# Extension of low-code development platform with intelligent components

Fedor Chikhachev
University of Luxembourg
Email: fedor.chikhachev.001@student.uni.lu

***Abstract*—This Bachelor Semester Project focuses on extending the BESSER low-code development platform by integrating intelligent components to optimise web application creation. The central research question is whether such integration can streamline and enhance the web application development process. The project involves enhancing the BESSER platform's capabilities through the incorporation of new code-generation features for Streamlit and Django applications. In addition to conventional rule-based code generation, this project explores the potential of Large Language Models (LLMs) for implementing code generators, offering a comparative analysis of both approaches. By combining low-code development principles with intelligent software support, this project aims to demonstrate the feasibility of the concept and provide a practical tool for more efficient web application development.**

## I. Introduction

In today's digital age, the software industry demands speed without compromising on quality. Low-code platforms have emerged as one of the solutions, aiming to simplify and expedite application development. Yet, the realm of integrating artificial intelligence within these platforms remains largely uncharted. The main point of this deliverable lies in answering the following question: "Whether the integration of intelligent components within a low-code development platform can streamline and enhance the process of creating web applications?".

This question aims to uncover:

- The actual advantages, if they exist, of combining intelligence and low-code methods.
- The possible difficulties and drawbacks of this combination.
- The wider effects of this fusion on the field of software engineering.

## II. Context and Motivation

### A. Low-code Development Platform

A Low-code Development Platform is a software development environment that allows users to create applications with minimal hand-coding and manual programming effort. Unlike traditional code-first development methods which require extensive coding knowledge, low-code platforms reduce the amount of hand-coding to a minimum. They utilise visual tools, drag-and-drop functionality, and automation to facilitate the creation of applications. This approach is generally faster and easier compared to traditional development methods.

Low-code development platforms are particularly useful for organisations that want to accelerate their application development process, reduce development costs, and empower citizen developers (non-technical users) to contribute to application creation. They are often used to build a wide range of applications, from simple mobile apps and web forms to complex enterprise software solutions [1].

### B. LLMs (Large Language Models)

LLMs, or Large Language Models, refer to a class of artificial intelligence models designed for natural language understanding and generation tasks. These models are characterised by their vast size, with billions or even trillions of parameters, which allows them to exhibit remarkable language-related capabilities.

LLMs are typically built upon deep learning architectures, such as transformers, and are trained on massive text and data corpus from the Internet. They gained their popularity recently and massively, which leaded the society to the set of questions, concerning their possibilities in substituting human power in Software Engineering domain. However, the current state of technology had some critical drawbacks and was considered over-hyped, yet there are still some tasks, where AI models can assist developers, but not completely autonomously [10].

### C. Code Generation

Code generation extends to automatic programming, where source code of software is generated based on models or templates. This generation can be done at run time in self-modifying code and just-in-time compilation scenarios. The realm of code generation has expanded to predicting code structures from various data types, like incomplete code or natural language descriptions, facilitating the creation of automatic programming tools to enhance programming productivity. Significant studies of this domain explores generating code for automatic software development using machine learning and predictive models [5].

### D. Motivation

The advent of Artificial Intelligence (AI) has brought about the development of AI code generation software tools. These

**This report has been produced under the supervision of:**
**Jordi Cabot Sagrera**
**Luxembourg Institute of Science and Technology (LIST)**
**Email: jordi.cabot@list.lu**

tools are designed to save time by enabling faster code generation, which in turn reduces the manual work of writing lines of code. Generative AI can also efficiently test and debug computer code, showcasing the evolving landscape of code generation with AI's assistance [6], [13].

Low-code platforms are indeed trending as of now, and the nature of this can be attributed to their ability to expedite software development processes, lower the entry barrier for non-professional developers, and contribute to cost efficiency in project management, among other benefits.

One of the architectures proposed for such platforms consists of several blocks: a model editor, code generator, training, deployment, traditional software components, and monitoring and feedback systems, aiming to lower the entry barrier for future smart software developers [3].

It is also hypothesised that the future of low-code platforms will be heavily intertwined with LLMs. Instead of manually crafting every line of code, developers would design models, which, acting as high-level specifications, would then be converted into precise prompts, instructing the LLMs to generate the required code, thus streamlining the whole development process [11], [12].

The approach in this project is to evaluate the effectiveness and correctness of LLM acting as a code generator within a low-code platform, where ”low-code” is defined as ”human text prompt”. By creating three parallel development processes – the first with the full integration of intelligent components, the second with slight assistant and the third completely unaided – it is possible to provide an assessment and contrast the efficiencies and outcomes of all of them. This will provide clear insights into the advantages or disadvantages of this integration.

As a theory on how to improve the quality of code generation by AI models, it is supposed to combine these three approaches while experimenting with LLM models:

- Fully handwritten templates for code generation, without any involvement of LLMs. This is an algorithmic approach to code generation problem.
- The ”Black Box”: with the prompts and other user-specific input provided, an expected output is the final code which can be run ”out of the box”.
- Mix of AI and template code generation: what if the LLM model is asked to create not the code, but the template generator for it, which can then produce the completed code? The hypothesis of this method is a reduction in the number of errors made by AI, as well as an increased proximity of the generated code to the provided technical specifications [2].

## III. Project Requirements and Competencies

To succeed in answering the scientific question of this project, it is not required to have any special knowledge concerning the topic.

To successfully develop the software that aligns with the specified requirements, a solid foundation in Python programming is essential. Proficiency in various aspects of Python, including data structures, third-party package utilisation and management, object-oriented programming (OOP) principles, and advanced OOP concepts like inheritance, interfaces, handlers, setters, and getters, is crucial. Additionally, it is advisable to have an understanding of data models and their potential representations through tools like the Entity-Relationship Model (ERM) or Unified Modeling Language (UML). These skills and knowledge areas will form the bedrock for creating a robust and efficient software solution that meets the specified criteria.

## IV. Methodology

This exploration is part of the FNR PEARL BESSER project [9]: a low-modeling low-code open-source platform, which provides Python-based personal interpretation of a ”Universal Modeling Language”. The technical deliverable of this project revolves around extending this platform aimed at automating the generation of Streamlit and Django applications (or their essential parts with the assumption of future cooperation with the developer). By leveraging both the Jinja template engine and the potential capabilities of Large Language Models (LLMs), this initiative seeks to redefine the boundaries of low-code platforms, pushing them beyond traditional code-generation paradigms.

### A. Functionality

The software has a target: build a low-code component that given a dataset description is able to generate the corresponding database and a set of predefined visualisations to help user interpret the data. In order to achieve this objective, the resultant software should possess a variety of functions:

- Create a database interpretation of BUML-described model, that is, convert user-specified data model into relational database representation.
- Generate a Django model representation of BUML-described model, using the same approach as with classic SQL queries.
- Load a database with the provided description in BUML, get insights about the data loaded.
- Provide visualisation insights with the usage of Streamlit web application building platform.

Each of this steps should be then replicated and/or with the usage of LLM-based code generator, which can possibly save the developer's time and provide more accurate insights about possible strategies for data visualisation. Overall, the goal is to implement a prototype of Model Driven Development process, which is leaded by code generation tools and supported by intelligent component.

### B. Design

In the software development process, it was important not just to write code but also to delve into database modeling principles. This involved understanding of mappings between modeling abstracts in different modeling frameworks and object relational mappings (ORM).

Here are presented the steps towards producing the contribution to BESSER project.

*1) Template-based generation with Jinja:* Jinja template engine will be utilised to automate the code generation for different user input.

- **BUML to SQL:** Develop a generic Jinja template that can interpret and convert any user-specific BESSER-UML model into its corresponding SQL structure.
- **BUML to Django models:** Develop a generic Jinja template that can interpret and convert any user-specific BUML model into Django model structure, which is a way Django Framework manages database schema.
- **Streamlit Web application:** Design a Streamlit script template that can read any BESSER-UML model and provide some visualisation for the data.

*2) LLM-based code generation:* Large Language Models will be utilised to see if they can mimic or enhance the code generation process. The project uses ChatGPT version 4 as a LLM in this context. It is the most known LLM and GPT model for the moment, its advantages are adaptability, better knowledge base, enhanced understanding and contextual awareness.

The key lies in crafting the perfect prompts to guide the LLM into generating the desired code. The generated code (SQL, Django and Streamlit scripts) will be compared against Jinja version in terms of correctness, efficiency, and adaptability.

*3) Combination of LLM and template-based code generation:* Moreover, the LLM model can be asked to generate not the final code directly, but the templates (for example, in already mentioned Jinja template language) from which the final code can be obtained by loading user-specific input.

This is a good environment for comparing three different approaches when working with code generation: pure algorithms, complete trust to LLMs in generation or support in creating algorithmic generators.

### C. Related concepts

*1) Jinja Template Engine:* Jinja is a widely-used template engine for Python programming. It allows for the dynamic generation of content based on predefined templates. It was firstly (and mainly) used for generating HTML templates, providing some hits and tips for dynamic text output, but it can be successfully used with other languages and grammars too [8].

*2) Streamlit:* Streamlit is a Python library that facilitates the rapid creation of web applications for data science and machine learning project. With its simplified syntax, Streamlit empowers developers to transform Python scripts into interactive web applications without the need for detailed front-end knowledge. It provides some complex and sophisticated features, such as real-time data updates, user interaction widgets, data exploration etc., with little amount of code and integration with popular data science packages, such as Pandas, Matplotlib, Seaborn etc [14].

*3) Django:* Django is a high-level Python web framework promoting rapid development, clean design, and pragmatic coding. Engineered by adept developers, it mitigates many web development hassles, allowing focus on app creation without redundant efforts. Django implements data management with its own interface called "Models", which provides developers an environment to build databases, manage and validate data, using specially designed Python's classes [7].

## V. Introduction to BUML

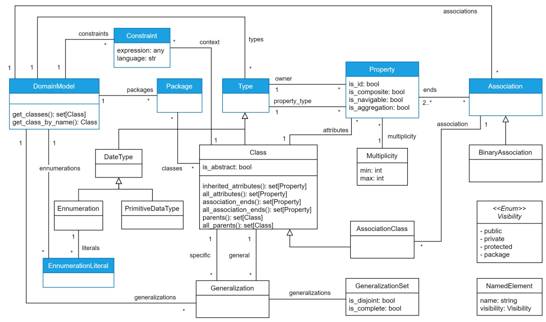



Fig. 1. Metamodel of BESSER-BUML language

BUML (from BESSER-UML) – a domain-specific language, inspired by Unified Modeling Language, created for describing data models. It has concepts similar to UML, such as Classes, Properties (UML's Attributes), Associations (Relationships), Generalisations (Inheritance), etc. This DSL is fully implemented on Python, using OOP concepts and principles; figure 1 refers to the metamodel of this language.

BUML is a part of BESSER project about creating low-modeling low-code open-source platform. In the context of this work, BUML is considered as low-code modeling language, which encourages usage of it for model-driven development.

Two examples of data models are used further in the project. They are created with usage of all crucial BUML's modeling abstracts.

### A. Cinema model

This BUML model represents a domain model for a cinema system, comprising three primary classes: *Movie*, *Studio*, and *Star*. Each class is characterised by specific attributes:

- **Movie Class**:
  - Attributes: `year` (int), `length` (int), `title` (str).
- **Studio Class**:
  - Attributes: `name` (str), `address` (str).
- **Star Class**:
  - Attributes: `name` (str), `age` (int).

The model also establishes relationships between these classes:

1) **Star-Movie Relationship**: A many-to-many relationship, where a `Star` can play in many `Movies` and a `Movie` can feature many `Stars`. Represented by the `starring` property in `Movie` and `starredIn` property in `Star`.
2) **Studio-Movie Relationship**: A one-to-many relationship, where a `Studio` owns multiple `Movies`, but each `Movie` is owned by only one `Studio`. Represented by the `ownedBy` property in `Movie` and `owns` property in `Studio`.

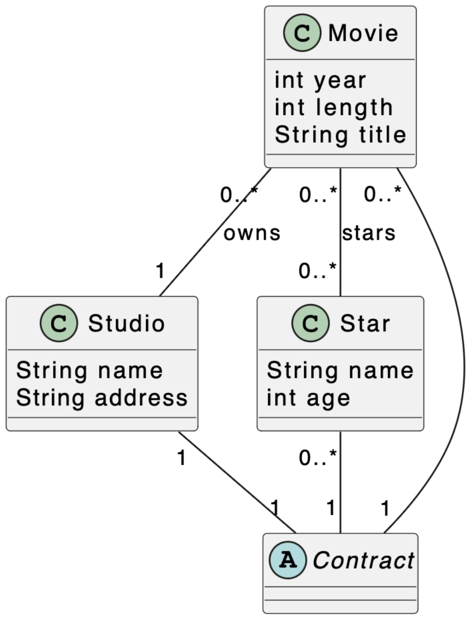



Fig. 2. UML diagram of Cinema data model

3) **Contract Relationship**: An association involving `Star`, `Studio`, and `Movie`, indicating contractual relationships. This is a complex relationship linking all three classes.

A UML diagram of this model is presented at figure 2. Note that due to the limitations of UML language, N-ary relationship `Contract` is presented as a fake class with three connections to the actual classes.

A BUML view is presented in appendix, listing 2.

### *B. Library model*

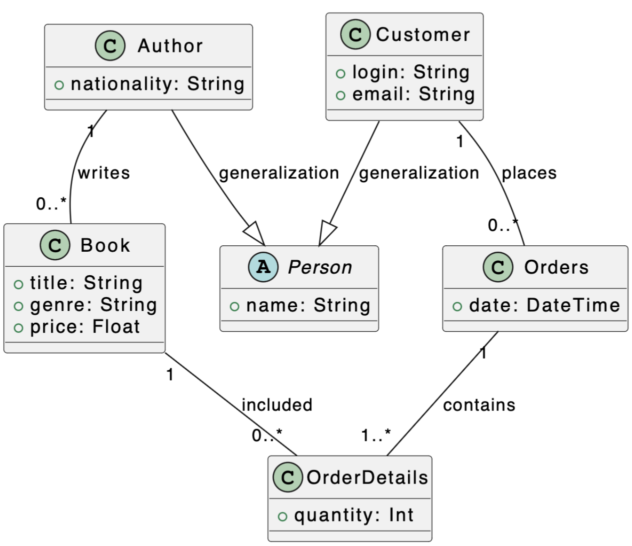



Fig. 3. UML diagram of Library data model

The BUML model presented describes a bookstore system with several interconnected entities: Person, Author, Customer, Book, Order, and OrderDetails.

- **Person Class**: Defined as an abstract class with a single attribute `name` of type string. This class serves as a base for other classes.
  - Attributes: `name` (str).
- **Author Class**: Inherits from Person.
  - Attributes: `nationality` (str).
- **Customer Class**: Inherits from Person.
  - Attributes: `login` (str), `email` (str).
- **Book Class**:
  - Attributes: `title` (str), `genre` (str), `price` (float).
- **Order Class**:
  - Attributes: `date` (datetime).
- **OrderDetails Class**: Intermediate class to represent Book-Customer association.
  - Attributes: `quantity` (int).

The model also establishes relationships between these classes:

- **Author to Book**: A one-to-many relationship where an Author can write many Books. This is represented as a BinaryAssociation named `writes`.
- **Customer to Order**: Indicates that a Customer can place many Orders, also a one-to-many relationship, represented as `places`.
- **Order to OrderDetails**: A one-to-many relationship showing that an Order can include many OrderDetails, represented as `contains`.
- **Book to OrderDetails**: Another one-to-many relationship indicating that a Book can be included in many OrderDetails, represented as `included`.

A UML diagram of this model is presented at figure 3. A BUML view of this model is presented in appendix, 3.

## VI. Manual Implementation

First part of the project consists of creating the listed tools manually, based on requirements. For prototype purposes, it is assumed that every Domain Model consists of following BUML data modelling primitives:

- Class: characterisation of common properties of a set of related real-world objects
- Property: describes attributes of Class, as well as auxiliary objects ('ends') for specifying Associations between classes.
- Association: unites several 'ends', which are referenced to Classes, into a relationship between these Classes with provided in 'ends' multiplicities.
  - BinaryAssociation: specific instances for describing relationship between exactly 2 classes.
- Generalisation: implements inheritance mechanism between Class objects.

### *A. BUML to SQL converter*

At first, let's set the correspondence between BUML data modelling primitives and SQL objects.

- **Classes** in BUML directly translate to tables in SQL Schema.

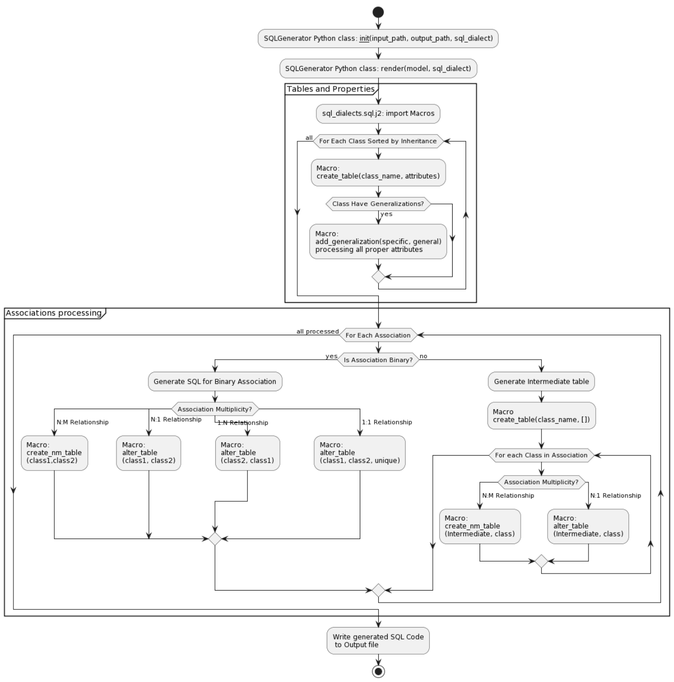



Fig. 4. Activity diagram for BUML to SQL converter

- **Properties** within BUML classes attributes (columns) in the SQL table. The conversion process includes mapping the BUML property data types to SQL data types. For example, a BUML property of type `string` might be mapped to an SQL `VARCHAR` type.
- **Associations** in BUML, representing relationships between entities, are translated into foreign key constraints in SQL. This translation varies depending on the nature of the association: *many-to-many* and *N-ary (with more than 2 ends)* relationships involve an intermediate auxiliary table for representation.
- **Generalisation**, indicative of inheritance in BUML, is translated into a form of table inheritance in SQL. This is typically achieved by creating foreign key references from the derived class table to the base class table.

An activity diagram of BUML to SQL converter architecture is depicted in figure 4. It represents graphically a review of translation process. Note that Jinja template language executes all commands sequentially, one by one, so the final composition of the file will be composed of the same logical blocks in the same order. A more detailed description for this process is presented below.

*1) Template Import and Iteration over Classes:* The process begins with the importation of a Jinja2 template file named `sql_dialects.sql.j2`. This auxiliary file is crucial as it houses template statements for SQL operations such as table creation, column addition, and type conversions. It supports various SQL dialects, making the converter adaptable to different database systems like PostgreSQL, MySQL, and SQLite. New dialects can be seamlessly integrated into the converter's workflow by updating this file.

A `for` loop iterates over each class in the model, where `class_name` and `attributes` are determined for each class. The `sql_templates.create_table` function generates SQL `CREATE TABLE` statements, encapsulating class attributes into SQL table columns, considering the specified SQL dialect.

*2) Processing Associations and Generating SQL Statements:* This segment iterates over each association within the model. Associations in BUML, indicative of the relationships between entities, dictate how relational connections are established in the SQL schema. The type of relationship influences the SQL representation – foreign keys for one-to-many relationships, or linking tables for many-to-many relationships.

- N:M (many-to-many) relationships necessitate an intermediate linking table in SQL. This table bridges the two associated tables by referencing their primary keys. The `sql_templates.create_nm_table` function facilitates the creation of this intermediate table, accurately reflecting the many-to-many relationship.
- N:1 (many-to-one) relationships are implemented by adding a foreign key to the 'many' side, pointing to the 'one' side's unique identifier. The `sql_templates.alter_table` function is employed to modify existing tables, adding these foreign key constraints to enforce the relationships.
- In 1:1 relationships, a unique foreign key reference is created between the two associated tables. This involves adding a foreign key in one of the tables that points to the primary key of the other, often accompanied by a uniqueness constraint. The `sql_templates.alter_table` function is again utilised to create this relationship.

*3) Handling N-ary Associations:* Associations involving more than two entities (N-ary associations) are more complex. They require an intermediate table to represent the multifaceted relationship. The name of this table typically corresponds to the association's name in BUML. The `sql_templates.create_table` function generates the SQL `CREATE TABLE` statement for this purpose.

A loop then processes each end of the N-ary association. For each end, the relationship with the intermediate table is determined based on its multiplicity. If the multiplicity indicates a many-to-many relationship, another linking table is created using `sql_templates.create_nm_table`. If it's a one-to-many relationship, the existing table is altered to add a foreign key using `sql_templates.alter_table`.

*4) Flexibility and Extensibility:* One of the converter's significant strengths is its flexibility and extensibility:

- **Support for Various SQL Dialects**: By abstracting SQL patterns into a separate template file, the converter can easily adapt to different SQL dialects. Adding support for a new dialect involves extending the `sql_dialects.sql.j2` file, without altering the main conversion logic.
- **Modularity**: The modular design of the SQL patterns allows for easy updates and maintenance. Changes in SQL syntax or the addition of new SQL features can be accommodated by updating the relevant snippets in the SQL dialects template.

*5) Integration into platform and code generation:* In order to incorporate a code generation template into the `generators` module of the BESSER project, the module employs a Python interface known as `GeneratorInterface`. This interface is designed to facilitate the processing of the BUML model description and manage the path specifications for both the template and the output. Consequently, a derived class, `SQLGenerator`, inherits these capabilities. Additionally, it is equipped to handle an extra parameter pertaining to the SQL dialect. This selection is important as it influences the generated code.

### *B. BUML to Django models converter*

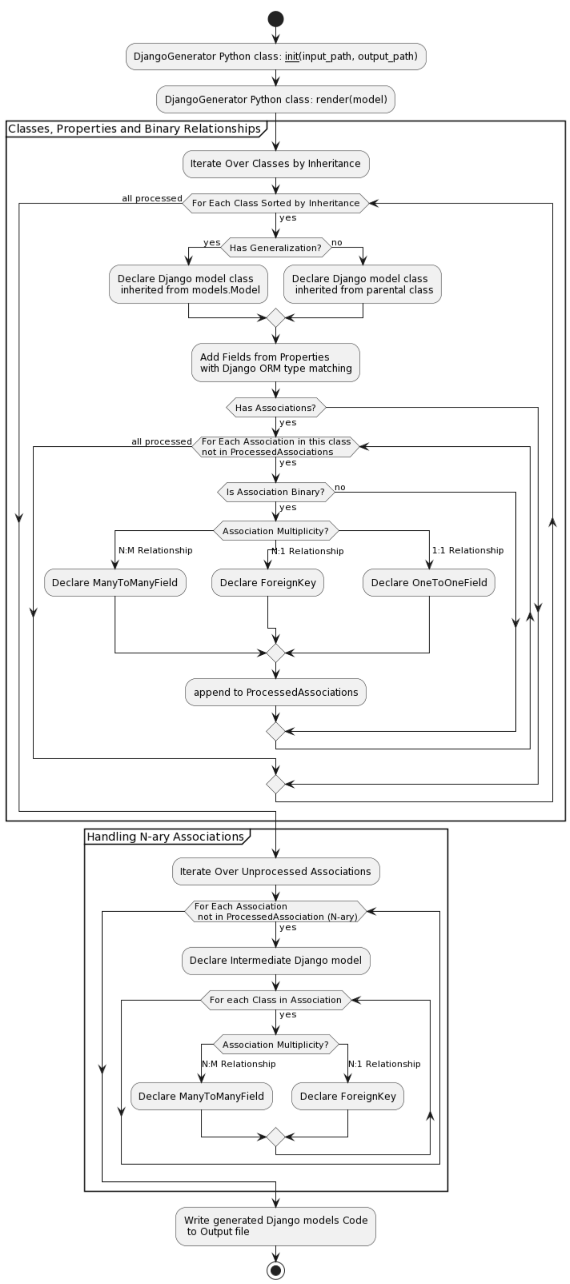



Fig. 5. Activity diagram for BUML to Django converter

Analogically with previous section, let's set the correspondence between BUML data modelling primitives and Django models syntax.

- **Classes** in BUML directly translate to Python classes in Django, extending from Django's `models.Model`.
- **Properties** within BUML classes become class attributes in Django, defined with various `models.Field()` derivatives, depending on their data type and characteristics.
- **Associations** in BUML, representing relationships between entities, are implemented as class attributes in Django models. The multiplicity of these relationships (one-to-many, many-to-many, etc.) defines the use of specific Django fields like `ForeignKey` or `ManyToManyField`.
- **Generalisation**, indicative of inheritance in BUML, is implemented via Python class inheritance in Django. Django ORM supports multiple inheritances, allowing for a flexible and hierarchical structure.

An activity diagram of BUML to Django models converter architecture is depicted in figure 5. The converter operates on a high-level abstraction, parsing the BUML model and dynamically generating Django model classes. This process involves several key steps:

1) **Template Import and Model Definition**: Utilising Jinja2 templating, the converter begins by importing necessary Django modules and auxiliary Jinja templates. These auxiliary templates (`django_fields.py.j2`) contain mappings from BUML property types to Django ORM field types, providing a flexible mechanism for type conversion.
2) **Processing Class Inheritance**: The converter iteratively processes each class in the BUML model, sorted by their inheritance hierarchy. This ensures that base classes are defined before their derived counterparts. For each class, the converter constructs the inheritance chain, which is crucial for preserving the hierarchical relationships in the resulting Django models.
3) **Attribute Definition and Association Processing**: Within each Django model class, the converter defines attributes based on BUML properties, using the appropriate Django field types. It then processes associations between classes. Binary associations are straightforward, typically translating into `ForeignKey` or `ManyToManyField` in Django. However, the complexity arises in handling n-ary associations, which necessitate the creation of intermediate models or tables.
4) **N-ary Relationship Handling**: For associations involving more than two classes (n-ary), the converter generates intermediate Django models. These models encapsulate the complex relationships among multiple classes, ensuring that the resulting Django models accurately reflect the intricacies of the original BUML associations.

Overall, in comparison with building SQL representation of BUML description, these algorithms share common structural approaches, e.g. processing of Binary Relationships. However, Django models also provides some abstractions over classic relational databases schemes, such as internal processing of generalisations and many-to-many relationships, which simplifies template's logic and reduces quantity of conditional branches.

### C. Streamlit template application

Finally, an important part of working with data is an option to visualise it. The initial requirements for this part is to generate a Streamlit application which provides visual insights about the data using as previously Jinja template engine for code generation. Streamlit is chosen as a data visualiser due to its simplicity and multi-functionality out of the box.

Nevertheless, it is crucial to mention that this task is data-dependent, i.e. the types of relevant visualisations are strongly connected with the data stored inside this data model. Thus, only some assumptions can be made on what should be visualised.

After some discussions and different approaches to this objective, this part was designated as a playground for creating a visual environment for developer's purposes. The code represents a generation template of different tables' views, based on BUML model description: a single table views or joint tables based on information about model's Associations. It is assumed, that there exists a corresponding SQL description of this model and the data in this database is consistent.

However, it can be concluded that usage of Jinja templates for generating visualisations in algorithmic way is not the best practice. Firstly, as it was mentioned before, with the limitations of input knowledge only to the structure of data, it is hardly possible to provide any relevant general solution for all possible data models with the same structure. Moreover, usage of Jinja seems unjustified in this context. Everything can be implemented in pure Python without an intermediate template, and it will even optimise the code in several aspects.

## VII. LLM-BASED IMPLEMENTATION

The main idea of this section is to experiment with prompts (requests) to the LLM model in order to reproduce the code, presented in previous section. To compare the correctness of generators result, some example data models were created, which represents all crucial aspects of data model description with BUML: classes, properties, associations (binary and N-ary) and generalisations.

### A. Teaching BUML to the LLM

The generators from previous section are based on domain specific language, which has not been yet widely published in the open network. Thus, the preliminary idea for a successful interaction with LLM model is to give it some context about the technology the answer should be based on. The very first request to the model was about its abilities to understand this level of abstraction and continuously work with it. The answer is affirmative, with some limitations such as *Retention of Knowledge* and *Understanding and Interpretation*, but it still allows to move on to next steps.

Full interaction can be find in appendix section XII-H.

BESSER repository contains documentation about usage of different modeling abstracts, as well as the source code of their implementation. To familiarise the LLM with BUML, a series of prompts were designed, each building on the insights and limitations observed from the previous responses. These prompts included::

1) The LLM was initially provided with the `metamodel-doc.md` from the BESSER-PEARL repository, which included descriptions and usage examples of BUML modeling language types. The model's response indicated an understanding of the DSL's key components and its application in modeling complex systems.
2) Subsequent prompts involved the `structural.py` source code, representing the implementation of BUML classes. Adjustments were made to the prompts, emphasising the importance of class methods, which enhanced the interaction quality.
3) The LLM's ability to apply its understanding was tested by requesting the generation of a BUML description for a data model. Initial attempts by the LLM revealed inaccuracies in structure and implementation. For example, it re-implemented some parts from `stuctural.py` and used it as a part of final code.
   Through iterative prompting and provision of correct examples, the LLM's responses were refined. Notably, challenges were observed in the correct definition of associations, which were addressed through targeted prompts and corrections.
4) After fine-tuning and enhancements, the LLM has generate a fully corrected BUML description for a blogging system. This task demonstrated the successful culmination of the learning process, where the LLM could accurately apply BUML concepts and documentation.

### B. BUML to SQL converter

Having finished with the description of our language, moving towards code generation. An example Library model provided earlier in V-B is used for all further experiments.

The approach to teaching the LLM for BUML to SQL conversion followed a structured, iterative process. Full interaction is presented in appendix XII-I.

*1) Direct conversion:* The project's first objective was to direct the LLM to translate a document from BUML, our proprietary format, directly into SQL, which refers to the usage of LLM as "a black box" for low-code platform services. This initial task aimed to evaluate the LLM's capacity to understand and accurately convert the structure and relationships defined in a BUML model into an SQL schema. The initial response from the LLM was promising, as it successfully created an SQL schema that reflected the primary structure of the provided BUML model. However, it was observed that the LLM had misinterpreted certain relationships.

After identifying the misinterpretation, the LLM was guided to correct the error. Thus, this first objective is considered accomplished, since ChatGPT generated fully correct and equivalent SQL schema having BUML description as input.

*2) LLM-generated template for algorithmic code generation:* Progressing further, the task was stated more generally and posed a greater complexity: to use the knowledge about the BUML language acquired so far to implement a code generation template for converting BUML to SQL. This step was crucial in evaluating the LLM's ability to replicate and apply the previously demonstrated understanding in a broader context. The initial response from the LLM involved providing a Jinja template designed to process a generic BUML model and generate corresponding SQL code. This template was a significant leap forward, incorporating various macros and logic to create SQL table definitions for classes, manage properties, handle generalisations (inheritance), and establish associations.

Nevertheless, the generated solution on first iteration has several significant drawbacks in terms of processing the relationships.

- N-ary relationships were ignored: the processing of relationships were limited only to binary. However, it was mentioned in the comments to the code, that LLM assumed all associations as binary.
- The associations (relationships) between classes in BUML are stored without any ordering. That is, while iterating over a set of relationships of particular class and processing them in-place, inside SQL Table definition, this can potentially generate a Foreign Key constraint to another SQL Table, which has not been declared yet.
- Generalisations were not processed correctly. Inside Domain Model, classes are stored inside unordered set. Thus, while processing class by class, the code could possibly process firstly more specific class, referencing it to a non-defined general. For example, in listing 1 presented an example of incorrect SQL query, which was generated by LLM-generated template: more specific class `Author` references to a non-defined general class `Person`. Hence, the script should use the provided in `structural.py` function `classes_sorted_by_inheritance()` from `DomainModel` class.

```
CREATE TABLE IF NOT EXISTS Author (
    Author_id SERIAL PRIMARY KEY,
    nationality varchar(100),
    Person_id INT REFERENCES Person(Person_id)
);

CREATE TABLE IF NOT EXISTS Person (
    Person_id SERIAL PRIMARY KEY,
    name varchar(100)
);

```

Listing 1. Example of incorrect SQL queries generation due to wrong order of parent and child classes

It is important to highlight that during the manual implementation of the code generation template, the task of processing N-ary relationships and Generalisations presented substantial challenges. This aspect of development proved to be exceptionally arduous and complex. Furthermore, this implementation phase led to the identification of certain limitations within the structure of the BUML language. These identified deficiencies were subsequently addressed and rectified, thereby enhancing the overall robustness and efficacy of the BUML framework.

The Language Model (LLM) has shown a remarkable ability to correct its mistakes when these are clearly pointed out. After several rounds of clarification, the LLM was able to produce an accurate code generation template. This template was then tested successfully on the earlier mentioned examples. As a result, the LLM completed the task effectively, providing detailed explanations of how the generated code functions.

### C. BUML to Django Models Converter

This subsection focuses on extending the capabilities of the BESSER low-code platform by integrating a BUML to Django models converter, a task that shares similarities with the SQL conversion component but also presents some challenges.

The full interaction process is documented in the appendix (see appendix XII-J). This section aims to provide a strategic overview of the prompt chain methodology and the learning curve involved in optimising the conversion process.

*1) Direct conversion:* The initial experiment involved translating a data model described in BUML format into the Django models format. The first attempt by the Large Language Model (LLM) resulted in a `models.py` file, which, however, included a common misinterpretation of the relationship type. This error indicated a potential gap in the model's initial comprehensive analysis of the data model.

Recognising this error, the LLM was prompted to correct its output. The refined response successfully adjusted the relationship type. This iterative refinement process exemplifies the adaptive learning capability of the LLM in response to targeted feedback.

*2) LLM-generated template for algorithmic code generation:* Building on the insights gained from the SQL conversion experience, the task was expanded to a more complex scenario: generating a Jinja template for the Django model converter. This time, potential issues identified in the SQL part, such as handling N-ary relationships and the order of class definitions, were preliminary addressed in the prompts.

The LLM responded to a complex prompt by creating an advanced Jinja template. This template was similar in structure to what was seen with SQL, but it was specially designed for Django models. The template managed to avoid the issues confronted earlier, showing that the LLM can use what it learns in one situation and apply it to another. However, there was a small mistake related to a method from `structural.py` that didn't exist, which the LLM fixed quickly after getting feedback.

In summary, the experiment demonstrated that the Language Model (LLM) solved the problem more quickly when its initial weaknesses were clearly pointed out. To keep the experiment fair, the model didn't have access to the results from working with the SQL generator. It only received information related to the Django generator. This method made sure that the evaluation of the LLM's ability to handle the task was unbiased and accurate.

### D. Streamlit template application

This section summaries the interaction with the Large Language Model (LLM) in generating a visual representation for data models described using BUML. The context for this experiment was the library example model (section V-B), with detailed interactions documented in Appendix XII-K. The experiment assumes the availability of a data model and corresponding data, structured using the SQL schema from the BUML-to-SQL converter

With the assumptions stated, the initial task was to generate a Streamlit web application code capable of visualising the BUML-described data model, including all table views and join combinations. The LLM's response provided a basic structure for the Streamlit application, encompassing the setup for PostgreSQL database connection, data retrieval, and UI elements in Streamlit. This foundational code included options for viewing tables and join combinations, alongside placeholders for data visualisations. However, the model's output was limited, offering only singular examples for each visualisation type: one plot and one table view.

Recognising the need for a broader range of visualisations, the following prompt requested the model to generate additional plots and graphs (3-4 types) based on the data schema. The LLM's enhanced response included diverse visualisations such as the number of books per author, average book prices by genre, customer order frequencies, and temporal sales analysis. This phase highlighted the model's ability to apply general knowledge and intuitive logic to the data, blending algorithmic precision with a human-like approach to data interpretation. Examples of generated data views are available in Appendix section XII-A.

A notable aspect of this interaction was the insistence on comprehensive, shortcut-free code generation in prompts. Despite this clear directive, the LLM occasionally returned incomplete answers, underscoring the importance of iterative refinement in the prompt strategy. Each interaction served as a learning opportunity, progressively honing the model's output to align more closely with the project's requirements.

## VIII. Results, Common Mistakes and Assessment

### A. Overview of Results

The project's results are overall satisfactory. The set objectives, involving the implementation of a series of tasks using a DSL for data modeling, were achievable with the support of a LLM. The assistance of the LLM notably enhanced the efficiency and speed of project completion. The tasks involved interpreting and converting data models described in BUML into other formats like Django models and SQL schemes. The LLM demonstrated a considerable understanding of these concepts, providing accurate and insightful interpretations and conversions.

### B. Common ChatGPT's behavioural patterns

Despite the overall success, some common patterns/mistakes/approaches were identified in the LLM's outputs, reflecting the current limitations in AI-driven code generation and model interpretation.

*1) Use of General Knowledge:* In tasks requiring the conversion of models described in BUML to other formats, the LLM occasionally misinterpreted the input, leading to logical errors. For instance, when dealing with a model involving humans and dogs, the LLM's interpretation slightly deviated from the expected logical structure. However, it's noteworthy that such misinterpretations were not consistently detrimental. In tasks focused on visualisation, the LLM adeptly utilised its general knowledge to enhance the output, showcasing its ability to apply broad understanding effectively in certain contexts.

*2) Code Analysis:* The LLM was provided with a structural code of BUML and detailed interpretations of each significant abstract's use case. Despite this, there were instances where the LLM's outputs were misaligned with the provided examples. Specifically, 1) In one case, the BUML model was inaccurately generated, completely overlooking the provided usage examples of BUML; 2) In another instance, while generating code for model converters, the LLM referred to methods that did not exist in the provided structural code, indicating a misinterpretation of the initial structural file. These instances underscore the challenges faced by LLMs in accurately parsing and applying complex, context-specific programming instructions.

*3) ChatGPT's Approach to Code Generation:* On several occasions, the model demonstrated a tendency to provide only partial solutions or code snippets when tasked with generating code based on specific requirements. This was observed even when explicit requests were made for complete code outputs. In some cases, ChatGPT ignored the request for a comprehensive solution, providing only fragments of the required code. This behaviour can be attributed to the LLM's inherent design, which may prioritise brevity or simplicity over completeness, especially in complex coding tasks. This aspect of the LLM's performance highlights the need for careful oversight and possibly iterative querying when using AI tools for detailed software development tasks.

### C. Assessment

The project's requirements have been largely satisfied, and the outcomes of the LLM demonstrate a high level of precision, particularly when compared with human-written code. The LLM's ability to interpret and convert complex data models, as evidenced in tasks involving BUML, showcases its proficiency and potential as a tool in software development. The project involved converting BUML-described models into other formats such as Django models and SQL schemes. The LLM handled these tasks with notable accuracy, demonstrating its understanding of different data modeling paradigms and programming languages.

In discussing two distinct applications of the Language Model (LLM) – as a creator of the final code version and as a generator of templates for subsequent template-based generation – it is evident from the conducted experiments that the LLM was unable to flawlessly execute either task on its initial attempt. Instead, an iterative method with progressive refinements was necessary. Consequently, for potential future integration of an AI component into a low-code platform, it is essential that the model is fine-tuned for self-enhancement of its solutions and adept handling of complex data modeling challenges.

The task of translating one data modeling format into another can be characterised as algorithmic in nature. Although various methodologies exist for describing algorithms, given its algorithmic essence, this task is doable for a template-based code generation approach, which has proven to be more efficacious, due to its deterministic nature. It is important to note that if the LLM were pre-equipped with comprehensive knowledge of potential challenges in generating templates for code generation, including the nuances of data modeling, it would be more adept at tackling advanced tasks such as template generation rather than executing straightforward code generation via prompts. Consequently, LLM-to-template is seemed to be more suitable for low-code platforms purposes.

Nonetheless, the LLM possesses the capability to generate complete solutions without the need for intermediate templates, which can be particularly useful for demonstration or prototyping purposes. While these solutions may not be entirely precise, they still fulfil the requirements, albeit with some minor discrepancies.

It is important to mention that in the visualisation problem in the context of this project, artificial intelligence bypassed the human solution to the problem by basing its solution on publicly available knowledge. Certainly, no universal solution for visualising BUML models has been obtained. This is a very complex task that will require more time. Perhaps, to effectively solve the visualisation problem, it will be possible to use additional DSL for a more detailed description of the data.

The success in basic tasks sets the stage for more complex coding challenges. Future projects might include creating Django views, where the AI translates data models into website components. This would show the AI's skills with web frameworks and MVC architecture. Making visual examples could also help simplify complex data models for a wider audience, including non-experts.

Furthermore, the project could benefit from the LLM suggesting and implementing a data management format of its own choice. This would provide an insight into the LLM's creative capabilities and its ability to innovate beyond the given parameters. Such a task would push the boundaries of the LLM's application in software development, paving the way for more autonomous and creative AI-driven coding solutions.

## IX. Conclusion

The assessment of the LLM-generated code, in comparison with human-written code, underscores the significant potential of Language Learning Models in the field of software development. The LLM has exhibited commendable proficiency in interpreting and converting complex data models, maintaining a high degree of precision and adhering closely to the specified requirements. This performance reinforces the growing hypothesis that the future of software development is increasingly intertwined with the utilisation of LLMs. While these models have not reached a point where they can fully replace human developers, their ability to assist productively is undeniable. They augment the development process, offering speed and efficiency, particularly in tasks that require rapid prototyping or the translation of complex models into various programming languages and formats. In summary, LLMs stand as valuable collaborators in the software development landscape, enhancing the capabilities of human developers and shaping the future of the industry.

Certainly, in context of the project, a contribution was made to BESSER project, including BUML-to-Django and BUML-to-SQL model description conversions, then can be found in the repository of project [9], in `src/generators` section. Parts of the code that were not included in the final contribution can be found in the project's repository [4], where all the work within that project is stored.

## Acknowledgements

Author would like to thank the BiCS management and education team for course organisation, Luxembourg Institute of Science and Technology and personally Jordi Cabot Sagrera, Ivan David Alfonso Diaz and Aaron Conrardy for productive cooperation, support and feedback.

## X. Plagiarism statement

I declare that I am aware of the following facts:

- I understand that in the following statement the term ”person” represents a human or any automated generation system.
- As a student at the University of Luxembourg I must respect the rules of intellectual honesty, in particular not to resort to plagiarism, fraud or any other method that is illegal or contrary to scientific integrity.
- My report will be checked for plagiarism and if the plagiarism check is positive, an internal procedure will be started by my tutor. I am advised to request a pre-check by my tutor to avoid any issue.
- As declared in the assessment procedure of the University of Luxembourg, plagiarism is committed whenever the source of information used in an assignment, research report, paper or otherwise published/circulated piece of work is not properly acknowledged. In other words, plagiarism is the passing off as one's own the words, ideas or work of another person, without attribution to the author. The omission of such proper acknowledgement amounts to claiming authorship for the work of another person. Plagiarism is committed regardless of the language of the original work used. Plagiarism can be deliberate or accidental. Instances of plagiarism include, but are not limited to:
  1) Not putting quotation marks around a quote from another person's work
  2) Pretending to paraphrase while in fact quoting
  3) Citing incorrectly or incompletely
  4) Failing to cite the source of a quoted or paraphrased work

5) Copying/reproducing sections of another person's work without acknowledging the source
6) Paraphrasing another person's work without acknowledging the source
7) Having another person write/author a work for oneself and submitting/publishing it (with permission, with or without compensation) in one's own name ('ghost-writing')
8) Using another person's unpublished work without attribution and permission ('stealing')
9) Presenting a piece of work as one's own that contains a high proportion of quoted/copied or paraphrased text (images, graphs, etc.), even if adequately referenced

Auto- or self-plagiarism, that is the reproduction of (portions of a) text previously written by the author without citing that text, i.e. passing previously authored text as new, may be regarded as fraud if deemed sufficiently severe.

# XI. References Section

## References


[1] Hana A ALSAADI, Dhefaf T RADAIN, Maysoon M ALZAHRANI, Wahj F ALSHAMMARI, Dimah ALAHMADI, and Bahjat FAKIEH. Factors that affect the utilization of low-code development platforms: survey study. *Romanian Journal of Information Technology & Automatic Control/Revista Română de Informatică și Automatică*, 31(3), 2021. link.

[2] Adam Bochenek. Can chatgpt replace a template-based code generator? 2023. link.

[3] Jordi Cabot and Robert Clarisó. Low code for smart software development. *IEEE Software*, 40(1):89–93, 2023. link.

[4] Fedor Chikhachev. Bsp-s3 technical deliverable source code. link.

[5] Enrique Dehaerne, Bappaditya Dey, Sandip Halder, Stefan De Gendt, and Wannes Meert. Code generation using machine learning: A systematic review. *Ieee Access*, 2022. link.

[6] IBM Education. Ai code-generation software: What it is and how it works. link.

[7] Django Software Foundation. Django framework website. link.

[8] Naim Kabir. Jinja + sql; macros for maintainable, testable data analytics. link.

[9] LIST. FNR PEARL BESSER project. link.

[10] Ipek Ozkaya, Anita Carleton, John E. Robert, and Douglas Schmidt. Application of large language models (LLMs) in software engineering: Overblown hype or disruptive change? link.

[11] Ahmed R Sadik, Sebastian Brulin, and Markus Olhofer. Coding by design: Gpt-4 empowers agile model driven development. *arXiv preprint arXiv:2310.04304*, 2023. link.

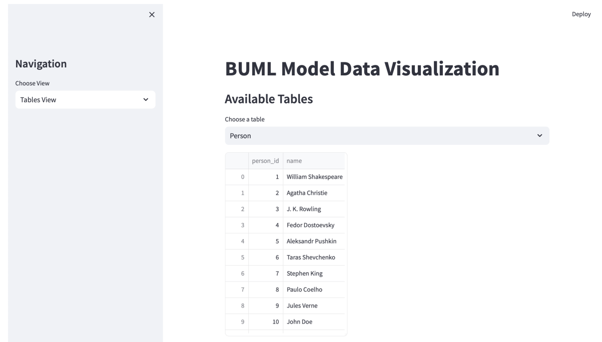
[12] Jordi Cabot Sagrera. From models-to-code to models-to-prompts: the next mde revolution? link.

[13] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. Automatic generation of programming exercises and code explanations using large language models. In *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1*, pages 27–43, 2022. link.

[14] Streamlit. Streamlit website. link.

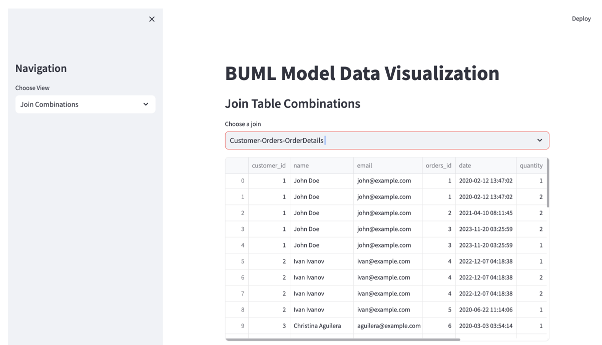

# XII. Appendix

## A. Streamlit: views generated by ChatGPT




| | person_id | name |
|---|---|---|
| 0 | 1 | William Shakespeare |
| 1 | 2 | Agatha Christie |
| 2 | 3 | J. K. Rowling |
| 3 | 4 | Fedor Dostoevsky |
| 4 | 5 | Aleksandr Pushkin |
| 5 | 6 | Taras Shevchenko |
| 6 | 7 | Stephen King |
| 7 | 8 | Paulo Coelho |
| 8 | 9 | Jules Verne |
| 9 | 10 | John Doe |

Fig. 6. Raw table views




| | customer_id | name | email | orders_id | date | quantity |
|---|---|---|---|---|---|---|
| 0 | 1 | John Doe | john@example.com | 1 | 2020-02-12 13:47:02 | 1 |
| 1 | 1 | John Doe | john@example.com | 1 | 2020-02-12 13:47:02 | 2 |
| 2 | 1 | John Doe | john@example.com | 2 | 2021-04-10 08:11:45 | 2 |
| 3 | 1 | John Doe | john@example.com | 3 | 2023-11-20 03:25:59 | 2 |
| 4 | 1 | John Doe | john@example.com | 3 | 2023-11-20 03:25:59 | 1 |
| 5 | 2 | Ivan Ivanov | ivan@example.com | 4 | 2022-12-07 04:18:38 | 1 |
| 6 | 2 | Ivan Ivanov | ivan@example.com | 4 | 2022-12-07 04:18:38 | 2 |
| 7 | 2 | Ivan Ivanov | ivan@example.com | 4 | 2022-12-07 04:18:38 | 2 |
| 8 | 2 | Ivan Ivanov | ivan@example.com | 5 | 2020-06-22 11:14:06 | 1 |
| 9 | 3 | Christina Aguilera | aguilera@example.com | 6 | 2020-03-03 03:54:14 | 1 |

Fig. 7. Join table views

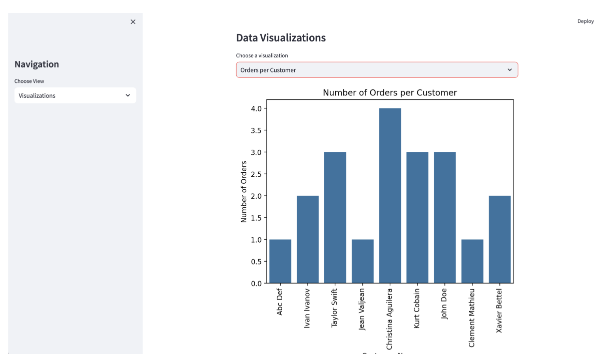



Fig. 8. Plot with aggregations functions

### B. Streamlit: views generated by template generation

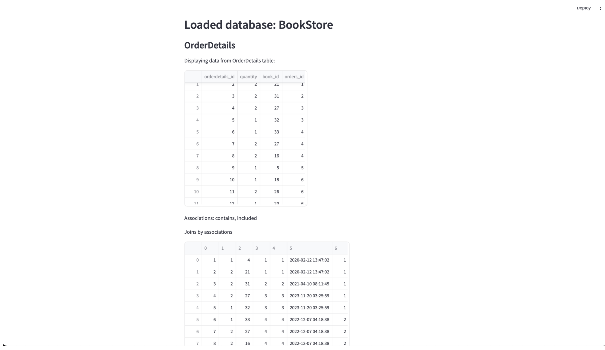



Fig. 9. Template-generated Table view page

### C. Library Model BUML

```python
from BUML.metamodel.structural.structural import Association, Class, Multiplicity, Property, BinaryAssociation, PrimitiveDataType, DomainModel

movie_year: Property = Property(
    name="year", owner=None, property_type=PrimitiveDataType('int'))
movie_length: Property = Property(
    name="length", owner=None, property_type=PrimitiveDataType('int'))
movie_title: Property = Property(
    name="title", owner=None, property_type=PrimitiveDataType('str'))

studio_name: Property = Property(
    name="name", owner=None, property_type=PrimitiveDataType('str'))
studio_address: Property = Property(
    name="address", owner=None, property_type=PrimitiveDataType('str'))

star_name: Property = Property(
    name="name", owner=None, property_type=PrimitiveDataType('str'))
star_age: Property = Property(
    name="age", owner=None, property_type=PrimitiveDataType('int'))

# Classes definition
movie_class: Class = Class(name="movie", attributes=set(
    [movie_length, movie_year, movie_title]))
studio_class: Class = Class(
    name="studio", attributes=set([studio_name, studio_address]))
star_class: Class = Class(name="star", attributes=set([star_name, star_age]))


# Ends definition
# One star can play in many movies, in one movie there are many stars
movie_starring: Property = Property(
    name="starring", owner=None, property_type=movie_class, multiplicity=Multiplicity(0, "*"))
star_starred_in: Property = Property(
    name="starredIn", owner=None, property_type=star_class, multiplicity=Multiplicity(0, "*"))

# Movie can be owned only by one studio
movie_owned_by: Property = Property(
    name="ownedBy", owner=None, property_type=movie_class, multiplicity=Multiplicity(0, "*"))
studio_owns: Property = Property(
    name="owns", owner=None, property_type=studio_class, multiplicity=Multiplicity(1, 1))

star_contract: Property = Property(
    name="star_contract", owner=None, property_type=star_class, multiplicity=Multiplicity(0, "*"))
studio_contract: Property = Property(
    name="studio_contract", owner=None, property_type=studio_class, multiplicity=Multiplicity(1, 1))
movie_contract: Property = Property(
    name="movie_contract", owner=None, property_type=movie_class, multiplicity=Multiplicity(0, "*"))

# BinaryAssociation definition
relation_stars: BinaryAssociation = BinaryAssociation(
    name="stars", ends={movie_starring, star_starred_in})
relation_owns: BinaryAssociation = BinaryAssociation(
    name="owns", ends={movie_owned_by, studio_owns})

# Association definition
relation_contracts: Association = Association(
    name="contract", ends={star_contract, studio_contract, movie_contract})

model_cinema: DomainModel = DomainModel(name="Cinema", types={
    movie_class, studio_class, star_class}, associations={relation_owns, relation_stars, relation_contracts}, generalizations=None, packages=None, constraints=None)
```

Listing 2. Example BUML model: Cinema

### D. Cinema Model BUML

```python
from BESSER.src.BUML.metamodel.structural.structural import Association, Class, Multiplicity, Property, BinaryAssociation, PrimitiveDataType, DomainModel, Generalization


person_name: Property = Property(
    name="name", owner=None, property_type=PrimitiveDataType('str'))

person_class: Class = Class(name="Person", attributes=[person_name])

author_natonality: Property = Property(
    name="nationality", owner=None, property_type=PrimitiveDataType('str'))

authors_class: Class = Class(name="Author", attributes=set(
    [author_natonality]))

generalization_person_author: Generalization = Generalization(general=person_class, specific=authors_class)

book_title: Property = Property(
    name="title", owner=None, property_type=PrimitiveDataType('str'))
book_genre: Property = Property(
    name="genre", owner=None, property_type=PrimitiveDataType('str'))
book_price: Property = Property(
    name="price", owner=None, property_type=PrimitiveDataType('float'))

```

```
books_class: Class = Class(name="Book", attributes=set(
    [book_title, book_genre, book_price]))

customer_email: Property = Property(
    name="email", owner=None, property_type=PrimitiveDataType('str'))

customers_class: Class = Class(name="Customer", attributes=set(
    [customer_email]))

generalization_person_customer: Generalization = Generalization(
    general=person_class, specific=customers_class)

order_date: Property = Property(
    name="date", owner=None, property_type=PrimitiveDataType('datetime'))

orders_class: Class = Class(name="Order", attributes=set(
    [order_date]))

details_quantity: Property = Property(
    name="quantity", owner=None, property_type=PrimitiveDataType('int'))

details_class: Class = Class(name="OrderDetails", attributes=set(
    [details_quantity]))

# Ends definition
# one Author - many Books
author_writes: Property = Property(
    name="writes", owner=None, property_type=authors_class, multiplicity=Multiplicity(1, 1))
book_writes: Property = Property(
    name="writes", owner=None, property_type=books_class, multiplicity=Multiplicity(0, "*"))

# One customer - many orders
customer_places: Property = Property(
    name="places", owner=None, property_type=customers_class, multiplicity=Multiplicity(1, 1))
order_places: Property = Property(
    name="places", owner=None, property_type=orders_class, multiplicity=Multiplicity(0, "*"))

# Many Order - Many Books => intermediate table OrderDetails

# One Order - many OrderDetails
order_contains: Property = Property(
    name="contains", owner=None, property_type=orders_class, multiplicity=Multiplicity(1, 1))
details_contains: Property = Property(
    name="contains", owner=None, property_type=details_class, multiplicity=Multiplicity(1, "*"))

# One book - many OrderDetails
book_included: Property = Property(
    name="included", owner=None, property_type=books_class, multiplicity=Multiplicity(1, 1))
details_included: Property = Property(
    name="included", owner=None, property_type=details_class, multiplicity=Multiplicity(0, "*"))

# BinaryAssociation definition
relation_writes: BinaryAssociation = BinaryAssociation(
    name="writes", ends={author_writes, book_writes})
relation_places: BinaryAssociation = BinaryAssociation(
    name="places", ends={order_places, customer_places})
relation_contains: BinaryAssociation = BinaryAssociation(
    name="contains", ends={order_contains, details_contains})
relation_included: BinaryAssociation = BinaryAssociation(
    name="included", ends={book_included, details_included})


library_model: DomainModel = DomainModel(name="Cinema", types={
    authors_class, books_class, customers_class, orders_class, details_class}, associations={relation_writes, relation_places, relation_contains, relation_included}, generalizations=None, packages=None, constraints=None)
```

Listing 3. Example BUML model: Library

## E. Manual implementation: BUML to SQL

```
{% import "sql_dialects.sql.j2" as sql_templates %}
{# Iterate over classes and generate CREATE TABLE statements #}
{% for class_obj in model.classes_sorted_by_inheritance() %}
    {% set class_name = class_obj.name %}
    {% set attributes = class_obj.attributes %}
    {{- sql_templates.create_table(class_name, attributes, types, sql_dialect) }}
    {# Handling generalizations (inheritance) #}
    {% if class_obj.generalizations %}
        {% for generalization in class_obj.generalizations %}
            {% if generalization.specific.name == class_name %}
                {{- sql_templates.add_generalization(class_name, generalization, sql_dialect) }}
            {% endif %}
        {% endfor %}
    {% endif %}
{% endfor %}
{# Iterate over associations and generate SQL statements based on association type #}
{% for association in model.associations %}
    {% if association.ends|length == 2 -%}
        {% set ns = namespace(end1=None, end2=None) %}
        {% for end in association.ends %}
            {% set ns.end1=end if loop.index == 1 else ns.end1 %}
            {% set ns.end2=end if loop.index == 2 else ns.end2 %}
        {% endfor %}
        {% set class1_name = ns.end1.type.name %}
        {% set class2_name = ns.end2.type.name %}
        {# Check multiplicity and generate appropriate SQL statement #}
        {% if ns.end1.multiplicity.max > 1 and ns.end2.multiplicity.max > 1 %}
        {# N:M Relationship: Create intermediate table #}
            {{- sql_templates.create_nm_table(class1_name, class2_name, sql_dialect) }}
        {% elif ns.end1.multiplicity.max > 1 and ns.end2.multiplicity.max == 1 %}
        {# N:1 Relationship: Add reference to Nary end #}
            {{- sql_templates.alter_table(class1_name, class2_name, class2_name, class2_name, sql_dialect) }}
```

```
        {% elif ns.end1.multiplicity.max == 1 and ns.end2.multiplicity.max > 1 %}
        {# 1:N Relationship: Add reference to Nary end #}
            {{- sql_templates.alter_table(class2_name, class1_name, class1_name, class1_name, sql_dialect) }}
        {% elif ns.end1.multiplicity.max == 1 and ns.end2.multiplicity.max == 1 %}
        {# 1:1 Relationship: Add unique reference to one of the ends #}
            {{- sql_templates.alter_table(class1_name, class2_name, class2_name, class2_name, sql_dialect) }}
        {% endif %}
    {% else %}
    {# Convert Nary relationship to Binary relationships and intermediate table #}
        {% set intermediate_class = association.name %}
        {{- sql_templates.create_table(intermediate_class, [], types, sql_dialect) }}
        {% set end_intermediate_multiplicity_max = 9999 %}
        {% for end in association.ends %}
            {# Check multiplicity and generate appropriate SQL statement #}
            {% set class1_name = intermediate_class %}
            {% set class2_name = end.type.name %}
            {# End of intermediate table should have multiplicity of the original end to the class #}
            {% if end.multiplicity.max > 1 and end_intermediate_multiplicity_max > 1 %}
            {# N:M Relationship: Create intermediate table #}
                {{- sql_templates.create_nm_table(class1_name, class2_name, sql_dialect) }}
            {% elif end.multiplicity.max == 1 and end_intermediate_multiplicity_max > 1 %}
                {# 1:N Relationship: Add reference to Nary end #}
                {{- sql_templates.alter_table(class1_name, class2_name, class2_name, class2_name, sql_dialect) }}
            {% endif %}
        {% endfor %}
    {% endif %}
{%- endfor %}
```

Listing 4. Manual implementation: BUML to SQL

## F. Manual implementation: BUML to Django models

```
{# Django model template generator #}
{% import "django_fields.py.j2" as django_fields %}
{# django_fields.j2 should contain mappings to Django's ORM field types #}
from django.db import models

{% set processed_associations = [] %}
{% for class_obj in model.classes_sorted_by_inheritance() %}
    {% set generals = namespace(names=[]) %}
    {% for parent in class_obj.parents() %}
        {% set _ = generals.names.append(parent.name) %}
    {% endfor %}
    {% set inheritance = generals.names | join(', ') %}
class {{ class_obj.name }}({{ inheritance | default('models.Model', true) }}):
    {% for attr in class_obj.attributes %}
    {{ attr.name }} = {{ django_fields.get_field(attr.type.name, attr.properties) }}
    {%- endfor %}
    {% for association in class_obj.associations %}
        {% if association.ends|length == 2 and association.name not in processed_associations %}
            {% set ns = namespace(end1=None, end2=None) %}
            {% for end in association.ends %}
                {% set ns.end1=end if end.type.name == class_obj.name else ns.end1 %}
                {% set ns.end2=end if end.type.name != class_obj.name else ns.end2 %}
            {% endfor %}
            {% set class1_name = ns.end1.type.name %}
            {% set class2_name = ns.end2.type.name %}
            {% if ns.end1.multiplicity.max > 1 and ns.end2.multiplicity.max > 1 %}
            {# N:M Relationship: Use ManyToManyField in Django #}
    {{ class2_name.lower() }} = models.ManyToManyField('{{ class2_name }}', on_delete=models.CASCADE)
                {% do processed_associations.append(association.name) %}
            {% elif ns.end1.multiplicity.max > 1 and ns.end2.multiplicity.max == 1 %}
            {# N:1 Relationship: Use ForeignKey in Django #}
    {{ class2_name.lower() }} = models.ForeignKey('{{ class2_name }}', on_delete=models.CASCADE)
                {% do processed_associations.append(association.name) %}
            {% elif ns.end1.multiplicity.max == 1 and ns.end2.multiplicity.max == 1 %}
            {# 1:1 Relationship: Use OneToOneField in Django #}
    {{ class2_name.lower() }} = models.OneToOneField('{{ class2_name }}', on_delete=models.CASCADE)
                {% do processed_associations.append(association.name) %}
            {% endif %}
        {% endif %}
    {% endfor %}

    def __str__(self):
        return str(self.id)

{% endfor %}

{% for association in model.associations %}
    {% if association.name not in processed_associations %}
{# Nary relationship handling (intermediate table) #}
class {{ association.name|capitalize }}(models.Model):
        {% for end in association.ends %}
            {% set class_name = end.type.name %}
            {% if end.multiplicity.max > 1 %}
                {# N:M Relationship: Use ManyToManyField in Django  intermediate table #}
    {{ class_name.lower() }} = models.ManyToManyField({{ class_name }}, on_delete=models.CASCADE)
            {% else %}
                {# 1:N Relationship: Use ForeignKey in Django  intermediate table #}
    {{ class_name.lower() }} = models.ForeignKey({{ class_name }}, on_delete=models.CASCADE)
            {% endif %}
        {% endfor %}
    {% endif %}
```

```
{% endfor %}
```

Listing 5. Manual implementation: BUML to Django models

## G. Manual implementation: Streamlit visualisations

```
import streamlit as st
import psycopg2
import pandas as pd

def execute_query(query):
    cursor.execute(query)
    data = cursor.fetchall()
    return data

st.title("{{ page_title }}")

db_params = {
    "host": "{{ host }}",
    "database": "{{ database }}",
    "user": "{{ user }}",
    "password": "{{ password }}",
}

conn = psycopg2.connect(**db_params)
cursor = conn.cursor()

st.header("Loaded database: {{ model.name }}")

{% for class in model.get_classes() %}
st.subheader("{{ class.name }}")
st.write(f"Displaying data from {{ class.name }} table:")
query = "SELECT * FROM {{ class.name }};"
df = pd.read_sql(query, conn)
# Display the DataFrame with column names
st.dataframe(df)

associations = "Associations: "
{% for relation in class.associations %}
associations += "{{relation.name}}, "
{% endfor %}
associations = associations.rstrip(", ")
st.write(associations)

st.write("Joins by associations")
{% for association in class.associations %}
    {% if association.ends|length == 2 -%}
        {% set ns = namespace(end1=None, end2=None) %}
        {% for end in association.ends %}
            {% set ns.end1=end if end.type.name == class.name else ns.end1 %}
            {% set ns.end2=end if end.type.name != class.name else ns.end2 %}
        {% endfor %}
        {% set class1_name = ns.end1.type.name %}
        {% set class2_name = ns.end2.type.name %}
        {# Check multiplicity and generate appropriate SQL statement #}
        {% if ns.end1.multiplicity.max > 1 and ns.end2.multiplicity.max > 1 %}
        {# N:M Relationship: JOIN via intermediate table #}
try:
    query = f"""SELECT {{ class1_name }}.*, {{ class2_name }}.* FROM {{ class1_name }}
        JOIN
            {{ class1_name }}_{{ class2_name }} ON {{ class1_name }}.{{ class1_name }}_id = {{ class1_name }}_{{ class2_name }}.{{ class1_name }}_id
        JOIN
            {{ class2_name }} ON {{ class2_name }}.{{ class2_name }}_id = {{ class1_name }}_{{ class2_name }}.{{ class2_name }}_id"""
except:
    query = f"""SELECT {{ class1_name }}.*, {{ class2_name }}.* FROM {{ class1_name }}
        JOIN
            {{ class2_name }}_{{ class1_name }} ON {{ class1_name }}.{{ class1_name }}_id = {{ class2_name }}_{{ class1_name }}.{{ class1_name }}_id
        JOIN
            {{ class2_name }} ON {{ class2_name }}.{{ class2_name }}_id = {{ class2_name }}_{{ class1_name }}.{{ class2_name }}_id"""
        {% elif ns.end1.multiplicity.max > 1 and ns.end2.multiplicity.max == 1 %}
        {# N:1 Relationship: JOIN natural #}
query = f"""SELECT {{ class1_name }}.*, {{ class2_name }}.* FROM {{ class1_name }}
        JOIN
            {{ class2_name }} ON {{ class1_name }}.{{ class2_name }}_id = {{ class2_name }}.{{ class2_name }}_id"""
        {% elif ns.end1.multiplicity.max == 1 and ns.end2.multiplicity.max > 1 %}
query = f"""SELECT {{ class1_name }}.*, {{ class2_name }}.* FROM {{ class2_name }}
        JOIN
            {{ class1_name }} ON {{ class2_name }}.{{ class1_name }}_id = {{ class1_name }}.{{ class1_name }}_id"""
        {% elif ns.end1.multiplicity.max == 1 and ns.end2.multiplicity.max == 1 %}
        {# 1:1 Relationship: JOIN natural #}
try:
    query = f"""SELECT {{ class1_name }}.*, {{ class2_name }}.* FROM {{ class1_name }}
        JOIN
            {{ class2_name }} ON {{ class1_name }}.{{ class2_name }}_id = {{ class2_name }}.{{ class2_name }}_id"""
except:
    query = f"""SELECT {{ class1_name }}.*, {{ class2_name }}.* FROM {{ class2_name }}
        JOIN
            {{ class1_name }} ON {{ class2_name }}.{{ class1_name }}_id = {{ class1_name }}.{{ class1_name }}_id"""

        {% endif %}
    {% endif %}
cursor.execute(query)
table_data = cursor.fetchall()
df = pd.DataFrame(table_data)
# Display the DataFrame with column names
st.dataframe(df)
{% endfor %}
{%- endfor %}

conn.close()
```

Listing 6. Manual implementation: Streamlit visualisations

## H. LLM Interaction: BUML teaching

**User:** If I describe you a DSL for describing data models, will you understand it and preserve this knowledge?

**ChatGPT:** If you describe a Domain-Specific Language (DSL) for describing data models, I can certainly understand the concepts and principles you explain. However, there are a few key points to consider:

1) **Retention of Knowledge**: As an AI, I don't retain information from one interaction to the

next for user privacy and confidentiality reasons. This means I won't "remember" or "preserve" the specific details of the DSL you describe in future interactions. Each session with me is stateless and independent.

2) **Understanding and Interpretation**: I can interpret and understand the concepts, syntax, and semantics of a DSL based on my training, which includes a wide range of programming languages and technical concepts. I can provide insights, explanations, and potentially help with debugging or improving the DSL based on the information you provide.
3) **Limitations**: My understanding is based on the data and programming languages I was trained on, which is extensive but not infinite. If the DSL is highly specialized or not widely used, my ability to provide detailed insights might be limited.

Feel free to describe the DSL, and I'll do my best to understand and discuss it with you!

**User:** Here is firstly a documentation file, revealing the structure of DSL and some examples of usage; *here is the full content of doc.md file*

**ChatGPT:** Thank you for sharing the documentation on BUML Modeling. Based on the information provided, BUML is a Python-based Domain-Specific Language (DSL) that extends the concepts of the Unified Modeling Language (UML). It allows for defining models in an object-oriented manner. The main components of this DSL include:

1) **Property**: Used to define either an attribute of a class or an end of an association. It includes parameters like name, owner, type, multiplicity, visibility, and flags for composition, navigability, and aggregation.
2) **Class**: Represents a class with attributes and an indication of whether it is abstract.
3) **Association**: Defines a relationship between classes, specified by its ends (properties).
4) **Binary Association**: A specialized form of association with exactly two ends.
5) **Association Class**: Combines the properties of a class and an association.
6) **Generalization**: Represents an inheritance relationship between a general (parent) class and a specific (child) class.
7) **Generalization Set**: A group of generalizations that share a parent class, with properties indicating disjointedness and completeness.
8) **Package**: A container for a set of classes, providing a namespace for them.
9) **Constraint**: A rule or condition defined within the context of a class.
10) **DomainModel**: Represents the entire domain model, encompassing types, associations, generalizations, packages, and constraints.

Each element is defined using specific Python classes and attributes, following the object-oriented paradigm. The documentation also includes example Python code snippets for each element, demonstrating how to define and instantiate these elements in practice.
This DSL seems well-structured for modeling complex systems using UML principles within a Python environment, facilitating the integration of model-driven engineering and software development processes.
If you have specific questions or need further clarification on any aspect of this DSL, feel free to ask!

**User:** Here is file `structural.py`, which contains implementations of all abstracts in this DSL; pay attention to the methods each abstract has; *here is the full content of file*.
**ChatGPT:** The `structural.py` file you provided contains the implementation details of the various abstracts in the BUML Modeling DSL. This Python code defines classes that represent different UML concepts, adhering to the object-oriented principles. Here's a summary of the key classes and their methods:

1) **Element and NamedElement**: Basic classes from which others inherit. `NamedElement` adds name and visibility attributes with appropriate getters and setters.
2) **Type, DataType, PrimitiveDataType, EnumerationLiteral, Enumeration**: Hierarchy of classes representing types in the model. `PrimitiveDataType` restricts the type names to a specific set of primitive data types.
3) **TypedElement and Multiplicity**: `TypedElement` is a `NamedElement` with a type. `Multiplicity` represents the min and max multiplicities of a Property.
4) **Property**: Represents a property with attributes like owner, type, multiplicity, and flags for composition, navigability, and aggregation.
5) **Class**: Represents a UML class, handling attributes, associations, generalizations, and inheritance. It provides methods for adding attributes, determining parent and child classes, and managing associations and generalizations.
6) **Association and BinaryAssociation**: `Association` represents relationships between classes. `BinaryAssociation` is a specialized form of `Association` with exactly two ends.
7) **AssociationClass**: Combines the features of an `Association` and a `Class`.
8) **Generalization**: Represents inheritance be-

tween two classes (general and specific).

9) **GeneralizationSet**: A set of `Generalization` relationships with properties for disjoint and complete.
10) **Package**: Represents a collection of classes.
11) **Constraint**: Represents a constraint applied to a class.
12) **DomainModel**: The root element comprising types, associations, generalizations, enumerations, packages, and constraints. It provides methods for retrieving classes and managing the model structure.

Each of these classes is equipped with appropriate Python properties and setter methods to enforce specific constraints and rules, such as ensuring unique names within a set or validating multiplicity bounds. If you have specific questions or need further clarification on any aspect of these implementations, feel free to ask!

**User:** Have you understood how to describe data models with BUML? Can you generate an example of model, described with BUML?

**ChatGPT:**
Certainly! A BUML model is a representation of a data model using a syntax that emulates UML structures in Python. The following example illustrates a conceptual representation that might be used in a BUML-like system. This example is for illustrative purposes and may not correspond to the syntax of an actual BUML library.

```
# Define the basic elements of the BUML modeling system
class BUMLClass:
    def __init__(self, name, properties, associations, generalizations):
        self.name = name
        self.properties = properties  # List of BUMLProperty
        self.associations = associations  # List of BUMLAssociation
        self.generalizations = generalizations  # List of BUMLGeneralization

class BUMLProperty:
    def __init__(self, name, prop_type, multiplicity):
        self.name = name
        self.prop_type = prop_type  # BUMLClass or data type like 'int', 'str'
        self.multiplicity = multiplicity  # Tuple (min, max)

class BUMLAssociation:
    def __init__(self, name, end1, end2):
        self.name = name
        self.end1 = end1  # BUMLProperty
        self.end2 = end2  # BUMLProperty

class BUMLGeneralization:
    def __init__(self, parent, child):
        self.parent = parent  # BUMLClass
        self.child = child  # BUMLClass

# Example usage to create a model
# Define properties
name_property = BUMLProperty("name", "str", (1, 1))
age_property = BUMLProperty("age", "int", (0, 1))

# Define classes
person_class = BUMLClass("Person", [name_property, age_property], [], [])
student_class = BUMLClass("Student", [], [], [BUMLGeneralization(person_class, None)])

# Define associations
enrollment_association = BUMLAssociation(
    "enrollment",
    BUMLProperty("student", student_class, (0, -1)),  # -1 for unlimited multiplicity
    BUMLProperty("course", None, (0, -1))  # Course class not defined in this example
)

# The entire model can now be represented as
buml_model = {
    "classes": [person_class, student_class],
    "associations": [enrollment_association],
    "generalizations": [BUMLGeneralization(person_class, student_class)]
}
```

Listing 7. BUML Model Example

In this example, `BUMLClass` represents a UML Class, `BUMLProperty` a UML Property (an attribute or an association end), `BUMLAssociation` a UML Association, and `BUMLGeneralization` a UML Generalization (inheritance relationship). The multiplicity is expressed as a tuple, with ‘-1‘ denoting an unlimited number of instances (UML’s ‘*‘). This model serves as the basis for generating UML diagrams or for further code generation and analysis tools.

**User:** You wrongly understand the structure of BUML model description, here is an example how it should looks like: [full source code of Library model, from appendix 3].

**ChatGPT:**
Consider a model for a basic blogging system, involving `Users`, `Posts`, and `Comments`. The model is created using BUML (Business UML), demonstrating classes, properties, and associations.

```
from BESSER.src.BUML.metamodel.structural.structural import Association, Class, Multiplicity, Property, PrimitiveDataType, DomainModel

# Properties
user_name = Property(name="name", owner=None, property_type=PrimitiveDataType('str'))
user_email = Property(name="email", owner=None, property_type=PrimitiveDataType('
```

```
    str'))

post_title = Property(name="title", owner=
    None, property_type=PrimitiveDataType('
    str'))
post_content = Property(name="content",
    owner=None, property_type=
    PrimitiveDataType('str'))

comment_text = Property(name="text", owner=
    None, property_type=PrimitiveDataType('
    str'))

# Classes
user_class = Class(name="User", attributes={
    user_name, user_email})
post_class = Class(name="Post", attributes={
    post_title, post_content})
comment_class = Class(name="Comment",
    attributes={comment_text})

# Associations
# User to Post (one-to-many)
user_post = Association(name="user_posts",
    ends={Property(name="author", owner=
    None, property_type=user_class,
    multiplicity=Multiplicity(1, 1)),
    Property(name="posts", owner=None,
    property_type=post_class, multiplicity=
    Multiplicity(0, "*"))})

# Post to Comment (one-to-many)
post_comment = Association(name="
    post_comments", ends={Property(name="
    post", owner=None, property_type=
    post_class, multiplicity=Multiplicity
    (1, 1)), Property(name="comments",
    owner=None, property_type=comment_class
    , multiplicity=Multiplicity(0, "*"))})

# Domain Model
blog_model = DomainModel(name="BlogModel",
    types={user_class, post_class,
    comment_class}, associations={user_post
    , post_comment}, generalizations=None,
    packages=None, constraints=None)
```

Listing 8. BUML Model for Blogging System

**Model Components:**

- **Classes**: `User`, `Post`, and `Comment`
- **Properties**: `User` has `name` and `email`; `Post` has `title` and `content`; `Comment` has `text`.
- **Associations**: Link `User` to `Post` and `Post` to
  textttComment.
- **Domain Model**: Encapsulates the classes and associations.

This is a basic example illustrating the use of BUML for data modeling. It can be extended to include more elements like `Generalization` or `Package` for more complex scenarios.

**User:** You are wrong with associations definition, take a look on the example: [relevant part of `metamodel-doc.md`].

**ChatGPT:**
Thank you for the clarification. Let's revise the example of the blogging system model using BUML with the correct format for associations.
The BUML data model for a blogging system has been revised to correctly format associations using `BinaryAssociation` for one-to-many relationships. The model includes classes `User`, `Post`, and `Comment`, each with defined properties and associations.

```
from BESSER.src.BUML.metamodel.structural.
    structural import Class, Multiplicity,
    Property, BinaryAssociation,
    DomainModel, PrimitiveDataType

# Properties
user_name = Property(name="name", owner=None
    , property_type=PrimitiveDataType('str'
    ))
user_email = Property(name="email", owner=
    None, property_type=PrimitiveDataType('
    str'))

post_title = Property(name="title", owner=
    None, property_type=PrimitiveDataType('
    str'))
post_content = Property(name="content",
    owner=None, property_type=
    PrimitiveDataType('str'))

comment_text = Property(name="text", owner=
    None, property_type=PrimitiveDataType('
    str'))

# Classes
user_class = Class(name="User", attributes={
    user_name, user_email})
post_class = Class(name="Post", attributes={
    post_title, post_content})
comment_class = Class(name="Comment",
    attributes={comment_text})

# Associations (using BinaryAssociation for
    one-to-many relationships)
# User to Post (one-to-many)
user_post_end1 = Property(name="author",
    owner=None, property_type=user_class,
    multiplicity=Multiplicity(1, 1))
user_post_end2 = Property(name="posts",
    owner=None, property_type=post_class,
    multiplicity=Multiplicity(0, "*"))
user_post_association = BinaryAssociation(
    name="user_posts", ends={user_post_end1
    , user_post_end2})

# Post to Comment (one-to-many)
post_comment_end1 = Property(name="post",
    owner=None, property_type=post_class,
    multiplicity=Multiplicity(1, 1))
post_comment_end2 = Property(name="comments"
    , owner=None, property_type=
    comment_class, multiplicity=
    Multiplicity(0, "*"))
post_comment_association = BinaryAssociation
    (name="post_comments", ends={
    post_comment_end1, post_comment_end2})

# Domain Model
blog_model = DomainModel(name="BlogModel",
    types={user_class, post_class,
    comment_class}, associations={
    user_post_association,
    post_comment_association},
    generalizations=None, packages=None,
```

```
    constraints=None)
```

Listing 9. Revised BUML Model for Blogging System

In this revised model:

- Classes `User`, `Post`, and `Comment` are defined with their respective properties.
- Binary associations `user_posts` and `post_comments` represent one-to-many relationships.
- The `DomainModel` encapsulates the classes and their associations.

This model correctly uses `BinaryAssociation` to structure one-to-many relationships, adhering to the provided BUML format.

*I. BUML to SQL converter*

**User:** Can you convert a model described in BUML to other data modelling formats, like SQL or Django models?

**ChatGPT:**
Converting a model described in BUML to other data modeling formats like SQL or Django models is certainly feasible. However, it requires a systematic approach to map the concepts and constructs of BUML to the corresponding constructs in SQL or Django. Here's a general outline of how this can be achieved:

*1. Mapping BUML to SQL:*

- **Classes:** Each BUML class typically maps to an SQL table. The class name becomes the table name, and the class attributes become the columns of the table.
- **Properties/Attributes:** Each BUML property maps to a column in the corresponding SQL table. The data type in BUML needs to be converted to an SQL-compatible data type (e.g., `int`, `varchar`, `date`, etc.).
- **Associations:** Associations in BUML can be mapped to foreign key relationships in SQL. For example, a binary association could translate to a foreign key from one table to another.
- **Generalizations (Inheritance):** SQL doesn't support inheritance natively. This can be handled using separate tables for each class and possibly using foreign keys to link them, or by creating a single table with all fields and additional type information.
- **Multiplicity:** This concept relates to the constraints on the relationships, which can be enforced through foreign keys and additional constraints in SQL.

*2. Mapping BUML to Django Models:*

- **Classes:** Similar to SQL, each BUML class maps to a Django model. The class name becomes the model name.
- **Properties/Attributes:** These map to fields in the Django model. Django ORM supports various field types like `CharField`, `IntegerField`, `DateField`, etc., which can be used based on the BUML property type.
- **Associations:** Associations can be represented using Django's relationship fields like `ForeignKey`, `ManyToManyField`, and `OneToOneField`.
- **Generalizations (Inheritance):** Django ORM supports model inheritance. This can be implemented using either abstract base classes or multi-table inheritance, depending on the BUML model's specifics.
- **Multiplicity:** This is managed through arguments to Django's field types like `blank`, `null`, `unique`, etc., and through the relationship fields' behavior.

*Implementation:*

To perform this conversion programmatically, you would need to write a script or module that takes a BUML model (like the one you've described) as input and then generates the corresponding SQL commands or Django model classes. This script would need to:

1) Parse the BUML model structure.
2) For each class and association, generate the corresponding SQL command or Django model definition.
3) Handle special cases like inheritance, multiplicity, and constraints.

This kind of conversion utility would be quite useful for streamlining the process of moving from conceptual data models in BUML to actual database schemas or ORM definitions. If you need more specific guidance on how to implement such a converter, or have any particular aspects of the conversion process you'd like to explore, feel free to ask!

**User:** Convert this (BUML) model description to SQL format Provide the full code of models: *Library model full description is given.*

**ChatGPT:**
To convert the given BUML model into Django models and SQL format, let's first understand the structure of the model:

*Classes and Properties*

- **Person** with property **name**.
- **Author** inheriting from **Person** with property **nationality**.
- **Book** with properties **title**, **genre**, **price**.
- **Customer** inheriting from **Person** with property **email**.

- **Order** with property **date**.
- **OrderDetails** with property **quantity**.

*Associations*

- Author to Book (one-to-many).
- Customer to Order (one-to-many).
- Order to OrderDetails (one-to-many).
- Book to OrderDetails (one-to-many).

*SQL Schema*

For the SQL schema, we need to create tables for each class with columns for each property and handle associations through foreign keys or additional tables for many-to-many relationships.

```sql
CREATE TABLE person (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL
);

CREATE TABLE author (
    person_id INT PRIMARY KEY,
    nationality VARCHAR(255) NOT NULL,
    FOREIGN KEY (person_id) REFERENCES person(id)
);

CREATE TABLE customer (
    person_id INT PRIMARY KEY,
    email VARCHAR(255) NOT NULL,
    FOREIGN KEY (person_id) REFERENCES person(id)
);

CREATE TABLE book (
    id INT PRIMARY KEY AUTO_INCREMENT,
    title VARCHAR(255) NOT NULL,
    genre VARCHAR(255) NOT NULL,
    price FLOAT NOT NULL
);

CREATE TABLE order (
    id INT PRIMARY KEY AUTO_INCREMENT,
    date DATETIME NOT NULL,
    customer_id INT,
    FOREIGN KEY (customer_id) REFERENCES customer(person_id)
);

CREATE TABLE order_details (
    id INT PRIMARY KEY AUTO_INCREMENT,
    order_id INT,
    book_id INT,
    quantity INT NOT NULL,
    FOREIGN KEY (order_id) REFERENCES order(id),
    FOREIGN KEY (book_id) REFERENCES book(id)
);

CREATE TABLE author_book (
    author_id INT,
    book_id INT,
    PRIMARY KEY (author_id, book_id),
    FOREIGN KEY (author_id) REFERENCES author(person_id),
    FOREIGN KEY (book_id) REFERENCES book(id)
);
```

**User:** If all associations are one-to-many, why do you use Intermediate table for Author-Book relationship?
Correct this mistake.

**ChatGPT:**
You're correct, if all associations are one-to-many, the use of `ManyToManyField` in the Book class was inappropriate. Let's correct that mistake for the SQL schema.

*SQL Schema*

In the SQL schema, we'll update the `Book` table to include a foreign key column referencing the `Author` table.

```sql
CREATE TABLE person (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL
);

CREATE TABLE author (
    person_id INT PRIMARY KEY,
    nationality VARCHAR(255) NOT NULL,
    FOREIGN KEY (person_id) REFERENCES person(id)
);

CREATE TABLE customer (
    person_id INT PRIMARY KEY,
    email VARCHAR(255) NOT NULL,
    FOREIGN KEY (person_id) REFERENCES person(id)
);

CREATE TABLE book (
    id INT PRIMARY KEY AUTO_INCREMENT,
    title VARCHAR(255) NOT NULL,
    genre VARCHAR(255) NOT NULL,
    price FLOAT NOT NULL,
    author_id INT,
    FOREIGN KEY (author_id) REFERENCES author(person_id)
);

CREATE TABLE order (
    id INT PRIMARY KEY AUTO_INCREMENT,
    date DATETIME NOT NULL,
    customer_id INT,
    FOREIGN KEY (customer_id) REFERENCES customer(person_id)
);

CREATE TABLE order_details (
    id INT PRIMARY KEY AUTO_INCREMENT,
    order_id INT,
    book_id INT,
    quantity INT NOT NULL,
    FOREIGN KEY (order_id) REFERENCES order(id),
    FOREIGN KEY (book_id) REFERENCES book(id)
);
```

In this updated schema:

- The `book` table now has a `author_id` column to establish a one-to-many relationship with the `author` table.
- The junction table `author_book` is removed as it's no longer needed.

These corrections should now accurately reflect the one-to-many associations in the SQL schema.

**User:** The task is following: can you create a template on Jinja template language, which takes as an input a model, described in BUML format, and generates SQL code for data model? Template should be model-independent, i.e. for every model described in BUML it should produce a valid SQL query for replicating this data schema. Do not forget about N-ary relationships (N¿2) and generalizations.

**ChatGPT:**
Creating a Jinja template that takes a BUML model as input and generates SQL code for the data model is a complex task, but it's definitely feasible. The template needs to be dynamic enough to handle various elements of the BUML model like classes, properties, associations, generalizations, and n-ary relationships.
Here's a basic outline of the template. This template will iterate over the classes and their properties to create SQL table definitions. It will also handle generalizations (inheritance) and associations.

```
{# Jinja Template for generating SQL from BUML Model #}

{% macro sql_type(buml_type) %}
    {# Maps BUML types to SQL types #}
    {% if buml_type in [’int’, ’float’, ’bool’, ’datetime’, ’date’, ’time’, ’timedelta’] %}
        {{ buml_type | upper }}
    {% elif buml_type == ’str’ %}
        VARCHAR(255)
    {% else %}
        INT  {# Default to INT for foreign keys and unrecognized types #}
    {% endif %}
{% endmacro %}

{% for class in buml_model.classes %}
    {# Create SQL table for each class #}
    CREATE TABLE {{ class.name | lower }} (
        id INT PRIMARY KEY AUTO_INCREMENT,
        {% for attribute in class.attributes %}
            {{ attribute.name | lower }} {{ sql_type(attribute.property_type.name) }},
        {% endfor %}

        {# Handling generalizations (inheritance) #}
        {% if class.generalizations %}
            {% for generalization in class.generalizations %}
                {{ generalization.general.name | lower }}_id INT,
                FOREIGN KEY ({{ generalization.general.name | lower }}_id) REFERENCES {{ generalization.general.name | lower }}(id),
            {% endfor %}
        {% endif %}

        {# Handling associations #}
        {% for association in class.associations %}
            {% if association.ends | length == 2 %}
                {# Binary associations are handled as foreign keys #}
                {% for end in association.ends %}
                    {% if end.type.name != class.name %}
                        {{ end.type.name | lower }}_id INT,
                        FOREIGN KEY ({{ end.type.name | lower }}_id) REFERENCES {{ end.type.name | lower }}(id),
                    {% endif %}
                {% endfor %}
            {% endif %}
        {% endfor %}
    );
{% endfor %}

{# Handling n-ary associations (N>2) #}
{% for association in buml_model.associations %}
    {% if association.ends | length > 2 %}
        CREATE TABLE {{ association.name | lower }} (
            id INT PRIMARY KEY AUTO_INCREMENT,
            {% for end in association.ends %}
                {{ end.type.name | lower }}_id INT,
                FOREIGN KEY ({{ end.type.name | lower }}_id) REFERENCES {{ end.type.name | lower }}(id),
            {% endfor %}
        );
    {% endif %}
{% endfor %}
```

*Explanation:*

1. **sql_type Macro**: Converts BUML types to corresponding SQL data types. 2. **Class Iteration**: For each class, a table is created with columns for each attribute. 3. **Generalizations**: If a class inherits from another class (generalizations in BUML), foreign key references are added. 4. **Associations**: Binary associations (one-to-many or many-to-one) are handled with foreign keys. N-ary associations (N ¿ 2) create separate tables linking all participating classes.

*Usage:*

- To use this template, pass a BUML model object (`buml_model`) with `classes` and `associations` as its properties.
- The model's classes should have `name`, `attributes`, `associations`, and `generalizations` as properties.
- Each attribute should have a `name` and `property_type`.
- Associations should have `name` and `ends`, where `ends` is a list of associated classes.

Please note, this is a basic template and might require further customization based on the specific details of your BUML model and the SQL dialect you are using. Complex scenarios like self-referencing associations, many-to-many relationships without an intermediate class, and advanced SQL features are not covered in this template and would need additional logic.

**User:** This template has a drawback: as you remember, class.associations returns a set of associations, without any order. It could happen, that you will define an association to another class before declaration of corresponding to this class table

**ChatGPT:**
In SQL, the order of table creation matters due to foreign key constraints: a table that is referenced by a foreign key should be created first. To address this, the Jinja template is adjusted to first create tables without foreign keys and then alter these tables later to add the foreign key constraints. This approach ensures that all tables are created before any foreign key constraints are defined, avoiding the issue of referencing a table that has not yet been declared.
Here's the revised template:

```
{# Jinja Template for generating SQL from BUML Model #}

{% macro sql_type(buml_type) %}
    {# Maps BUML types to SQL types #}
    {% if buml_type in ['int', 'float', 'bool', 'datetime', 'date', 'time', 'timedelta'] %}
        {{ buml_type | upper }}
    {% elif buml_type == 'str' %}
        VARCHAR(255)
    {% else %}
        INT  {# Default to INT for foreign keys and unrecognized types #}
    {% endif %}
{% endmacro %}

{# First, create all tables without foreign keys #}
{% for class in buml_model.classes %}
    CREATE TABLE {{ class.name | lower }} (
        id INT PRIMARY KEY AUTO_INCREMENT,
        {% for attribute in class.attributes %}
            {{ attribute.name | lower }} {{ sql_type(attribute.property_type.name) }},
        {% endfor %}
    );
{% endfor %}

{# Then, alter tables to add foreign key constraints for generalizations and binary associations #}
{% for class in buml_model.classes %}
    {# Handling generalizations (inheritance) #}
    {% if class.generalizations %}
        {% for generalization in class.generalizations %}
            ALTER TABLE {{ class.name | lower }}
            ADD COLUMN {{ generalization.general.name | lower }}_id INT,
            ADD FOREIGN KEY ({{ generalization.general.name | lower }}_id) REFERENCES {{ generalization.general.name | lower }}(id);
        {% endfor %}
    {% endif %}

    {# Handling binary associations #}
    {% for association in class.associations %}
        {% if association.ends | length == 2 %}
            {% for end in association.ends %}
                {% if end.type.name != class.name %}
                    ALTER TABLE {{ class.name | lower }}
                    ADD COLUMN {{ end.type.name | lower }}_id INT,
                    ADD FOREIGN KEY ({{ end.type.name | lower }}_id) REFERENCES {{ end.type.name | lower }}(id);
                {% endif %}
            {% endfor %}
        {% endif %}
    {% endfor %}
{% endfor %}

{# Handling n-ary associations (N>2) #}
{% for association in buml_model.associations %}
    {% if association.ends | length > 2 %}
        CREATE TABLE {{ association.name | lower }} (
            id INT PRIMARY KEY AUTO_INCREMENT,
            {% for end in association.ends %}
                {{ end.type.name | lower }}_id INT,
                FOREIGN KEY ({{ end.type.name | lower }}_id) REFERENCES {{ end.type.name | lower }}(id),
            {% endfor %}
        );
    {% endif %}
{% endfor %}
```

*Updated Approach:*

1) **Create Tables First**: All tables are created first without any foreign key columns.
2) **Alter Tables for Generalizations and Associations**: After all tables are created, `ALTER TABLE` statements are used to add foreign key columns and constraints. This is done for both generalizations (inheritance) and binary associations.
3) **N-ary Associations**: Tables for n-ary associations are created separately, as they inherently depend on the existence of other tables.

This approach should resolve the issue of table creation order in SQL, ensuring that all referenced tables exist before foreign key constraints are applied.

**User:** This code seems to be true, can you run it on model I provided to you?

**ChatGPT:**
Currently, executing code directly, including the Jinja template rendering or running SQL queries, is not possible. However, guidance can be provided on how to use the Jinja template with your BUML model to generate SQL code.

*Steps to Use the Jinja Template:*

1) **Install Jinja**: Install Jinja in your Python environment using pip.

```
pip install Jinja
```

2) **Prepare Your BUML Model**: Ensure your BUML model is defined in a way that can be passed to the Jinja template. Your model should be a Python object with classes and associations as properties.
3) **Render the Template**: Use Python to render the Jinja template with your model.

```
from Jinja import Environment,
    FileSystemLoader
# Assuming your template is saved as '
    buml_to_sql.Jinja'
file_loader = FileSystemLoader('path/to
    /your/templates')
env = Environment(loader=file_loader)

template = env.get_template('
    buml_to_sql.Jinja')

# Your BUML model goes here
buml_model = {
    'classes': [person_class,
    authors_class, books_class,
    customers_class, orders_class,
    details_class],
    'associations': [relation_writes,
    relation_places, relation_contains,
     relation_included]
}

output = template.render(buml_model=
    buml_model)
print(output)
```

4) **Execute the SQL Output**: The output from the script will be SQL commands generated from your BUML model. Execute these commands in your SQL database.

This process allows you to generate the SQL schema from your BUML model using the Jinja template. Adjust file paths and object names according to your specific setup.

### J. BUML to Django converter

**User:** Convert this (BUML) model description to Django models. Provide the full code of models: *source code of Library model, presented in 3.*

**ChatGPT:**

To convert the given BUML model into Django models, let's first understand the structure of the model:

*Classes and Properties*

- **Person** with property **name**.
- **Author** inheriting from **Person** with property **nationality**.
- **Book** with properties **title**, **genre**, **price**.
- **Customer** inheriting from **Person** with property **email**.
- **Order** with property **date**.
- **OrderDetails** with property **quantity**.

*Associations*

- Author to Book (one-to-many).
- Customer to Order (one-to-many).
- Order to OrderDetails (one-to-many).
- Book to OrderDetails (one-to-many).

*Django Models*

In Django, each class will be a model, and the properties will be fields in those models. Associations will be represented using ForeignKey and ManyToManyField.

```
from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=255)

class Author(Person):
    nationality = models.CharField(
    max_length=255)

class Customer(Person):
    email = models.EmailField()

class Book(models.Model):
    title = models.CharField(max_length=255)
    genre = models.CharField(max_length=255)
    price = models.FloatField()
    authors = models.ManyToManyField(Author,
     related_name='books')

class Order(models.Model):
    date = models.DateTimeField()
    customer = models.ForeignKey(Customer,
    related_name='orders', on_delete=models
    .CASCADE)

class OrderDetails(models.Model):
    order = models.ForeignKey(Order,
    related_name='order_details', on_delete
    =models.CASCADE)
    book = models.ForeignKey(Book,
    related_name='order_details', on_delete
    =models.CASCADE)
    quantity = models.IntegerField()
```

**User:** If all associations are one-to-many, why do you use ManyToManyField in Book class? Correct this mistake.

**ChatGPT:**

You're correct, if all associations are one-to-many, the use of `ManyToManyField` in the Book class was inappropriate. Let's correct that mistake for the Django models.

```
from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=255)

class Author(Person):
    nationality = models.CharField(max_length=255)

class Customer(Person):
    email = models.EmailField()

class Book(models.Model):
    title = models.CharField(max_length=255)
    genre = models.CharField(max_length=255)
    price = models.FloatField()
    author = models.ForeignKey(Author, related_name='books', on_delete=models.CASCADE)

class Order(models.Model):
    date = models.DateTimeField()
    customer = models.ForeignKey(Customer, related_name='orders', on_delete=models.CASCADE)

class OrderDetails(models.Model):
    order = models.ForeignKey(Order, related_name='order_details', on_delete=models.CASCADE)
    book = models.ForeignKey(Book, related_name='order_details', on_delete=models.CASCADE)
    quantity = models.IntegerField()
```

In this revised version, the `Author` to `Book` relationship is one-to-many, meaning each book can have only one author but an author can write many books. This will be represented with a `ForeignKey` in the `Book` model pointing to the `Author` model.

**User:** The task is following: can you create a template on Jinja template language, which takes as an input a model, described in BUML format, and generates Django models code for data model? Template should be model-independent, i.e. for every model described in BUML it should produce a valid SQL query for replicating this data schema
Do not forget about N-ary relationships (N¿2) and generalizations.
`class.associations` returns a set of associations, without any order. It could happen, that you will define an association to another class before declaration of corresponding to this class table. There exists method `classes_sorted_by_inheritance` in Domain Model object, in that case it will return the list of classes topologically sorted by inheritance.

**ChatGPT:**
Creating a Jinja template to generate Django models from a BUML-described model is a complex but feasible task. The template needs to account for various aspects such as class attributes, inheritance (generalizations), and associations (including binary and n-ary relationships). The use of the `classes_sorted_by_inheritance` method from the Domain object will be crucial to handle inheritance properly.
Here is a Jinja template that should meet your requirements:

```
{# Import Django models #}
from django.db import models

{# Function to convert BUML type to Django field type #}
{% macro to_django_field(property) %}
    {% if property.type.name in ['int', 'float', 'bool'] %}
        return models.{{ property.type.name|capitalize }}Field()
    {% elif property.type.name in ['str'] %}
        return models.CharField(max_length=255)
    {% elif property.type.name in ['datetime'] %}
        return models.DateTimeField()
    {% elif property.type.name in ['date'] %}
        return models.DateField()
    {% elif property.type.name in ['time'] %}
        return models.TimeField()
    {% endif %}
{% endmacro %}

{# Generate Django models #}
{% for class in model.classes_sorted_by_inheritance() %}
class {{ class.name }}({{ 'models.Model' if not class.parents() else class.parents()|map(attribute='name')|join(', ') }}):
    {% for attribute in class.attributes %}
    {{ attribute.name }} = {{ to_django_field(attribute) }}
    {% endfor %}

    {% for association in class.associations %}
        {% for end in association.ends %}
            {% if end.owner.name == class.name and end.type.name != class.name %}
                {% if end.multiplicity.max > 1 %}
                    {{ end.type.name|lower }} = models.ForeignKey('{{ end.type.name }}', related_name='{{ class.name|lower }}_{{ association.name }}', on_delete=models.CASCADE)
                {% else %}
                    {{ end.type.name|lower }}_set = models.ManyToManyField('{{ end.type.name }}', related_name='{{ class.name|lower }}_{{ association.name }}')
                {% endif %}
            {% endif %}
        {% endfor %}
    {% endfor %}
{% endfor %}
```

*Explanation:*

1) **Django Model Fields**: The macro `to_django_field` converts BUML property types to Django field types. It handles common types like integers, strings, dates, and times.
2) **Class Definition**: For each class in the model (sorted by inheritance), a Django model class is defined.
3) **Attributes**: Each attribute of a class is converted into a Django field using the *to_django_field* macro.
4) **Associations**: Associations are handled by adding ForeignKey or ManyToManyField fields based on the multiplicity.
5) **Inheritance**: The template takes care of inheritance by ensuring that derived classes inherit from their parent classes.
6) **N-ary Associations**: Handled by defining appropriate relationships in the Django model.

This template should work for a wide range of BUML models to generate Django models code. However, it might require fine-tuning for specific cases or advanced modeling features.

**User:** Haven't you forgotten to process n-ary relationships, where several classes are involved in one associations?

**ChatGPT:**
The handling of n-ary relationships (where an association involves more than two classes) is complex and was not fully addressed in the previous template. The updated Jinja template now creates a separate Django model for each n-ary association, with each model holding foreign keys to the classes involved in the association.
Here's the updated template:

```
{# Import Django models #}
from django.db import models

{# Function to convert BUML type to Django field type #}
{% macro to_django_field(property) %}
    {% if property.type.name in ['int', 'float', 'bool'] %}
        return models.{{ property.type.name|capitalize }}Field()
    {% elif property.type.name in ['str'] %}
        return models.CharField(max_length=255)
    {% elif property.type.name in ['datetime'] %}
        return models.DateTimeField()
    {% elif property.type.name in ['date'] %}
        return models.DateField()
    {% elif property.type.name in ['time'] %}
        return models.TimeField()
    {% endif %}
{% endmacro %}

{# Generate Django models #}
{% for class in model.classes_sorted_by_inheritance() %}
class {{ class.name }}({{ 'models.Model' if not class.parents() else class.parents()|map(attribute='name')|join(', ') }}):
    {% for attribute in class.attributes %}
    {{ attribute.name }} = {{ to_django_field(attribute) }}
    {% endfor %}

    {% for association in class.associations %}
        {% for end in association.ends %}
            {% if end.owner.name == class.name and end.type.name != class.name %}
                {% if end.multiplicity.max > 1 %}
                    {{ end.type.name|lower }} = models.ForeignKey('{{ end.type.name }}', related_name='{{ class.name|lower }}_{{ association.name }}', on_delete=models.CASCADE)
                {% else %}
                    {{ end.type.name|lower }}_set = models.ManyToManyField('{{ end.type.name }}', related_name='{{ class.name|lower }}_{{ association.name }}')
                {% endif %}
            {% endif %}
        {% endfor %}
    {% endfor %}
{% endfor %}

{# Handle n-ary associations #}
{% for association in model.associations %}
    {% if association.ends|length > 2 %}
class {{ association.name|capitalize }}(models.Model):
        {% for end in association.ends %}
    {{ end.type.name|lower }} = models.ForeignKey('{{ end.type.name }}', on_delete=models.CASCADE)
        {% endfor %}
{% endif %}
{% endfor %}
```

*Explanation of N-ary Associations Handling:*

1) **N-ary Association Model**: A separate Django model is created for each n-ary association.
2) **ForeignKey Fields**: ForeignKey fields are added for each class participating in the n-ary association.
3) **Association Name Capitalization**: The association's name is capitalized to adhere to Django's naming conventions.

This updated template now properly handles n-ary associations by creating separate models for them, supporting a more comprehensive range of BUML models. Adjustments might be needed for specific cases or advanced features.

**User:** Don't forget to check that inside the class you process only BinaryAssociations i.e. only two classes are involved into association.
**ChatGPT:**
To refine the Jinja template so that it processes only BinaryAssociations within each class and handles n-

ary associations separately, the template has been updated. Specifically, when iterating over associations within each class, only associations that involve exactly two classes are handled.
Here’s the updated template:

```
{# Import Django models #}
from django.db import models

{# Function to convert BUML type to Django field type #}
{% macro to_django_field(property) %}
    {% if property.type.name in ['int', 'float', 'bool'] %}
        return models.{{ property.type.name|capitalize }}Field()
    {% elif property.type.name in ['str'] %}
        return models.CharField(max_length=255)
    {% elif property.type.name in ['datetime'] %}
        return models.DateTimeField()
    {% elif property.type.name in ['date'] %}
        return models.DateField()
    {% elif property.type.name in ['time'] %}
        return models.TimeField()
    {% endif %}
{% endmacro %}

{# Generate Django models #}
{% for class in model.classes_sorted_by_inheritance() %}
class {{ class.name }}({{ 'models.Model' if not class.parents() else class.parents()|map(attribute='name')|join(', ') }}):
    {% for attribute in class.attributes %}
    {{ attribute.name }} = {{ to_django_field(attribute) }}
    {% endfor %}

    {# Process only Binary Associations within each class #}
    {% for association in class.associations %}
        {% if association.ends|length == 2 %}
            {% for end in association.ends %}
                {% if end.owner.name == class.name and end.type.name != class.name %}
                    {% if end.multiplicity.max > 1 %}
                        {{ end.type.name|lower }} = models.ForeignKey('{{ end.type.name }}', related_name='{{ class.name|lower }}_{{ association.name }}', on_delete=models.CASCADE)
                    {% else %}
                        {{ end.type.name|lower }}_set = models.ManyToManyField('{{ end.type.name }}', related_name='{{ class.name|lower }}_{{ association.name }}')
                    {% endif %}
                {% endif %}
            {% endfor %}
        {% endif %}
    {% endfor %}
{% endfor %}

{# Handle n-ary associations separately #}
{% for association in model.associations %}
    {% if association.ends|length > 2 %}
class {{ association.name|capitalize }}(models.Model):
        {% for end in association.ends %}
    {{ end.type.name|lower }} = models.ForeignKey('{{ end.type.name }}', on_delete=models.CASCADE)
        {% endfor %}
{% endif %}
{% endfor %}
```

*Changes Made:*

1) **Binary Associations**: Only binary associations are processed within each class definition.
2) **N-ary Associations**: N-ary associations are handled separately, creating a new Django model for each.
3) **Foreign Key and ManyToManyField**: Appropriate field types are used based on the relationship’s multiplicity.

These updates ensure that the template correctly generates Django model code for both binary and n-ary associations defined in a BUML model, adhering to the correct relational patterns in Django.

**User:** It could happen, that you will define an association to another class before declaration of corresponding to this class table, as the classes were sorted by inheritance, not by associations.
**ChatGPT:**
Addressing the issue of potentially referencing a class in an association before its declaration, the Jinja template has been adjusted to use string-based model names in foreign key and many-to-many relationships. This change allows Django to resolve model names at runtime, thus handling forward references effectively.
Here’s the adjusted template:

```
{# Import Django models #}
from django.db import models

{# Function to convert BUML type to Django field type #}
{% macro to_django_field(property) %}
    {% if property.type.name in ['int', 'float', 'bool'] %}
        return models.{{ property.type.name|capitalize }}Field()
    {% elif property.type.name in ['str'] %}
        return models.CharField(max_length=255)
    {% elif property.type.name in ['datetime'] %}
        return models.DateTimeField()
    {% elif property.type.name in ['date'] %}
        return models.DateField()
    {% elif property.type.name in ['time'] %}
        return models.TimeField()
    {% endif %}
{% endmacro %}

{# Generate Django models #}
{% for class in model.classes_sorted_by_inheritance() %}
```

```
class {{ class.name }}({{ 'models.Model' if not class.parents() else class.parents()|map(attribute='name')|join(', ') }}):
    {% for attribute in class.attributes %}
    {{ attribute.name }} = {{ to_django_field(attribute) }}
    {% endfor %}

    {# Process only Binary Associations within each class #}
    {% for association in class.associations %}
        {% if association.ends|length == 2 %}
            {% for end in association.ends %}
                {% if end.owner.name == class.name and end.type.name != class.name %}
                    {% if end.multiplicity.max > 1 %}
                        {{ end.type.name|lower }} = models.ForeignKey('{{ end.type.name }}', related_name='{{ class.name|lower }}_{{ association.name }}', on_delete=models.CASCADE)
                    {% else %}
                        {{ end.type.name|lower }}_set = models.ManyToManyField('{{ end.type.name }}', related_name='{{ class.name|lower }}_{{ association.name }}')
                    {% endif %}
                {% endif %}
            {% endfor %}
        {% endif %}
    {% endfor %}
{% endfor %}

{# Handle n-ary associations separately #}
{% for association in model.associations %}
    {% if association.ends|length > 2 %}
class {{ association.name|capitalize }}(models.Model):
        {% for end in association.ends %}
    {{ end.type.name|lower }} = models.ForeignKey('{{ end.type.name }}', on_delete=models.CASCADE)
        {% endfor %}
{% endif %}
{% endfor %}
```

*Key Changes:*

1) **String-based Model Names**: Relationships now use the associated class name as a string.
2) **Forward References**: By using string literals for model names, forward references to models are effectively handled.

This template generates Django models that correctly handle inheritance and associations, including forward references to other models, useful in complex models where the order of class definitions can vary.

**User:** `Class` object does not have `parents()` method.

**ChatGPT:**

Given that the `parents()` method is not available in the BUML model, the template has been adjusted to use the `generalizations` attribute for determining parent classes. This modification allows the template to correctly handle inheritance relationships.

Here's the revised template:

```
...

{# Generate Django models #}
{% for class in model.classes_sorted_by_inheritance() %}
class {{ class.name }}({{ 'models.Model' if not class.generalizations else ', '.join([gen.general.name for gen in class.generalizations if gen.specific.name == class.name]) }}):
    {% for attribute in class.attributes %}
    {{ attribute.name }} = {{ to_django_field(attribute) }}
    {% endfor %}
    ...
{% endfor %}
```

*Explanation of the Changes:*

1) **Handling Inheritance**: The inheritance relationship is determined by iterating through the `generalizations` of each class.

This revised template now correctly handles inheritance and generates valid Django model code for the given BUML model.

### K. Data visualisation with Streamlit

**User:** Can you create a Streamlit web application with visualisation insights about this data model [library model presented in V-B], given in BUML? For example, if you see an association between two tables, can you provide some stats about the join of this tables and draw a suitable by type plot? Or any other useful visualisation insights basing on described model? Make also a view of all tables available, as well as the possible joins view combinations

Suppose that model for this data is inside PostgreSQL database, which was generated with the queries you generated.

Provide the full code for application, without any exclusions; that is, it can be copy-pasted to python file and run.

**ChatGPT:**

Creating a full Streamlit application to visualize insights from a BUML data model and connect to a PostgreSQL database involves several components, including database connection, data retrieval, visualization, and UI design. Here's a simplified version of the code for a Streamlit app:

```
import streamlit as st
import pandas as pd
```

```
import psycopg2
import matplotlib.pyplot as plt
import seaborn as sns

# Database Connection
def connect_to_db():
    try:
        conn = psycopg2.connect(
            host="your_host",
            database="your_database",
            user="your_username",
            password="your_password")
        return conn
    except Exception as e:
        st.error(f"Error connecting to database: {e}")
        return None

# Query Function
def run_query(conn, query):
    try:
        return pd.read_sql(query, conn)
    except Exception as e:
        st.error(f"Error executing query: {e}")
        return None

# Streamlit UI
def main():
    st.title("BUML Model Data Visualization")

    conn = connect_to_db()
    if not conn:
        st.stop()

    st.sidebar.title("Navigation")
    choices = ["Tables View", "Join Combinations", "Visualizations"]
    choice = st.sidebar.selectbox("Choose View", choices)

    if choice == "Tables View":
        st.subheader("Available Tables")
        tables = ["person", "author", "book", "customer", "order", "order_details"]
        table_choice = st.selectbox("Choose a table", tables)
        query = f"SELECT * FROM {table_choice} LIMIT 10"
        df = run_query(conn, query)
        if df is not None:
            st.dataframe(df)

    elif choice == "Join Combinations":
        st.subheader("Join Table Combinations")
        st.markdown("### Author and Book Join")
        query = "SELECT * FROM author JOIN book ON author.person_id = book.author_id LIMIT 10"
        df = run_query(conn, query)
        if df is not None:
            st.dataframe(df)

    elif choice == "Visualizations":
        st.subheader("Data Visualizations")
        st.markdown("### Price Distribution in Books")
        query = "SELECT price FROM book"
        df = run_query(conn, query)
        if df is not None:
            fig, ax = plt.subplots()
            sns.histplot(df['price'], kde=True, ax=ax)
            st.pyplot(fig)

if __name__ == "__main__":
    main()
```

*Notes:*

- **Database Connection**: Replace database credentials with actual values.
- **Query Functions**: Modify SQL queries based on the database schema.
- **Visualizations**: Include histograms and other plots as needed.
- **Tables and Joins**: Extend the views to cover all tables and relevant join combinations.
- **Performance**: Limit query results for better performance.

This code serves as a starting point for a more comprehensive application. Run it by saving the code in a Python file and using Streamlit to execute it.

**User:** Which other data visualisations would you recommend to implement? Provide a code for each of them (at least 3-4 different plots), and integrate it in Streamlit application above.

**ChatGPT:**
For a data model involving books, authors, customers, orders, and order details, various insightful visualizations can be implemented. Below are code snippets for these visualizations, which can be integrated into the Streamlit application.

*1. Number of Books per Author*

This visualization shows the number of books written by each author.

```
def plot_books_per_author(conn):
    query = """
    SELECT author.person_id, COUNT(book.id) AS book_count
    FROM author
    JOIN book ON author.person_id = book.author_id
    GROUP BY author.person_id
    """
    df = run_query(conn, query)
    if df is not None and not df.empty:
        fig, ax = plt.subplots()
        sns.barplot(x='person_id', y='book_count', data=df, ax=ax)
        ax.set_title('Number of Books per Author')
        ax.set_xlabel('Author ID')
        ax.set_ylabel('Number of Books')
        st.pyplot(fig)
```

*2. Average Book Price by Genre*

This plot shows the average book price by genre.

```
def plot_average_price_by_genre(conn):
    query = """
    SELECT genre, AVG(price) AS avg_price
```

```
    FROM book
    GROUP BY genre
    """
    df = run_query(conn, query)
    if df is not None and not df.empty:
        fig, ax = plt.subplots()
        sns.barplot(x='genre', y='avg_price', data=df, ax=ax)
        ax.set_title('Average Book Price by Genre')
        ax.set_xlabel('Genre')
        ax.set_ylabel('Average Price')
        st.pyplot(fig)
```

*3. Number of Orders per Customer*

This visualization shows how many orders each customer has placed.

```
def plot_orders_per_customer(conn):
    query = """
    SELECT customer.person_id, COUNT(order.id) AS order_count
    FROM customer
    JOIN order ON customer.person_id = order.customer_id
    GROUP BY customer.person_id
    """
    df = run_query(conn, query)
    if df is not None and not df.empty:
        fig, ax = plt.subplots()
        sns.barplot(x='person_id', y='order_count', data=df, ax=ax)
        ax.set_title('Number of Orders per Customer')
        ax.set_xlabel('Customer ID')
        ax.set_ylabel('Number of Orders')
        st.pyplot(fig)
```

*4. Sales Over Time*

This plot provides insights into sales trends over time.

```
def plot_sales_over_time(conn):
    query = """
    SELECT DATE(order.date) AS order_date, SUM(order_details.quantity) AS books_sold
    FROM order
    JOIN order_details ON order.id = order_details.order_id
    GROUP BY order_date
    ORDER BY order_date
    """
    df = run_query(conn, query)
    if df is not None and not df.empty:
        fig, ax = plt.subplots()
        sns.lineplot(x='order_date', y='books_sold', data=df, ax=ax)
        ax.set_title('Sales Over Time')
        ax.set_xlabel('Date')
        ax.set_ylabel('Books Sold')
        st.pyplot(fig)
```

*Integrating Visualizations into Streamlit Application*

Update the ‘Visualizations‘ section within the ‘main()‘ function of the Streamlit application:

```
elif choice == "Visualizations":
    st.subheader("Data Visualizations")
    vis_choices = ["Books per Author", "Average Price by Genre", "Orders per Customer", "Sales Over Time"]
    vis_choice = st.selectbox("Choose a visualization", vis_choices)

    if vis_choice == "Books per Author":
        plot_books_per_author(conn)
    elif vis_choice == "Average Price by Genre":
        plot_average_price_by_genre(conn)
    elif vis_choice == "Orders per Customer":
        plot_orders_per_customer(conn)
    elif vis_choice == "Sales Over Time":
        plot_sales_over_time(conn)
```

These snippets add a dropdown menu in the Streamlit UI for users to select and view different visualizations. Handling cases where the query returns empty or null data is important, as well as ensuring the database has sufficient data for meaningful plots.

**User:** Provide full code with a choice of visualisations, with all possible joins etc.

**ChatGPT:**
Below is the complete Streamlit application code that visualizes insights from a BUML data model and connects to a PostgreSQL database. This code provides various functionalities including viewing different tables, join combinations, and data visualizations.

```
import streamlit as st
import pandas as pd
import psycopg2
import matplotlib.pyplot as plt
import seaborn as sns

# Database Connection
def connect_to_db():
    try:
        conn = psycopg2.connect(
            host="your_host",
            database="your_database",
            user="your_username",
            password="your_password")
        return conn
    except Exception as e:
        st.error(f"Error connecting to database: {e}")
        return None

# Query Function
def run_query(conn, query):
    try:
        return pd.read_sql(query, conn)
    except Exception as e:
        st.error(f"Error executing query: {e}")
        return None

# Visualization Functions
def plot_books_per_author(conn):
    query = """
    SELECT author.person_id, COUNT(book.id) AS book_count
    FROM author
    JOIN book ON author.person_id = book.author_id
```

```python
    GROUP BY author.person_id
    """
    df = run_query(conn, query)
    if df is not None and not df.empty:
        fig, ax = plt.subplots()
        sns.barplot(x='person_id', y='book_count', data=df, ax=ax)
        ax.set_title('Number of Books per Author')
        ax.set_xlabel('Author ID')
        ax.set_ylabel('Number of Books')
        st.pyplot(fig)

def plot_average_price_by_genre(conn):
    query = """
    SELECT genre, AVG(price) AS avg_price
    FROM book
    GROUP BY genre
    """
    df = run_query(conn, query)
    if df is not None and not df.empty:
        fig, ax = plt.subplots()
        sns.barplot(x='genre', y='avg_price', data=df, ax=ax)
        ax.set_title('Average Book Price by Genre')
        ax.set_xlabel('Genre')
        ax.set_ylabel('Average Price')
        st.pyplot(fig)

def plot_orders_per_customer(conn):
    query = """
    SELECT customer.person_id, COUNT(order.id) AS order_count
    FROM customer
    JOIN order ON customer.person_id = order.customer_id
    GROUP BY customer.person_id
    """
    df = run_query(conn, query)
    if df is not None and not df.empty:
        fig, ax = plt.subplots()
        sns.barplot(x='person_id', y='order_count', data=df, ax=ax)
        ax.set_title('Number of Orders per Customer')
        ax.set_xlabel('Customer ID')
        ax.set_ylabel('Number of Orders')
        st.pyplot(fig)

def plot_sales_over_time(conn):
    query = """
    SELECT DATE(order.date) AS order_date, SUM(order_details.quantity) AS books_sold
    FROM order
    JOIN order_details ON order.id = order_details.order_id
    GROUP BY order_date
    ORDER BY order_date
    """
    df = run_query(conn, query)
    if df is not None and not df.empty:
        fig, ax = plt.subplots()
        sns.lineplot(x='order_date', y='books_sold', data=df, ax=ax)
        ax.set_title('Sales Over Time')
        ax.set_xlabel('Date')
        ax.set_ylabel('Books Sold')
        st.pyplot(fig)

# Streamlit UI
def main():
    st.title("BUML Model Data Visualization")

    conn = connect_to_db()
    if not conn:
        st.stop()

    st.sidebar.title("Navigation")
    choices = ["Tables View", "Join Combinations", "Visualizations"]
    choice = st.sidebar.selectbox("Choose View", choices)

    if choice == "Tables View":
        st.subheader("Available Tables")
        tables = ["person", "author", "book", "customer", "order", "order_details"]
        table_choice = st.selectbox("Choose a table", tables)
        query = f"SELECT * FROM {table_choice} LIMIT 10"
        df = run_query(conn, query)
        if df is not None:
            st.dataframe(df)

    elif choice == "Join Combinations":
        st.subheader("Join Table Combinations")
        join_choices = ["Author-Book", "Customer-Order", "Order-OrderDetails"]
        join_choice = st.selectbox("Choose a join", join_choices)

        if join_choice == "Author-Book":
            query = "SELECT * FROM author JOIN book ON author.person_id = book.author_id LIMIT 10"
        elif join_choice == "Customer-Order":
            query = "SELECT * FROM customer JOIN order ON customer.person_id = order.customer_id LIMIT 10"
        elif join_choice == "Order-OrderDetails":
            query = "SELECT * FROM order JOIN order_details ON order.id = order_details.order_id LIMIT 10"

        df = run_query(conn, query)
        if df is not None:
            st.dataframe(df)

    elif choice == "Visualizations":
        st.subheader("Data Visualizations")
        vis_choices = ["Books per Author", "Average Price by Genre", "Orders per Customer", "Sales Over Time"]
        vis_choice = st.selectbox("Choose a visualization", vis_choices)

        if vis_choice == "Books per Author":
            plot_books_per_author(conn)
        elif vis_choice == "Average Price by Genre":
            plot_average_price_by_genre(conn)
        elif vis_choice == "Orders per Customer":
            plot_orders_per_customer(conn)
        elif vis_choice == "Sales Over Time":
            plot_sales_over_time(conn)

if __name__ == "__main__":
    main()
```

*Running the Application*

To run the application, save this code into a Python file (e.g., ‘app.py‘) and execute it using Streamlit:

```
streamlit run app.py
```

Replace database connection parameters with your own, and ensure your database schema matches the queries used in the code. Adjust the queries or add error handling as necessary based on your database and data availability.