# Model Driven Software Engineering for Data Warehousing — Part 1: Code Generation

Everything should be made as simple as possible, but not simpler.

— Attributed to Albert Einstein

This article is the first part of a series of articles in which I want to give an overview of how I think Model Driven Software Engineering (MDSE) can be used for data warehousing. This article is about the concepts of model driven software engineering and why you should use it while developing your Data Warehouse (DWH). Finally, the topics described in this article are explained using a Python code example.

## What is Model Driven (Software) Engineering?

According to Wikipedia (<https://en.wikipedia.org/wiki/Model-driven_engineering>, 18 November 2022):

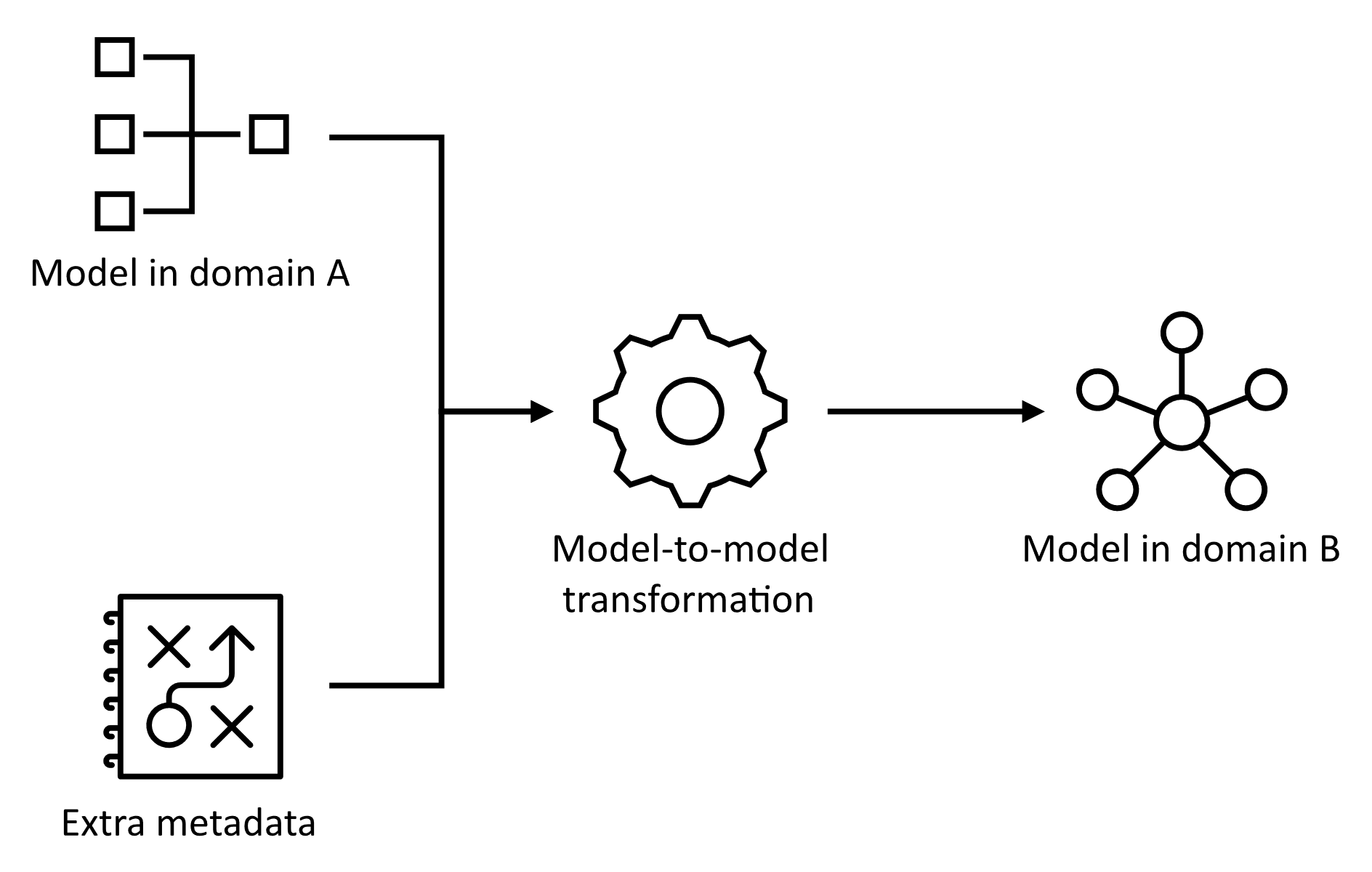
**Model-driven engineering** (**MDE**) is a software development methodology that focuses on creating and exploiting domain models, which are conceptual models of all the topics related to a specific problem. Hence, it highlights and aims at abstract representations of the knowledge and activities that govern a particular application domain, rather than the computing (i.e., algorithmic) concepts.

I have used Model Driven Software Engineering throughout my professional career and in the last 8 years in combination with Datawarehouse automation. In my view, model driven software engineering can be expressed in terms of (meta-)models, model-to-model transformations and code generation.

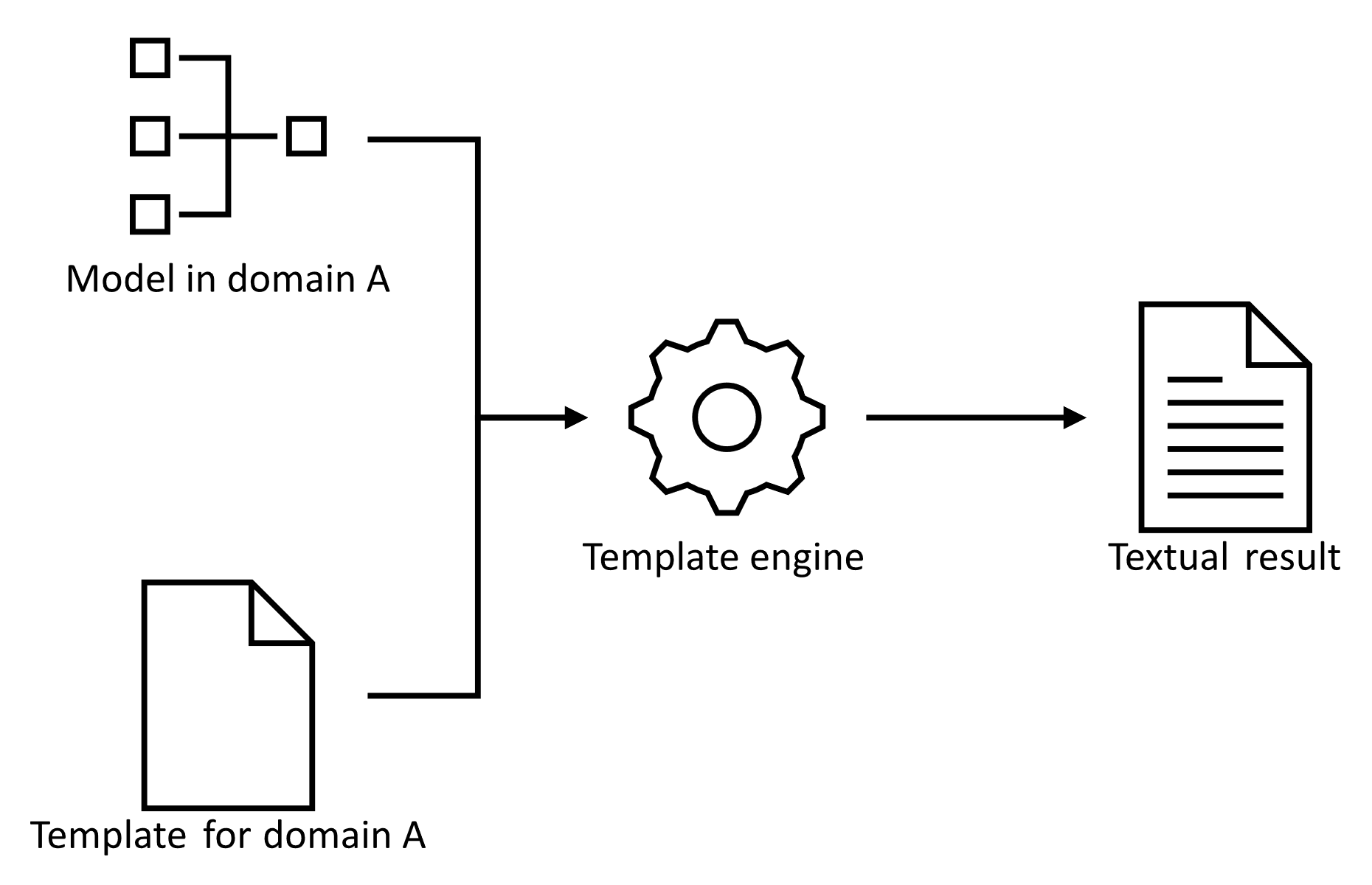
*Models* describe objects in a (problem) domain. Objects in the model should capture all important concepts of the domain. To make sure models do not become too complex, care must be taken not to overdo it: models should not contain too many details. When I apply MDSE, I use what is generally called *meta-model* (model describing models) to describe what objects can be used in models. The meta-model must be complete in the sense that it captures all important concepts of the domain, but not to extensive as this tends to complicate matters.

The concepts of model and meta-model translate directly to databases. One could say that the table structures are meta-models, they define what can be stored, for example, an email address is a string of at most 128 characters. Primary key constraints and foreign key constraints are also part of the meta-model as they limit the data that can be stored in a table. The data in the tables relates to models, it contains actual data such as the email address example@blog.com.

What you often see in Enterprise Architecture tools, is that people develop logical data models. Eventually, these logical models must be implemented in a database, but to do this a number of extra steps are required to transform objects in the logical level to objects in the physical level. These tools support automation of this process. This is what is generally called *model to model transformations* or *M2M*. For example, entities in the logical domain are transformed into tables in the physical domain, attributes are transformed into columns and relations are transformed into foreign key constraints. Often, the model in the source domain of the transformation lacks information to fully transform into the target domain. An example of this is that the type of information of logical attributes is missing or not explicit enough to directly translate into column data types in the physical domain. The image below shows the conceptual picture of model-to-model transformations.



Models are nice, but to use them it is often required to convert them into some textual format, such as code or documentation. To do this, I usually use templates. What this text is depends on my requirements. It can be executable code, such as ETL statements to load data into tables, or descriptive like documentation about data lineage, or anything else that can be represented in a textual format. A template engine is used to apply the template to the model and generating the textual result. What a template engine basically does is that it tries to replace parts of the provided template with parts of the provided model resulting in one or more textual outputs. This approach is referred to as code generation. The last section of this article contains an example that makes this more concrete. It is good practice to separate functionality and form: the model contains the concepts and templates define the output syntax, to keep templates simple it should purely focus on generating text. The image below shows the conceptual picture of model-to-text transformations.



## When to use Model Driven Software Engineering

In my experience, the practical benefits of this methodology are threefold.

1. Reduce repetitive work

I believe that it is important to work on ‘fun’ things. Doing the same thing over and over again does not qualify as being fun (at least for me). When developing a data warehouse, many tasks are repetitive. Maybe the most common example is creating a staging layer in which data from source systems is stored prior to be stored in the actual DWH. Often the structure of the source tables is copied and several common columns to store processing data are added. This activity has to be done for every relevant table in every source system. I already get bored writing about it!

1. Reduce the risk of errors

When doing manual work there is a high risk of introducing errors. Typos for instance. Often, finding and fixing these errors requires a lot of work (the later in the development process an error is found, the more it costs to fix it). When applying MDSE, all text that is generated is based on the same models and templates. This means that is enough to validate the correctness of your models and your templates to make sure that the result of the code generation process will be correct. Especially when applied at large scale this will save you a lot of headaches.

1. Increase efficiency

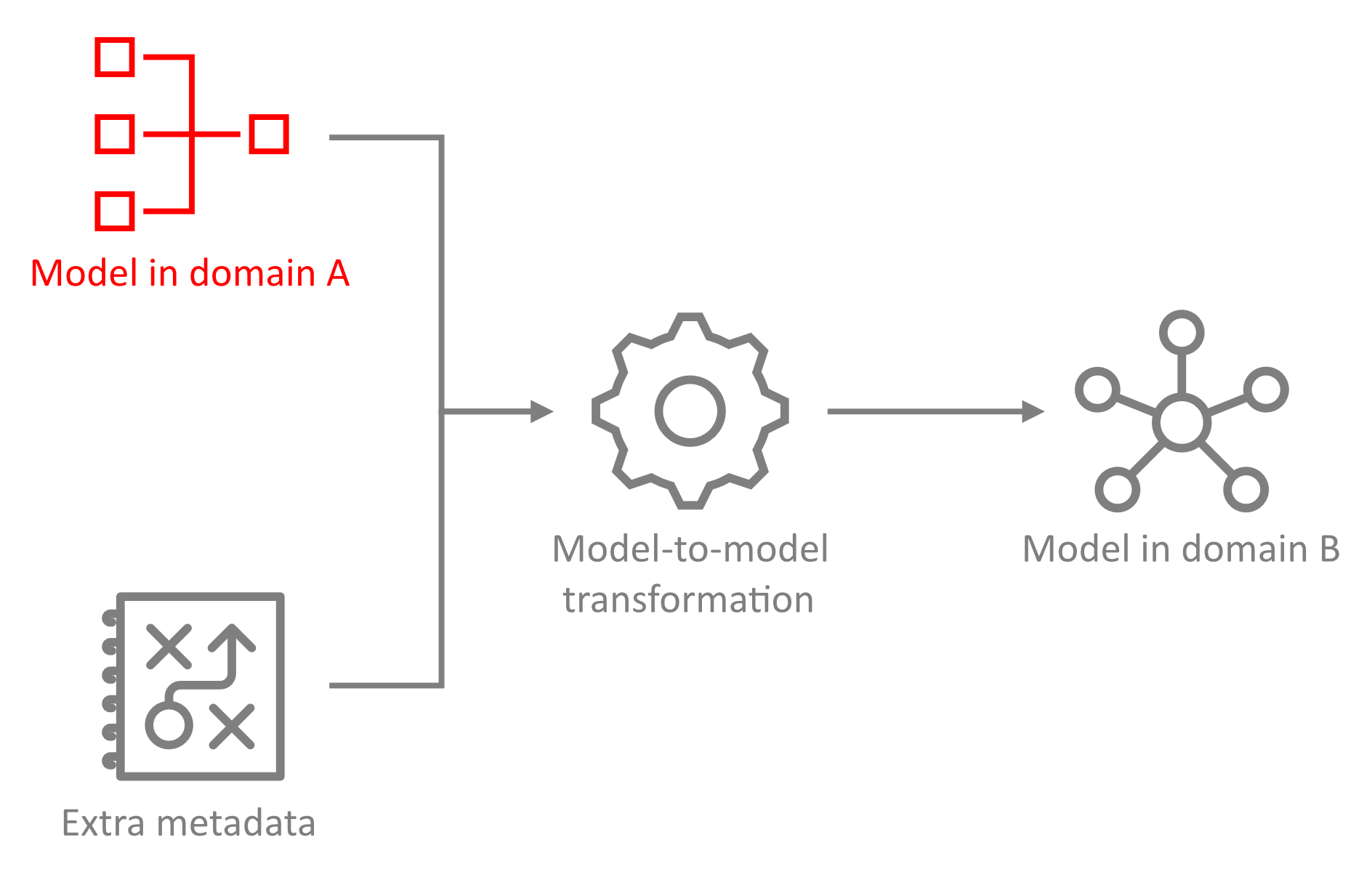
One could argue that manual work is s-l-o-w. According to the Guinness World Records, since 2005 the official holder of the title of fastest typist in the world on a computer is Barbara Blackburn with a top typing speed of 212 words per minute. Although this is way, way faster than I ever type, it is far from the performance that you get when generating your code. When applying MDSE at large scale, the time it takes to develop and maintain your models and templates does not compare to the time it would take to write your code manually. This means that developers have more time to work on fun things (or to drink coffee!).

The points in the list above indicate that it becomes profitable to apply MDSE when you have workloads that are highly repetitive. As the setup for MDSE i.e., defining your (meta-)models, transformations and templates takes time, doing it only for a small number of applications is not efficient.

## Code Generation Example

This section explains the concepts described in the previous section with a Python code example. In this example, I will show how to convert entities in logical model to tables in a physical model (RDBMS) using a model-to-model transformation. The tables in the physical model are then transformed into actual TSQL DML statements. The code in this section can be found on Github: <https://github.com/Maarten-gh/mdsedwhblog/>.

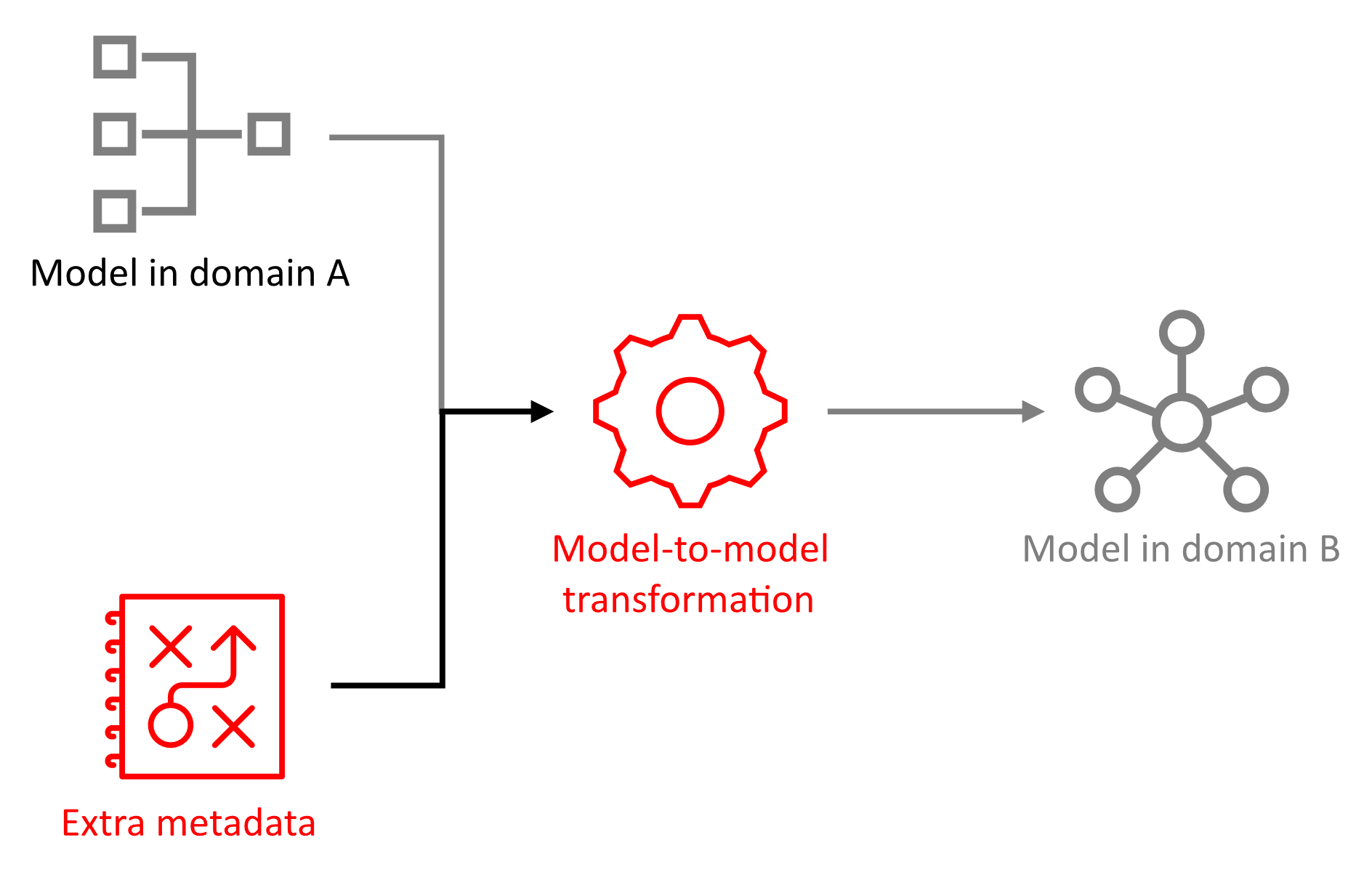
### The Logical Domain



The first step is to create the model that we need. The model contains the order domain, that in turn contains three entities, Customer, Product and Order. Every entity has *properties* that define the state of an entity and *relations* to other entities. The classes LogicalModel, Domain, Entity, Property and Relation are custom classes that I created for readability of this code. These classes form the meta-model of the logical domain.

1. orders\_domain\_model **=** LogicalModel(
2. domains**=**[
3. Domain(
4. name**=**"orders",
5. entities**=**[
6. Entity(
7. name**=**"Customer",
8. properties**=**[
9. Property(
10. name**=**"address",
11. datatype**=**"string",
12. ),
13. ],
14. ),
15. Entity(
16. name**=**"Product",
17. properties**=**[
18. Property(
19. name**=**"description",
20. datatype**=**"string",
21. ),
22. ],
23. ),
24. Entity(
25. name**=**"Order",
26. properties**=**[
27. Property(
28. name**=**"orderTimestamp\_utc",
29. datatype**=**"timestamp",
30. ),
31. Property(
32. name**=**"amount",
33. datatype**=**"int",
34. ),
35. ],
36. relations**=**[
37. Relation(
38. role**=**"orderedBy",
39. domain\_name**=**"orders",
40. entity\_name**=**"Customer",
41. ),
42. Relation(
43. role**=**"orderFor",
44. domain\_name**=**"orders",
45. entity\_name**=**"Product",
46. ),
47. ],
48. ),
49. ],
50. ),
51. ],
52. )

### Transforming from the Logical Domain to the Physical Domain



Now, assume that we want to create a data model in a database to store data for this domain. To simplify this, we can transform the model above into a model that is fit to generate DML code from. This means that we need to:

* convert domains to schemas,
* convert entities to tables,
* convert properties to columns with correct data types,
* add primary key constraints by introducing an extra id column to each table, and
* add foreign key constraints for each relation

We can define a model-to-model transformation (M2M) that implements this conversion. The Python function transform\_domain\_model\_to\_database\_model(domain\_model) below implements this transformation. The classes PhysicalModel, Schema, Table, Column and ForeignKeyConstraint are convenience classes. These classes from the meta-model of the physical domain. The functions shown below define transformations from logical objects to physical objects using their meta-model definitions.

1. **def** logical\_model\_to\_physical\_model(logical\_model: LogicalModel) **-**> PhysicalModel:
2. **return** PhysicalModel(
3. schemas**=**[domain\_to\_schema(d) **for** d **in** logical\_model.domains]
4. )

7. **def** domain\_to\_schema(domain: Domain) **-**> Schema:
8. **return** Schema(
9. name**=**domain.name,
10. tables**=**[entity\_to\_table(e) **for** e **in** domain.entities],
11. )

14. **def** entity\_to\_table(entity: Entity) **-**> Table:
15. id\_column **=** Column(
16. name**=**"id",
17. datatype**=**"uniqueidentifier",
18. nullable**=**False,
19. )
21. primary\_key\_constraint **=** PrimaryKeyConstraint(
22. name**=**f"pk\_{entity.name}",
23. column\_names**=**["id"],
24. )
26. **return** Table(
27. name**=**entity.name,
28. columns**=**[id\_column] **+** [property\_to\_column(p) **for** p **in** entity.properties] **+** [
29. relation\_to\_column(r) **for** r **in** entity.relations],
30. primary\_key\_constraint**=**primary\_key\_constraint,
31. foreign\_key\_constraints**=**[
32. relation\_to\_foreign\_key\_constraint(r) **for** r **in** entity.relations],
33. )

36. **def** property\_to\_column(property: Property) **-**> Column:
37. **return** Column(
38. name**=**property.name,
39. nullable**=**True,
40. **\*\***property\_datatype\_to\_database\_datatype(property),
41. )

44. **def** relation\_to\_column(relation: Relation) **-**> Column:
45. **return** Column(
46. name**=**f"{relation.role}\_{relation.entity\_name}\_id",
47. datatype**=**"uniqueidentifier",
48. nullable**=**False,
49. )

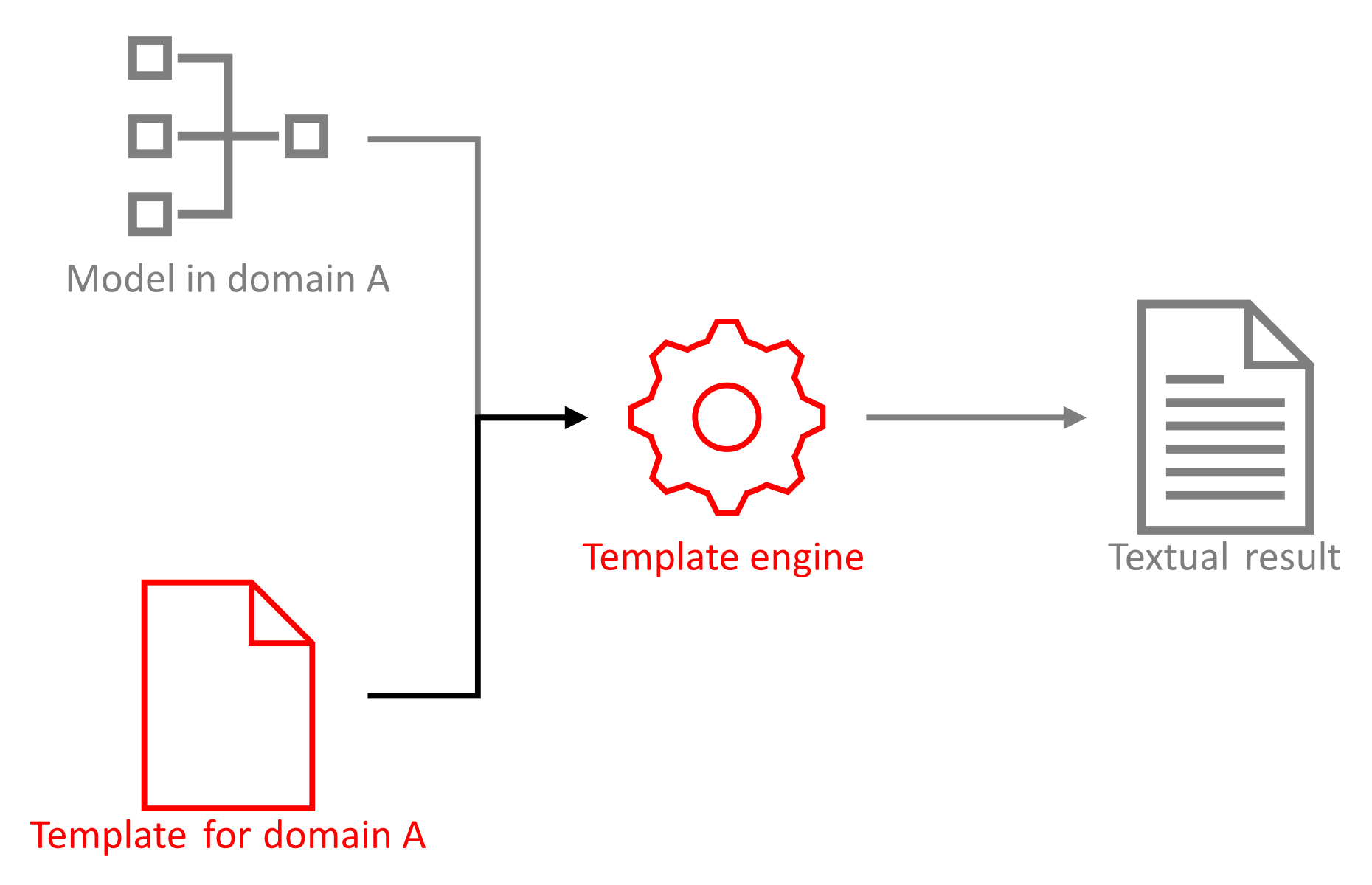
52. **def** relation\_to\_foreign\_key\_constraint(relation: Relation) **-**> ForeignKeyConstraint:
53. **return** ForeignKeyConstraint(
54. name**=**f"fk\_{relation.role}\_{relation.entity\_name}",
55. column\_names**=**[f"{relation.role}\_{relation.entity\_name}\_id",],
56. foreign\_schema\_name**=**relation.domain\_name,
57. foreign\_table\_name**=**relation.entity\_name,
58. foreign\_column\_names**=**["id"],
59. )

62. **def** property\_datatype\_to\_column\_datatype(property: Property) **-**> dict[str, any]:
63. """
64. Returns a column datatype, length and scale based on the data type of the
65. provided property. If the type cannot be resolved, the datatype of the
66. property is used and length and scale are both `None`.
67. """
68. result **=** dict(
69. datatype**=**property.datatype,
70. length**=**None,
71. scale**=**None,
72. )
74. **if** "string" **==** property.datatype:
75. result["datatype"] **=** 'nvarchar'
76. result["length"] **=** 255
77. **elif** "timestamp" **==** property.datatype:
78. result["datatype"] **=** 'datetime2'
80. **return** result

The transformation consists of several functions that convert different components of the logical data model into corresponding components of the physical data model. Here is a brief description of each function:

* logical\_model\_to\_physical\_model(logical\_model: LogicalModel) -> PhysicalModel:   
  This function takes a LogicalModel object as input and returns a PhysicalModel object. It does so by iterating through each domain in the logical model, calling the domain\_to\_schema() function on each domain, and collecting the resulting schemas in a list. The list of schemas is then used to create the PhysicalModel object.
* domain\_to\_schema(domain: Domain) -> Schema:  
  This function takes a Domain object as input and returns a Schema object. It does so by creating a Schema object with the name of the domain, and by iterating through each entity in the domain, calling the entity\_to\_table() function, adding the resulting list of tables to the schema.
* entity\_to\_table(entity: Entity) -> Table:  
  This function takes an Entity object as input and returns a Table object. It creates an id Column object and a PrimaryKeyConstraint object. It iterates over the properties of the entity to create Column objects by calling the property\_to\_column function on each property. Moreover, it creates a Column object and a ForeignKeyConstraint object for each relation of the entity by calling the functions relation\_to\_column and relation\_to\_foreign\_key\_constraint respectively.
* property\_to\_column(property: Property) -> Column:  
  This function takes a Property object and transforms it to a Column object. It uses the name of the property as the name of the column. The created column will be nullable, and type information is added by calling the property\_datatype\_to\_column\_datatype function.
* relation\_to\_column(relation: Relation) -> Column:  
  This function takes a Relation object and returns a Column object. It does so by creating a column name based on the entity and role of the relation. The datatype of the created column is uniqueidentifier and the column is not nullable.
* relation\_to\_foreign\_key\_constraint(relation: Relation) -> ForeignKeyConstraint:  
  This function takes a Relation object as input and returns a ForeignKeyConstraint object. The name of the foreign key is based on the entity and the role of the relation. The column names of the foreign key object are the same as the name of the column created by calling the relation\_to\_column function on the provided Relation object. The foreign domain, entity and columns of the foreign key object refer to the corresponding schema, domain and id column respectively.
* property\_datatype\_to\_column\_datatype(property: Property) **-**> dict[str, any]:  
  This function takes a Property object as input and returns a dictionary with the members datatype, length and scale. By default, the datatype is set to the datatype of the input Property object and length and scale are None. If the datatype of the Property object is string or timestamp the values for datatype and length are changed. Note that this function adds extra datatype information. For example, the length of the nvarchar columns is fixed to 255.

### Code Generation



Now the database model is created, we can easily generate DDL code. The code below shows a simple Jinja2 template that is used to generate the code. There are a number of important things to notice (more details can be found here: <https://jinja.palletsprojects.com/en/3.0.x/templates/>).

* {% **for** value **in** sequence %} ... {% **endfor** %} loops through all values in sequence and enables the use each element in the sequence as value in the part between for and endfor in the template. The green bordered text is written in the host language, i.e., not part of the template language, but it can contain other template parts.
* {{ expression }} show the value of expression as output of the template.
* {# expression #} are comments and the contents will not be shown in the output.

For example, the following template…

1. {# Loop through the list of names and output a greeting. #}
2. {% **for** name **in** ['World', 'Maarten', 'Reader'] %}
3. Hello {{ name }}!
4. {% **endfor** %}

…results in the following output.

1. Hello World!
2. Hello Maarten!
3. Hello Reader!

To generate DML code for the physical model that was created in the previous section, the code shown below can be used.

1. dml\_template\_text **=** """
2. {# Define functionality to quote and join lists of names #}
3. {% **macro** q(names) -%}
4. [{{ names|join('].[') }}]
5. {%- **endmacro** %}
6. {# Define a macro to put commas ',' before items in a loop,
7. except before the first item #}
8. {% **macro** c(loop) -%}
9. {{ ' ' **if** loop.index == 0 **else** ', ' }}
10. {%- **endmacro** %}
11. {# Loop through all schemas and create SQL DML statements -#}
12. {% **for** schema **in** schemas %}
13. **CREATE** **SCHEMA** {{ q([schema.name]) }};
14. **GO**
15. {# Loop through all tables in the schema and output a CREATE TABLE statement -#}
16. {% **for** table **in** schema.tables %}
17. **CREATE** **TABLE** {{ q([schema.name, table.name]) }} (
18. {% **for** column **in** table.columns -%}
19. {{ c(loop) }}{{ q([column.name]) }} {{ column.fulltype }}
20. {% **endfor** -%}
21. , **CONSTRAINT** {{ q([table.primary\_key\_constraint.name]) }}
22. **PRIMARY** **KEY** ({{ q(table.primary\_key\_constraint.column\_names) }})
23. );
24. **GO**
25. {% **endfor** -%}
27. {# Loop through tables with foreign keys and output FOREIGN KEY constraints -#}
28. {% **for** table **in** schema.tables|selectattr('foreign\_key\_constraints') %}
29. **ALTER** **TABLE** {{ q([schema.name, table.name]) }}
30. **ADD**
31. {% **for** fkey **in** table.foreign\_key\_constraints -%}
32. {{ c(loop) }}**CONSTRAINT** {{ q([fkey.name]) }}
33. **FOREIGN** **KEY**
34. ({{ q(fkey.column\_names) }})
35. **REFERENCES** {{ q([fkey.foreign\_schema\_name, fkey.foreign\_table\_name]) }}
36. ({{ q(fkey.foreign\_column\_names) }})
37. {% **endfor** -%}
38. ;
39. **GO**
40. {% **endfor** %}
42. {% **endfor** %}
43. """

46. **def** generate\_code(template\_text: str, model: object) **-**> str:
47. """Generates code based on a model dictionary and a template string."""
48. **from** jinja2 **import** Environment, BaseLoader
49. template **=** Environment(loader**=**BaseLoader).from\_string(template\_text)
51. **return** template.render(**\*\***model.\_\_dict\_\_).strip() **+** '\n'

54. **def** generate\_dml\_code(physical\_model: PhysicalModel) **-**> str:
55. """Generates DDL code based on database model."""
56. **return** generate\_code(dml\_template\_text, physical\_model)

59. dml\_code **=** generate\_dml\_code(orders\_database\_model)

The first step of the code above is to define a template text. This template text might look a little bit daunting, but I’ll try to explain it. Note that the bordered text on the green background is SQL code, the rest is Jinja template code. First, a macro q is defined. This macro is used throughout the template to quote names. Then the actual work begins. The template loops through schemas and for each schema a CREATE SCHEMA statement is created. Then the template loops over each table in the schema and creates a CREATE TABLE statement that contains column definitions for each column in the table and a primary key constraint definition. The last part of the template loops through each table in the schema and generates an ALTER TABLE statement that adds foreign key constraint to each table that has foreign key defined.

The function generate\_code takes a template\_text string and a model object as input and returns a string. The resulting string is created by using a Jinja2 template object based on the provided template\_text and applying it to the provided model object by using the render method. This is the basic implementation of our template engine.

The generate\_dml\_code function applies the generate\_code function on the dml\_template\_text and a PhysicalModel object. Executing the code results in the following TSQL output.

1. **CREATE** **SCHEMA** [orders];
2. **GO**
3. **CREATE** **TABLE** [orders].[Customer] (
4. , [id] **uniqueidentifier** NOT NULL
5. , [address] **nvarchar**(255) NULL
6. , **CONSTRAINT** [pk\_Customer]
7. **PRIMARY** **KEY** ([id])
8. );
9. **GO**
10. **CREATE** **TABLE** [orders].[Product] (
11. , [id] **uniqueidentifier** NOT NULL
12. , [description] **nvarchar**(255) NULL
13. , **CONSTRAINT** [pk\_Product]
14. **PRIMARY** **KEY** ([id])
15. );
16. **GO**
17. **CREATE** **TABLE** [orders].[Order] (
18. , [id] **uniqueidentifier** NOT NULL
19. , [orderTimestamp\_utc] **datetime2** NULL
20. , [amount] **int** NULL
21. , [orderedBy\_Customer\_id] **uniqueidentifier** NOT NULL
22. , [orderFor\_Product\_id] **uniqueidentifier** NOT NULL
23. , **CONSTRAINT** [pk\_Order]
24. **PRIMARY** **KEY** ([id])
25. );
26. **GO**
27. **ALTER** **TABLE** [orders].[Order]
28. **ADD**
29. , **CONSTRAINT** [fk\_orderedBy\_Customer]
30. **FOREIGN** **KEY**
31. ([orderedBy\_Customer\_id])
32. **REFERENCES** [orders].[Customer]
33. ([id])
34. , **CONSTRAINT** [fk\_orderFor\_Product]
35. **FOREIGN** **KEY**
36. ([orderFor\_Product\_id])
37. **REFERENCES** [orders].[Product]
38. ([id])
39. ;
40. **GO**

## Conclusion

Based on the code in the example, you could wonder why you should go through all the trouble of defining a logical model, defining a model-to-model transformation and a model-to-text transformation to generate 45 lines of SQL code. Hand coding the SQL would have been more efficient.

Of course, you are right. For this example. In the next article I will discuss the MDSE principle to generate actual data warehousing artifacts such as DML and ETL code and lineage documentation. The fact that the same models will be the source for all these artifacts show that the benefits mentioned in this article will be more apparent if the number of models objects and output objects increase.