration process. Furthermore, when changes in the model can be derived, the generation of a migration script becomes possible. **Our practical experience has shown that model-driven approaches can support this, but should not be used to fully automate data migration.** In our opinion, it is too risky to use migration scripts on data of a running system without additional manual checks. The task is difficult to automate completely, as some changes depend on the semantic context and should be handled differently. When adding new information one might want to choose a specific default value. Taking the MaCoCo contract class as an example, if it receives a new flag indicating its status as officially confirmed or only planned, then an opposing default value can be implemented, such as: `true` if the flag state is `confirmed` and `false` if its state is `planned`. Either way, tracking the changes in the data structure is easier when a model-driven approach is involved and new possibilities for simplifying the migration process emerge.

**Using a code generator allows development effort to be shifted,** e. g., from GUI details to more complex business logic. The longer software is maintained, the greater its source code will get with each additional use case that needs to be implemented. In the case of the data-centric MaCoCo application, further extensions to the graphical user interface are developed to satisfy user demands for new features. With the help of the generator, these requests are processed systematically by adding new parts to the GUI and domain models. This saves us time, which we use for the implementation of more complex business logic or optimizations in the persistence layer. Another benefit is that the developer does not need to know the exact structure of added code, as it is provided by the generator.

**The refactoring process changes when using code generation.** Small changes to the generator can be used to systematically update the application, as the implementation is largely generated and, thus, a result of the generator's configuration. Changing the algorithm on how specific data is cached in an application as large as MaCoCo would require changing hundreds of classes by hand. As each class is different, this would be a time-consuming refactoring task. By adapting the template used to create the classes, a change has to be performed only at a single point in the generator and will cascade to the entire application. However, if a change involves restructuring interfaces, the refactoring has to be done manually and also includes the additional effort of adjusting the generator's implementation.

**Integrating optimizations in the generated code is difficult.** Models are an abstraction of entities that they represent. Thus the generator developer has to make assumptions about what implementations are needed when configuring

the generator to synthesize the application code. The larger the targeted implementation gets, the harder it becomes to configure the generator to provide efficient code for all use cases. In our project the generator was configured to handle small to medium size institutes, but not large ones. Adapting the application to handle new requirements triggered either an adaptation of the generator or an extensive adaptation of the handwritten code and run-time environment. If the general requirements of the application change, e. g., a dramatic increase in the size of the user count or database, then the application is not only limited by its own architecture, but also by the architecture of the generator, that was configured for the initial requirements.

**Models provide a good system overview and help to avoid redundancy in the implementation.** The data structure in a form of models provides a very good overview of a system (Torchiano et al. 2013). Despite the considerable growth of the MaCoCo project over the course of several years, any considerable changes can still be easily tracked during the re-designing process. For example, when the project management domain was introduced and integrated with a financial block, some new structures were added. However, the requirements for both domains were overlapping and could potentially become a source of redundant specification in the system, such as attributes holding duplicated information in different objects. Such problems were successfully avoided since models were used during the development providing a comprehensible summary of changes and being the primary artifact for the implementation of features.

**Small issues in the data structure may lead to significant problems in a generated system.** Using the data structure model simplifies certain development activities, such as data migration. This holds true so long as the data structure is fine-grained, i. e. it is broken down into simpler, more manageable parts. In MaCoCo, the user-specific table settings consisting of several properties, such as the number of entries displayed and column widths, were observed as a whole, which led to spending additional effort on data migration when the default setup of any table was changed between product releases. Even when a piece of data has a simple structure, tracking down and handling changes within the data introduces considerable overhead if one cannot view and manage its fragments separately.

## 8 Conclusion

We have shown an example of a model-driven approach that is used to continuously develop and maintain a full-size real-world application. We apply it at RWTH Aachen University to create an enterprise information system for the financial, project, and staff management of chairs and institutes. The benefits of using such an approach greatly exceed the drawbacks of having this layer of abstraction. We intend to further incorporate generative methods to systematically replace handwritten code with generated parts, especially for testing the application.

## References

Akdur D., Garousi V., Demirors O. (2018) A survey on modeling and model-driven engineering practices in the embedded software industry. In: Journal of Systems Architecture 91

Ambrosy R., Heise S., Kirchhoff-Kestel S., Müller-Böling D. (1997) Integrierte Kostenrechnung: Unterwegs zu einem modernen Hochschulmanagement! In: Wissenschaftsmanagement 4, pp. 204–213

Adam K., Michael J., Netz L., Rumpe B., Varga S. (2020) Enterprise Information Systems in Academia and Practice: Lessons learned from a MBSE Project. In: 40 Years EMISA: Digital Ecosystems of the Future: Methodology, Techniques and Applications (EMISA'19). LNI Vol. P-304. Gesellschaft für Informatik e.V., pp. 59–66

Adam K., Netz L., Varga S., Michael J., Rumpe B., Heuser P., Letmathe P. (2018) Model-Based Generation of Enterprise Information Systems. In: Enterprise Modeling and Information Systems Architectures (EMISA'18). CEUR Workshop Proceedings Vol. 2097. CEUR-WS.org, pp. 75–79

Brdesee H. (2021) A Divergent View of the Impact of Digital Transformation on Academic Organizational and Spending Efficiency: A Review and Analytical Study on a University E-Service. In: Sustainability 13(13)

Brodie M. L., Stonebraker M. (1995) Migrating legacy systems: gateways, interfaces & the incremental approach. Morgan Kaufmann Pub

Bünder H., Kuchen H. (2019) A Model-Driven Approach for Behavior-Driven GUI Testing. In: 34th ACM/SIGAPP Symposium on Applied Computing. ACM, pp. 1742–1751

Ciccozzi F., Famelis M., Kappel G., Lambers L., Mosser S., Paige R. F., Pierantonio A., Rensink A., Salay R., Taentzer G., Vallecillo A., Wimmer M. (2018) How Do We Teach Modelling and Model-Driven Engineering? A Survey. In: ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings. ACM, pp. 122–129

Clavel M., da Silva V., Braga C., Egea M. (2008) Model-Driven Security in Practice: An Industrial Experience. In: Model Driven Architecture – Foundations and Applications. Springer, pp. 326–337

Drave I., Gerasimov A., Michael J., Netz L., Rumpe B., Varga S. (2021) A Methodology for Retrofitting Generative Aspects in Existing Applications. In: Journal of Object Technology 20, pp. 1–24

Domain-specific conceptual modeling: Concepts, methods and tools. Springer

France R., Rumpe B. (2007) Model-driven Development of Complex Software: A Research Roadmap. In: Future of Software Engineering (FOSE '07), pp. 37–54

Frank U. (2013) Domain-Specific Modeling Languages: Requirements Analysis and Design Guidelines. In: Domain Engineering. Springer, pp. 133–157

Freemarker (2022) Freemarker https://freemarker.apache.org/ Last Access: 2022-12-16

Gerasimov A., Heuser P., Letmathe P., Michael J., Netz L., Rumpe B., Varga S., Volkova G. (2022) Domain Modelling of Financial, Project and Staff Management

Gerasimov A., Heuser P., Ketteniß H., Letmathe P., Michael J., Netz L., Rumpe B., Varga S. (2020a) Generated Enterprise Information Systems: MDSE for Maintainable Co-Development of Frontend and Backend. In: Companion Proc. of Modellierung 2020 Short, Workshop and Tools & Demo Papers. CEUR Workshop Proceedings, pp. 22–30

Grönniger H., Krahn H., Rumpe B., Schindler M. (2006) Integration von Modellen in einen codebasierten Softwareentwicklungsprozess, German. In: Modellierung 2006 Conference, pp. 67–81

Gerasimov A., Michael J., Netz L., Rumpe B., Varga S. (2020b) Continuous Transition from Model-Driven Prototype to Full-Size Real-World Enterprise Information Systems. In: 25th Americas Conference on Information Systems (AMCIS 2020). AIS, pp. 1–10

Gerasimov A., Michael J., Netz L., Rumpe B. (2021) Agile Generator-Based GUI Modeling for Information Systems. In: Modelling to Program (M2P). Springer, pp. 113–126

Hesse W., Mayr H. C. (2008) Modellierung in der Softwaretechnik: eine Bestandsaufnahme. In: Informatik-Spektrum 31(5), pp. 377–393

Hick J.-M., Hainaut J.-L. (2003) Strategy for database application evolution: The DB-MAIN approach. In: International Conference on Conceptual Modeling. Springer, pp. 291–306

Hölldobler K., Kautz O., Rumpe B. (2021) MontiCore Language Workbench and Library Handbook: Edition 2021. Aachener Informatik-Berichte, Software Engineering, Band 48. Shaker Verlag

Hölldobler K., Michael J., Ringert J. O., Rumpe B., Wortmann A. (2019) Innovations in Model-based Software and Systems Engineering. In: The Journal of Object Technology 18(1), pp. 1–60

Hornbostel S. (2001) Third party funding of German universities. An indicator of research activity? In: Scientometrics 50(3), pp. 523–537

Hölldobler K., Rumpe B. (2017) MontiCore 5 Language Workbench Edition 2017. Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag

Karsai G., Krahn H., Pinkernell C., Rumpe B., Schindler M., Völkel S. (2009) Design Guidelines for Domain Specific Languages. In: Domain-Specific Modeling Workshop (DSM'09). Techreport B-108. Helsinki School of Economics, pp. 7–13

Krahn H., Rumpe B., Völkel S. (2010) MontiCore: a Framework for Compositional Development of Domain Specific Languages. In: International Journal on Software Tools for Technology Transfer (STTT) 12(5), pp. 353–372

Küpper H.-U. (2007) Neue Entwicklungen im Hochschulcontrolling. In: Controlling & Management 51(3), pp. 82–90

Küpper H.-U. (2009) Effizienzreform der deutschen Hochschulen nach 1990–Hintergründe, Ziele, Komponenten. In: Beiträge zur Hochschulforschung 31(4), pp. 50–75

Michael J., Mayr H. C. (2015) Creating a Domain Specific Modelling Method for Ambient Assistance. In: Int. Conf. on Advances in ICT for Emerging Regions (ICTer). IEEE, pp. 119–124

Müller-Bromley N. (2011) Hochschulen richtig reformieren. In: Die neue Hochschule 1, pp. 1–2

Mussa M., Ouchani S., Sammane W. A., Hamou-Lhadj A. (2009) A Survey of Model-Driven Testing Techniques. In: 2009 9th Int. Conf. on Quality Software, pp. 167–172

Object Management Group (2014) Object Constraint Language

Object Management Group (2017) OMG Unified Modeling Language (OMG UML)

Rumpe B. (2016) Modeling with UML: Language, Concepts, Methods. Springer International

Sartaj H., Iqbal M. Z., Jilani A. A. A., Khan M. U. (2019) A Search-Based Approach to Generate MC/DC Test Data for OCL Constraints. In: Search-Based Software Engineering. Springer, pp. 105–120

Schwaber K. (1997) SCRUM Development Process. In: Business Object Design and Implementation. Springer London, pp. 117–134

Stachowiak H. (1973) Allgemeine Modelltheorie. Springer-Verlag

Torchiano M., Tomassetti F., Ricca F., Tiso A., Reggio G. (2013) Relevance, benefits, and problems of software modelling and model driven techniques—A survey in the Italian industry. In: Journal of Systems and Software 86(8), pp. 2110–2126

Völter M., Benz S., Dietrich C., Engelmann B., Helander M., Kats L. C. L., Visser E., Wachsmuth G. (2013) DSL Engineering - Designing, Implementing and Using Domain-Specific Languages. dslbook.org

About SAP.. https://www.sap.com/about.html. Last Access: [Online; accessed 16-December-2022]