HAW HAMBURG

BACHELOR THESIS
Marcel Dießner

Model specification for template-based source code generation

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE
WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Marcel Dießner

Model specification for template-based source code generation

Bachelor thesis submitted for examination in Bachelor´s degree
in the study course *Bachelor of Science Angewandte Informatik*
at the Department Computer Science
at the Faculty of Computer Science and Technology
at University of Applied Science Hamburg

Supervisor: Prof. Dr. Stefan Sarstedt
Supervisor: Prof. Dr. Olaf Zukunft

Submitted on: 29. September 2022

Marcel Dießner

Title of Thesis

Model specification for template-based source code generation

Keywords

Meta-Model, Templates, Code-Generation, CobiGen, Pre-Processing

Abstract

Template-based source code generation is used in industrial generator frameworks as a fully automated process. Varying input data is used to evaluate placeholders in Templates. When less data is available than is required for template evaluation, many generators terminate, leading to no generation output. By pre-processing input and template, missing data can be discovered and manually added by the generator's user to the input data, before the actual template evaluation. This ensures that the generation will give output under any incomplete input, with the downside of falling into a semi-automated process. In this thesis we will explore this idea in detail, investigating different popular Template Engines and Frameworks and afterwards applying our solution to advanced template-based source code generators, to collect problems and future ideas to be expanded on further.

Kurzzusammenfassung

Die Template-basierte Source Code Generierung wird in modernen Generator-Frameworks als vollautomatischer Prozess eingesetzt. Variierende Eingabedaten werden verwendet, um Platzhalter in Templates auszuwerten. Wenn weniger Daten zur Verfügung stehen, als für die Evaluation einer Templates benötigt werden, brechen viele Generatoren ab, sodass keine Ausgabe generiert wird. Durch die Vorverarbeitung von Eingabe und Template können fehlende Daten entdeckt und vom Benutzer des Generators manuell zu den Eingabedaten hinzugefügt werden, bevor die tatsächliche Template-Evaluierung durchgeführt wird. Auf diese Weise wird sichergestellt, dass die Generierung auch bei unvollständigen Eingabedaten eine Ausgabe liefert. In dieser Arbeit wird diese Idee im Detail erläutert, indem verschiedene populäre Template Engines sowie Frameworks analysiert werden und anschließend unser Lösungsansatz auf komplexe Template-basierte Source Code Generatoren angewandt wird. Entstehende Probleme und potentielle Weiterentwicklungsansätze können so gesammelt und künftig weiter vorangetrieben werden.

Contents

List of Figures

List of Tables

Acronyms

CobiGen Code-based incremental Generator.

CRUD Create, Read, Update, Delete.

DI Dependency Injection.

IDE Integrated Development Environment.

ISP Interface Segregation Principle.

SoC Separation of Concerns.

TMM Template Meta Model.

TPP Template Pre-Processor.

UML Unified Modeling Language.

1 Introduction

In the era of agile[1] software development, where faster deliveries and smaller code increments lead to continuous changes, Code Generation grants the developer more utility than ever before. Commonly used to avoid creating boilerplate code manually, such as *getter* and *setter* methods, there is an ever-increasing range of possibilities, offered by complex code generator frameworks[10]. Not only can initial skeletons be generated, but also an existing code base can be modified, in the context of incremental Code Generation[7].

There are multiple kinds of template generators, which differ on fundamental approaches. The most commons are *abstract syntax tree* based, *printf* based, *term rewriting* and *text-template* based Code Generators[2]. While they all have their advantages and disadvantages, this thesis will operate exclusively in the context of template-based code generation.

According to Arnoldus, a template is a text document, which can include so-called placeholders. Placeholders contain some action, or expression, declaring how to obtain a piece of text to replace it [2, p.20] and can be described by the regular expression (*text*| *placeholder*)+. While *text* is the static content of a template, which is directly copied to the output, *placeholders* are dynamically replaced on generation. This is done by a *Template Evaluator* (see Figure 1.1). The *Template Evaluator* fills the templates, for which it needs the target template and data to fill it with. This process is managed by a *Code Generator*, a meta program, which creates a data model and initializes the evaluator to fill the template with the data model. When templates are evaluated, they result in a document only consisting of *text*.

Template Engine is a term often synonymously used for a *Template Evaluator*. While the term *Template Evaluator* is repeatedly used by Arnoldus for the core component of template processing, the term *Template Engine* is mainly used in correlation with

an actual system and product, such as FreeMarker[1], Velocity[2] or Mustache[3], which use different implementations. Therefore, we will use *Template Evaluator*, when referring to the abstract component used in every template-based code generator, while referring to the actual implementations, such as FreeMarker as *Template Engines*.

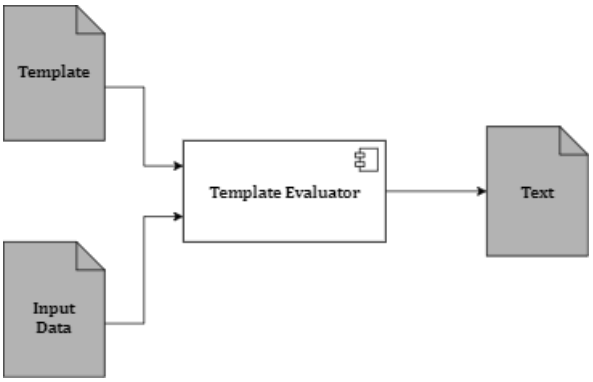

Figure 1.1: Template Evaluator

In this context, the terms object language and meta language are of great relevance. Meta language is the program language used for the placeholders, while object language is the programming language used in the non-placeholder part of a template and in the generated output. This implies that templates non-placeholders are not just seen as text, but as code of a programming language. See Listing 1.1 for example, which uses FreeMarker as a fixed meta language, while the object language can be chosen freely, though here it is intended to output Java code.

There are many code generation tools and frameworks that can help a developer in his daily work, like *CobiGen* or *JHipster*[4]. Additional to including one or multiple Template Engines, they also support different input and output options. From small input data, whole architectures can be generated, where only the business logic needs to be added to. Not only can they be used for creating new projects from scratch, but also for enriching existing projects with smaller code generation fragments. Next to the obvious reduction of work, this also helps to establish and maintain standards, both general coding standards, but also corporate specific standards.

[1] https://freemarker.apache.org/
[2] https://velocity.apache.org/
[3] https://mustache.github.io/
[4] https://www.jhipster.tech/

1.1 Problem Statement

In industrial applications, source code generation is often a fully automated process. While templates have to be created by a developer beforehand, the data model is created in most applications automatically, based on a different input source, such as an *Entity* class. This may lead to a deviation between the set of given data D_G and the set of required data D_R. While there is no problem, when $D_R \subseteq D_G$, there is one that needs to be addressed by *Template Engines*, if $D_G \subset D_R$, meaning the given data contains less information than required.

An example template is shown in Listing 1.1. This template uses the FreeMarker syntax and is evaluated with an example FreeMarker data model depicted in Listing 1.2. Listing 1.3 shows the result of this generation. While in this case the generation is successful, because there is no deviation, Table 1.1 lists examples where D_G misses required data.

	Substitute line	by	creating problem
A	1.1 line 1	class ${nam} {	nam is no valid key in the model 1.2
B	1.2 line 3	name = *null*	name has no value

Table 1.1: Possible deviations between template and data

Even though *A* and *B* have a different origin, *Template Evaluators* treat them as if they were the same. There are two possibilities on how *null* values can be handled on evaluation.

There are *Template Engines*, which always generate a result, such as Mustache or Velocity. Placeholders that can't be substituted simply end up blank. Both examples of Table 1.1 would result in `class {...}`. On the other hand, some *Template Engines* throw exceptions if a placeholder can't be substituted. By default, FreeMarker is interrupting on *null* values, though it can be configured otherwise.

Currently, starting a generation process with an interrupting evaluator does not offer any kind of guarantee, that the generation will build an output. This is especially important if a batch of dependent templates is rendered in a generation process. Missing one value in one template corrupts the whole generation, which would be very inefficient, when the exception occurs almost at the end of the generation process. If an entire class isn't available in the generated context, it's very likely that at least one other class is related to it, which would definitely result in a compilation error.

```
1 class ${name} {
2     ${vis} ${name}() {}
3     <#list fields as field>
4         private ${field.type} ${field.name};
5         ${field.type} ${"get" + field.type}() {
6             <#if log>
7             System.out.println("get" + ${"\""+ field.
      name +"\""}+"() is called.");
8             </#if>
9             return ${field.name};
10        }
11    </#list>
12 }
```

Listing 1.1: Java Template Example

```
1 (root)
2  |
3  +- name = "Customer"
4  |
5  +- vis = "public"
6  |
7  +- log = true
8  |
9  +- fields
10    |
11    +- (1st)
12    |   |
13    |   +- name = "firstName"
14    |   |
15    |   +- type = "String"
16    |
17    +- (2nd)
18        |
19        +- name = "lastName"
20        |
21        +- type = "String"
```

Listing 1.2: Freemarker Data Model Example

```
1 class Customer {
2   public Customer (){}
3   private String firstName ;
4   String getfirstName (){
5     System.out.println("get" +"firstName"+"() is
      called.");
6     return firstName ;
7   }
8   private String lastName ;
9   String getlastName (){
10    System.out.println("get" +"lastName"+"() is called
      .");
11    return lastName ;
12  }
13 }
```

Listing 1.3: Generation Result Example

While this may seem as a disadvantage at first, specifying cases where interruptions are allowed to happen, establishes quality requirements for the output. Completing a generation without interruption, but with the drawback that there is no guarantee that the output is as desired, is a big issue for the developers relying on generated code. This is especially important for syntax and type safe *Template Engines* (more in Chapter 2) to deny any output which is not syntax or type safe.

Still, for some Use-Cases, it might be acceptable for the input data to be incomplete. Many Code Generators allow different input sources, such as Entity Classes, UML diagrams, or OpenAPI[5], and differ in the information D_G, they can deliver. OpenAPI is good at documenting external interfaces, but in general does not have knowledge of internal components. Though, this data might be important to generate useful output. Giving the opportunity to add further data, i.e., through manual input, when the generation would be interrupted otherwise, not only increases the chance of successful generation, but also increases the usefulness of those code generators. Instead of deciding between full output or no output at all with a fully automated process, the main idea is to use a semi-automated backup process, when it can't be guaranteed that all required data is available at evaluation time. Therefore, the code generator needs to know before evaluating any template of a predefined set, if it has the required data to evaluate all of them without interruption, due to missing data and giving feedback as soon as possible, following the *Fail Fast Principle*[14]. If a comparison of templates and data model found required data is missing, the code generator has to go into the semi-automated process to get the missing data from the user through manual input or by other means. If all required values are available, the process stays fully automated.

1.2 Outline

The aim of this thesis will be to establish the described semi-automated backup process for template-based source code generation.

First, in Chapter 2, we will investigate template-based source code generation approaches and applications, where our process might be applied to. We will look at CobiGen exemplary for a full framework, as well as current approaches in syntax- and type-safe code generation on the example of Repleo and SafeGen. Also, we will look at current possibilities for missing value checking.

[5] https://www.openapis.org/

Chapter 3, will describe the process for recognizing and fixing missing data, from the input, that is required for generation. We will set the requirements for this process. Templates can differentiate a lot between template languages and hold much data not required in this process. Therefore, we will define a model specification as an abstraction for template meta language, with general rules, to transform a template to this abstract model.

Afterwards, Chapter 4, will apply the specified process of Chapter 3 to the template-based source code generator examples presented in Chapter 2, to discuss the usefulness and possible problems or difficulties in an advanced and industrial context.

At last, we will summarize our process, scientific findings and issues that have to be addressed in future work.

2 Related Work

This chapter will discuss related technologies in this area. Before going into detail, it is important to mention that currently there exist no approach targeting a semi-automated process or any kind of manual intervention before evaluation time. Therefore, the following three sections will present popular tools or frameworks based on template-based source code generation. The goal is to find out, if they use built-in error handling, and to gather additional requirements our solution has to address. Repleo, SafeGen and CobiGen are those technologies that we will investigate. At last, we discuss built-in expressions of some *Template Engines* that target *null* values.

2.1 Repleo

Arnoldus, Bijpost and van den Brand introduced Repleo[4], a system to build configurable syntax-safe *Template Evaluators*.

Arnoldus defines three classes of safety for code generators:

- no-safety
- syntax-safety
- type-safety

Syntax-safety guarantees that the output code of the generator is syntactically correct. This means that every output sentence is a valid sentence of the object language and can be parsed.

Type-safety, additionally, guarantees that the output is semantically correct. This means that duplicate variable declarations or type errors are detected, which is not detected by a syntax check. Therefore, every output sentence can be parsed and compiled successfully.

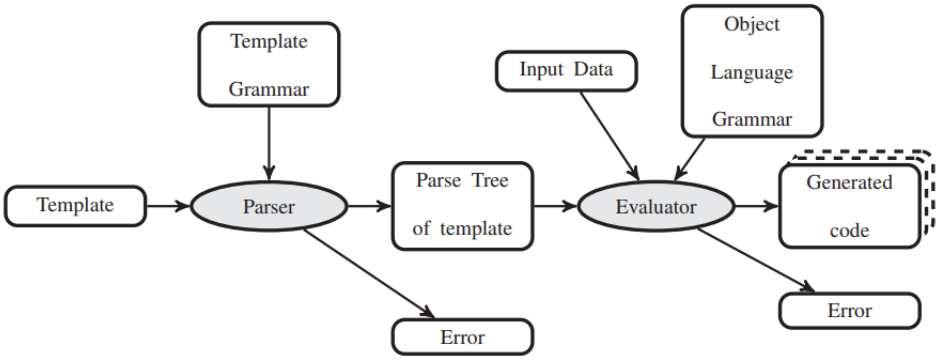

Figure 2.1: Syntax-safe architecture of Repleo [2]

If neither can be guaranteed, the evaluator is classified with *no-safety*. Most engines are classified with this, such as FreeMarker, Velocity or Mustache.

To guarantee syntax-safety, Repleo extends the one-phase model shown in Figure 1.1 to a two-phase model, shown in Figure 2.1, by processing templates internally as a *parse tree*. With this approach, templates are forced to follow a defined template grammar. This template grammar is a combination of object and meta grammar and production rules to connect them, which need to be configured manually[2, chapter 5.1]. After parsing in phase one, the syntax-safe parse tree can be evaluated with the input data to generate code in phase two. Important for us are the two shown errors in the figure. The first error occurs, when the template can't be parsed with the template grammar. The second error occurs, if the evaluator is not able to evaluate a placeholder of the parse tree to generate code that follows the defined object language grammar. This is either because the concrete value of the input data is not leading to syntax safe code or because the value is missing.

Repleo handles missing values, like the mentioned non-safe evaluator FreeMarker, by throwing an exception. Our solution should ideally be applicable to all *Template Engines*, which include syntax-safe evaluators. To support Repleo and similar approaches, the architecture of our solution should not be constricted by the internal processing of the templates by the evaluator. Therefore, our approach will only tackle the second mentioned error by Repleo and only if it's because of missing values. Anything else would harm the soundness of the system based on syntax-safety.

2.2 SafeGen

Next, we will look at SafeGen, a meta-programming tool that claims to be the only one that guarantees the type-correctness of the generated program at the compile time of the generator[9]. There might be certain bugs within a generator that occur only under certain input, which stays undetected for a long time. This is why SafeGen uses static checks, which do not try to find errors in the generated code or the input data, but in the generator directly. Therefore, if a generator written in SafeGen passes its compilers tests, it guarantees to only generate type-safe Java programs, not only for the given input, but for all possible inputs.

While Repleo allows the use of different programming languages and customizability for the developer by introducing new components, SafeGen is only applicable for generating Java programs. Errors are not handled differently as in previous approaches, though there are more cases in which an error occurs, so that the type-safety can be guaranteed. Important terms mentioned by Huang u. a. are *Soundness* and *Usefulness*[9]. Soundness in this case means that any output is correct depending on a given criterion. If the generator's goal is to be syntax-safe, then every output generated has to be syntax-safe for it to be sound. Therefore, if not syntax-safe code might be generated, it has to be rejected. This implies that rejecting everything and generating no output achieves soundness. This leads to Usefulness, which can't be measured and is based on personal judgement. But seeing the previous statement, it should be easy to agree that no output is not useful at all for a developer. By decreasing the chance for an interruption with our approach, we aim to increase the usefulness of any *Template Engine* adapted to it. The level of soundness, on the other hand, should be not meddled with. Each engine sets different quality requirements to their generated output, therefore me must guarantee that our pre-processing does not decrease the quality requirements defined by the *Template Engine* in use.

SafeGen templates use a special meta language syntax. Listing 2.1 shows an SafeGen example, which can generate an Interface including all method signatures for an input class. Firstly, SafeGen allows the use of a type based system in templates, which is used by the so-called *Theorem Prover* to statically check the syntax. Therefore, one can use types such as Class, Method, Field, Interface, etc. and use specific functions on them. A SafeGen template consists of two parts, the Generator definition(`#defgen ...`) and the body inside `{...}`. The Generator definition has an identifying name and can have an

```
1 #defgen makeInterface (Class c) {
2     interface I {
3         #foreach(Method m : MethodOf(m,c)) { void #[m]
       (); }
4     }
5 }
```

Listing 2.1: SafeGen: Generate an Interface for any Input class

input inside the brackets, which are either cursors or predicates describing constraints on the inputs. Cursors are a SafeGen concept to specify a type declaration, such as:

```
Method m : MethodOf(m,c) & Public(m) & !Abstract(m)
```

Here the Method m is specified as being a method of a class c, has to be public and non-abstract. In the generator definition, it may be used to specify to accept a single input class, such as:

```
#defgen myGen (Class c : !Abstract(c)) { ... }
```

A generator can also get a set as an input and generate for each element of that set by using a predicate, such as:

```
#defgen myGen (input(Class c) => !Abstract(c)) { ... }
```

Inside the body one can use three SafeGen constructs, which are `#[...]`, to access variables, `#foreach` and `#when`, similar to other template languages.

2.3 CobiGen

CobiGen[1] is an incremental source code generation framework, introduced by Brunnlieb and developed by *Capgemini* and available in open-source.

The term *incremental* refers to the process of code generation on an existing code base. Generated code can create a conflict with the existing code base, therefore requiring structural merging to resolve them. The source code is incrementally enriched according

[1] https://github.com/devonfw/cobigen

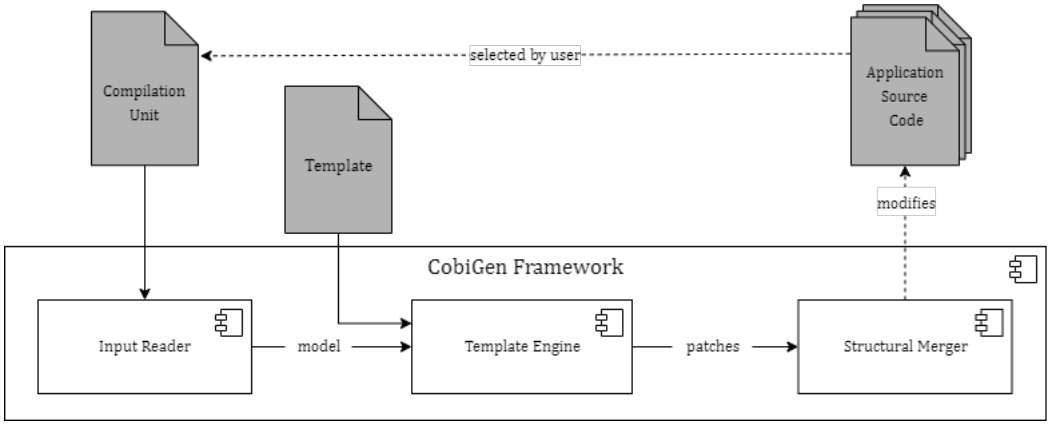

Figure 2.2: Main CobiGen Components with inputs and outputs[5][6, Fig.3 (translated)]

to the wishes of the user and encourages him to work towards small increments. Increments are a collection of templates, and define a unit within a use-case. Figure 2.2 depicts the main components of the CobiGen architecture, with its inputs and outputs. The Compilation unit consists of the input files selected by the user and defines the first constraint of generateable increments, that are possible based on the input files. In a second step, the user is prompted to select one of those increment options. The selected increment defines which templates have to be generated.

CobiGen can generate code from templates on every layer of a chosen target application. For example, it allows generating a whole CRUD application from a single Entity class. In this case the single Entity class is taken as input, converted internally into a data model, which is then used to fill the required templates needed for a full CRUD application, such as DAOs, Transfer Objects and simple CRUD use cases with REST services. While the whole CRUD application is the use-case, DAO's, REST services, etc. define each an incremental unit. The filled templates are referred to as *patches*, which are then merged with the code base, allowing multiple merging strategies. The main strength of CobiGen is its wide area of application, due to its many extensions, which are shown in Figure 2.3. Input Reader implementation allow files to be parsed and the data converted to an input model, that can be used for template evaluation. CobiGen can be extended to work with different Template Engines as well. It mainly uses either FreeMarker and Velocity, but also offers syntax-safe code generation with the *Syntax Safe Java FreeMarker Template Engine*. Evaluated templates result in a patch, which is already a valid output. However, CobiGen can work on an existing source code base, where the generated file, can lead to merge conflicts with an existing base file. Structural

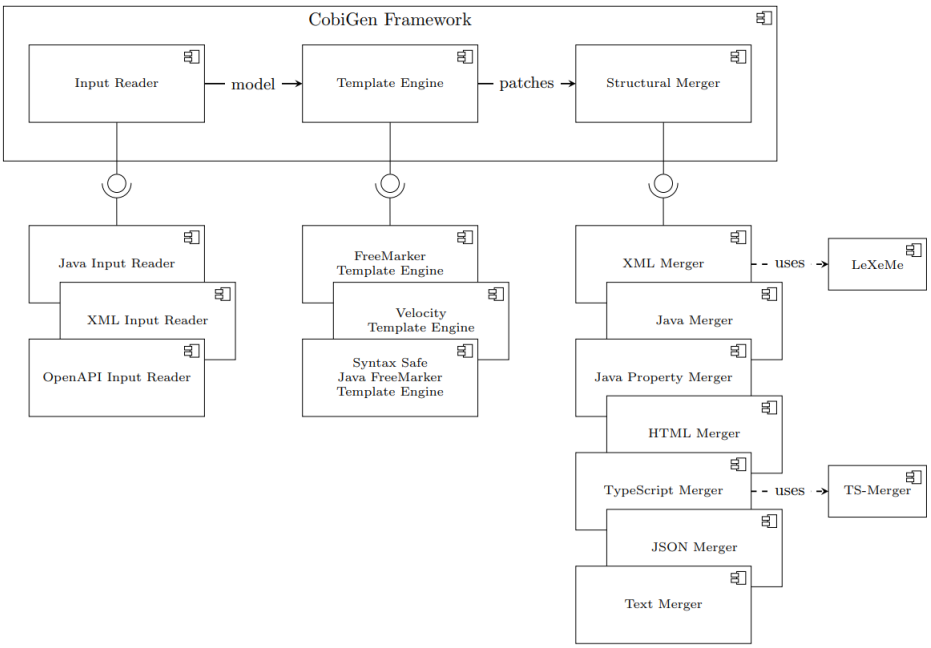

Figure 2.3: CobiGen extension points [5, Figure 4.6]

Merger implementations define which file formats can be merged and define the strategy to do so. Some might be as simple as to override the base file completely, while others allow complex merge strategies.

Each component defines a set(S) that includes all of its implementations. Every combination of the component's implementations is theoretically possible, resulting in the Cartesian product:

$$S_{InputReader} \times S_{TemplateEngine} \times S_{StructuralMerger} \tag{2.1}$$

And still, if a specific *Template Engine* or input type is not yet supported, it can be adapted as well. It offers great help for the developers due to Eclipse IDE integration and by improving the source code with small increments on the user's needs. Through a User interface, the developer is prompted to select what he wants to generate. The options are pre-filtered by the type of input, called Compilation Unit in the CobiGen context.

CobiGen uses a directory structure for relating templates, exemplary shown in Figure 2.4. There are two important types of configuration files, that define which templates have to be generated. These are the context configuration and the template configuration.

```
CobiGen_Templates
 |- templateFolder1
    |- templates.xml
 |- templateFolder2
    |- templates.xml
 |- context.xml
```

Figure 2.4: CobiGen Directory Structure

The Context Configuration[2] is defined within a file called `context.xml` in the root directory of a template directory. Based on the input compilation unit, the context configuration selects which template folders are worth generating and therefore defines the input based pre-filtering of generateable use-cases mentioned earlier.

The Template Configuration[3] is defined within a file called `templates.xml`, which needs to be in the according template directory (i.e., `templateFolder1` or `templateFolder2`). It defines the available increments that can be generated. As mentioned, the user is prompted to select one or multiple increments. The user selection defines which templates need to be generated.

Because of the different input and output options, there are a lot of combinations that can occur. Each has a different amount of data available(D_G) and required(D_R) and therefore results in a varied-sized difference of missing data D_M, where $D_M = D_R - D_G$. Currently, there is no automatic prediction of the size of D_M for each combination, which is why the context configuration has to be defined manually to set which combinations are allowed and based on their likeliness to succeed. With our, approach, we might be able to support more options.

[2] https://github.com/devonfw/cobigen/wiki/cobigen-core_configuration#context-configuration

[3] https://github.com/devonfw/cobigen/wiki/cobigen-core_configuration#templates-configuration

2.4 Built-in expressions for missing values

Some *Template Engines* offer options for the handling of missing values. FreeMarker has expressions to define default values and expressions to check, if a placeholder is missing, shown in Listing 2.2. While this is a good way to handle those errors, it still has some issues.

For once, it needs to be offered by the engine of use, while we try to find a process which works on every *Template Engine*. Also, these options mean extra work for the developers for every template there is, so that each placeholder would need to be checked before use. This would increase the complexity of the templates, while not directly solving our problem. If the value `name` would be missing, the whole section would just be left out, which depending on the context can be crucial, like in the example (Listing 2.2), where there would be no class declaration in the context of a Java class.

Our approach should not rely on the developers having to redesign templates with those mechanisms. Of course, if a placeholder is not seen as necessary in the context of the template, such techniques can be applied. Though, this introduces another issue in the context of conditional directives for our approach, which is how to handle the condition statement.

It can either ignore the condition and its result on evaluation and make sure to cover every placeholder in any branch, or pre-evaluate the condition before template evaluation time and only cover the needed branch. More to that in Section 3.1.4.

The expression to set default values, i.e.,`${name!"DefaultName"}`, on the other hand, does not need pre-evaluation and should be easy to parse as well. In this context, `name` always has backup value, so it's never seen as a potentially missing value for evaluation. This has to be defined as a rule on template parsing.

In any other case, each placeholder should be seen as necessary for the evaluation, and our approach should guarantee that they are available at evaluation time.

```
1        <#if name??>
2            class ${name}
3        </#if>
```

Listing 2.2: FreeMarker value checking

3 Template Pre-Processing for semi-automated Code Generation

As described in the problem statement in Section 1.1, the goal is to implement a semi-automated backup process to deny interruptions that break the whole generation process, caused by missing input data. Thus, increasing the usefulness of code generators, while maintaining the soundness. To achieve that, we need a process that compares the data we have in the current instantiation with the data we require for all selected templates through a target/actual-comparison, followed by requesting missing data found in the comparison. The requested data then needs to be added to the data model, which we will refer to as *enriching* the data model. The actual side is already available in the form of the data model at runtime. Though different *Template Engines* often work with slightly different internal representations of this model, fundamentally they are all tree-based. The initialization of it can be done with basic data types, such as lists or arrays, maps and strings or numerics, which will then be transformed into the internal structure through object wrapping. FreeMarker classifies the objects into scalars and containers, terms we will use too.

Scalars are non-compound data types, which means the data type only contains a single value[13]. This can be numeric values, boolean, strings and temporal data such as dates. The opposite are compound data types.

Compound data types are allowed to store multiple values [12]. Containers are a form of compound data types, that are allowed to store different typed values, compared to arrays that only allow the same type of values. In the context of template processing, the important distinction is between the abstract data types of sequences and maps. Both associate a unique identifier to each of its values. Sequences use an integer value referred to as *index* as identifier, starting with 0 and increasing for each value, while maps associate a string to each value. Some *Template Engines* such as FreeMarker allow custom objects as containers, if they follow certain requirements.

For the target side, we have to rely on the information the templates offer. Though templates differ much more in their structure and design between different *Template Engines* than the data model, they contain information we don't actually need for a comparison. We only require the used meta language of the templates, which although differs in syntax, are quite similar in their semantics, which we can represent in a model. This model should hold the required data in a simple, human-readable form, including all used variables in a template and the context they appear in, such as loops or conditionals. We will refer to this model as Template Meta Model (TMM).

Requirements for the Template Meta Model

Most importantly, the TMM shall be derivable from a template. Therefore, common directives used in templates can be translated to the TMM. Directives covered in Section 3.1 are:

1. variables (local and global)
2. iterations over sequences and maps.
3. conditionals (if, else if, else)
4. functions
5. macros
6. references to other files through imports and sub-templates

ReqF1: The translation from template to TMM is defined unambiguously by Transformation rules.

ReqF2: Transformation rules should have an appropriate complexity to enable simple manual creation of TMM.

ReqF3: Each template shall be associated with exactly one TMM.

The TMM lays the foundation for the request of missing values and enrichment of the data model, in case variables are missing. At the same time, it should be possible to depict relations for these variables and special contexts, defined in a template. For example, a template can reference a nested variable, such as `field.name` in FreeMarker, where `field` is a reference to a container, such as map or sequence and `name` is a variable defined in what is called the local scope of `field`. Therefore, `name` is dependent on the

existence of a container `field`. However, there are also template directives that do not or not necessarily create a scope. Conditionals do not create a scope, while functions allow to create a scope based on their arguments, which will be covered in more detail in Section 3.1.4 and Section 3.1.5. They are primarily used for structuring the meta code in a template. Therefore, TMM elements can be categorized into *data elements* and elements that are primarily used to structure the meta code and will not be used to compare and request data, such as conditionals, which we will refer to as *structure elements*. Structure elements create a special context, not to be confused with a scope. While scopes define the region in which a variable is valid, the structure context defines where a variable is used and therefore required within a template. The FreeMarker example, shown in Listing 3.1, should demonstrate the difference. `field` defines a local scope in which `name` is valid, while the if branch defines the context in which `field` and therefore `name` are used and required. Thus, given print is *true*, `field` and `field.name` need to be available in the data model in that context.

```
1 <#if print>
2 print ${field.name};
3 </#if>
```

Listing 3.1: Context vs. Scope example

ReqF4: TMM data elements should be modelled based on their expected abstract type: scalar, sequence or map.

ReqF5: TMM structure elements should be modelled based on the contexts they create: conditional branches, iterations, and references to functions or different template files.

ReqNF1: Readability: Structure and data elements should be reduced to a flatter representation, given that it does not change the meaning.

By transforming a template engine specific template into a general representation, the TMM can be used in the later process without requiring template engine specific knowledge.

ReqF6: The TMM uses an abstract representation, independent of *Template Engine* specifications.

The last requirement for the TMM targets the organization within a template directory. Multiple TMMs should be allowed to be stored in one file. This allows the user to define

one file per TMM on the same directory level as the according template, or to create one file per directory level or for all subdirectories.

ReqNF1: Usability: TMM's can be defined in one or separate files to allow a diverse usage.

Pre-Processing System Requirements

With the TMM we can implement a process, which runs before the evaluation of any template, that includes the comparison of a TMM and a data model, as well as the enrichment of the data model, referring to the process of adding the missing data to it.

ReqF7: The comparison delivers a set of missing values D_M with their respective type, according to *ReqF4*.

The missing data needs to be requested from the user i.e, via a command line interface. Only *scalars* should be requested. If a whole container is missing, all nested scalars should be requested and be arranged in to the according container by the system.

ReqF8: The user has to be prompted for all missing scalar values.

A missing sequence is allowed to request multiple elements, even though one would be enough for a successful check. If an entire sequence is undefined, it is likely that the user wants to define more than one element. For example, if the sequence *fields* is undefined, when the template in Listing 1.1 is supposed to be generated, the user might want to add more than one field.

ReqNF2: Usability: If a sequence is missing, multiple elements can be requested.

The data model should only be enriched with the user-requested data, meaning values can be added to it, but none should be removed or reorganized, because that could lead to unwanted consequences.

ReqF9: The data-model is only be allowed to be queried and enriched.

The pre-processing system should be expandable to different *Template Engines*, therefore should work with different engine architectures such as Repleo's two-phase model. Given the TMM follows *ReqF6*, the following process can be template engine independent as well, relying on interfaces for template engine specific tasks. This reduces the coding work

needed to extend the process for other template engines, because only the interfaces have to be specified.

ReqNF3: Extensibility: Decoupling of template engine specific behavior from the core components through extension points.

3.1 The Template Meta Model

A Template Meta Model is a short representation of a template reduced to the placeholders and the context they occur in. It is used as the basis for the later comparison. To create the TMM, according to *ReqF1*, it is important to establish transformation rules(TR) from template to TMM. This makes sure that a TMM can be unambiguously created from a template manually. Formally, this means:

$$f : template \xrightarrow[TR]{} tmm \tag{3.1}$$

Thus, we can't require information in the TMM, that is not available within a template.

This may seem as a simple mapping process, where template directives are mapped one-to-one. However, following *ReqNF1* in the transformation process, we can differentiate another type of transformation rules. In addition to mappings, we need rules that simplify the result of mappings based on the context they are mapped to by removing unnecessary structures. We will refer to those rules as *mapping transformation rules* and *reduction transformation rules*. The reduction transformation rules can be applied repetitively, until none are applicable, and the final TMM is created. The following sections, will describe transformation rules mostly in abstract form, explained mainly on FreeMarker examples. These abstract transformation rules need to be specified for a *Template Engine*, considering the engine specifics. It will be described how these rules should be prioritized to guarantee an unambiguous transformation.

3.1.1 The Foundation

First, a data format had to be determined. After looking at the many popular options, like YAML, JSON and XML, it would not be much of a functional difference to use any of them. We decided to use XML, due to the compact structure. XML Nodes allow the human reader to easily distinguish the kind of model element, and attributes reduce the

```
{
  "tmm": {
    "file": "templates/Template2.ftl",
    "templatEngine": "FreeMarker",
    "scalars": [
      {
        "key": "name"
      },
      {
        "key": "vis"
      }
    ],
    "sequences": [
      {
        "key": "fields",
        "element": "field",
        "scalars": [
          {
            "key": "type"
          },
          {
            "key": "name"
          }
        ]
      }
    ]
  }
}
```

Listing 3.2: TMM JSON Example

number of lines and nesting. Compare Listing 3.2 and Listing 3.3, which store the same information. The JSON example needs thrice the lines as the XML example.

Therefore, we will model a TMM, as shown in Listing 3.3 with the `tmm` node. To uniquely identify the related template, the `tmm` node requires the relative `PATH` from the XML file to the template in the `file` attribute, according to *ReqF3*. Optionally, the `templateEngine` attribute allows specifying the *Template Engine* which is being used. Although the structure of the TMM is general, keeping track of the *Template Engine* is helpful, mainly for engine specific configuration. Specific rules for the TMM, such as determining fast, which features the *Template Engine* is supposed to support and checking if the TMM includes some that are not supported, might be configured. Additionally, it could be used to load the right context of classes, for example with Dependency Injection (DI).

```
1 <tmm file="templates/Template2.ftl" templateEngine="
      FreeMarker">
2     <scalar key="name"/>
3     <scalar key="vis"/>
4     <sequence key="fields" elementKey="field">
5         <scalar key="type"/>
6         <scalar key="name"/>
7     </sequence>
8 </tmm>
```

Listing 3.3: TMM XML Example

```
1 <tmmConfiguration version="VERSION_NUMBER">
2     <tmm file=PATH [templateEngine=ENGINE]>
3         ...
4     </tmm>
5     ...
6 </tmm>
```

Listing 3.4: TMM XML Frame

XML only allows one root node, so to manage multiple TMMs in one file, to fulfill *ReqNF1*, each XML file including at least one `tmm` node has to be organized in a `tmmConfiguration` node, as shown in Listing 3.4.

Because there are currently no similar approaches, we had to start from scratch and develop the model exploratively by increasing the complexity of the TMM with each directive. The next subsections will cover each of these directives, showing how we modeled those features and describing the transformation rules as required by *ReqF1*, as well as describing issues that occurred while exploring. We will mainly use the *FreeMarker Template Engine* syntax for examples.

3.1.2 Variables

The core of every template are variables. They only have an identifying string, which when evaluated should match a key of the data model and is then replaced with the assigned value. In FreeMarker they are used as `${identifier}`. The identifier can refer to a key in the global and local namespace.

The global namespace is only implied by a template, because it refers to keys that need to be defined outside the template in the data model. Global variables can be used anywhere in a template, given the key-value pair exist in the data model.

Local namespaces are defined within a template. FreeMarker for example allows the directive `<#assign key=value>`, where a local variable can be declared and assigned to a value. Also, local variables can be declared in an iteration directive, where the elements of a sequence can be referred to over a locally defined name. Local variables are only accessible within the scope they are declared in. Thus, sequence element variables can only be accessed within the iteration directive scope and not outside of it.

The variable itself, in general, does not give information about the expected type. Therefore, the type of the expected value is derived from the context it appears in, which is why variables cannot be modelled in a unified way, but need to be distinguished, based on the context. The variable identifier can be used as the key for the data elements. Those are scalars, maps, and sequences.

Scalars

Scalars are modeled via the scalar node `<scalar key=IDENTIFIER/>`, as shown in Listing 3.5.

```
1 <tmm file=PATH [templateEngine=ENGINE]>
2     <scalar key="name"/>
3 </tmm>
```

Listing 3.5: Scalar node

It only has a key attribute, which equals the key used in the template. Scalars have to be unique through their key in the same scope, for instance it's not allowed to have two `scalar` nodes with the same key in the global scope. Additionally, this means, that we only need one representation of the same variable reference in the TMM, even though it might be used multiple times within a template.

Containers

Maps are always implied in templates. FreeMarker uses unescaped dots '.' in the variable identifier, i.e., `book.title`, which implies that the identifier `book` refers to a map

including the variable `title`. Repleo uses a '/' instead of a dot, while Mustache chooses a different approach. Listing 3.6 shows a Mustache template. Mustache does not use a dotted notation for referencing local scopes, instead uses a notation that has a start- and endpoint, here declared with the namespace `author`. Every variable in between is used in the scope of that namespace. `name` and `age` are therefore only valid in the scope of `author`, which is therefore implied to be a container, specifically a map. Listing 3.7 shows an example data model, to evaluate the template fully.

```
1 {{#author}}
2   {{name}}
3   {{age}}
4 {{/author}}
```

Listing 3.6: Mustache Template

```
1 {
2   "author":{
3     "name": "Marcel",
4     "age": 24
5   }
6 }
```

Listing 3.7: Mustache Data Model

Maps have to be modeled as a separate node, in which dependent scalar nodes have to be nested, as shown in Listing 3.8. For the `book.title` example, `book` is the `SCOPE_IDENTIFIER` and `title` is the `NESTED_IDENTIFIER`. This also allows multiple nested maps.

```
1 <map key=SCOPE_IDENTIFIER>
2     <scalar key=NESTED_IDENTFIER/>
3     ...
4 </map>
```

Listing 3.8: Map node

This also means that if a key-value pair in a map is missing in the data model, it has to be checked as well, if a map has to be initialized first or if this map is existing but misses just this pair.

Each key in the TMM must be able to be addressed uniquely, by a list of keys from root to the selected key. In the example, the scalar-key is `title` the full unique path is `[book, title]`.

Sequences are mainly used for iteration in templates, therefore they will be modelled in Section 3.1.3.

3.1.3 Iteration

The second directive type, we will transform to the TMM are iterations. They require a sequence or in fewer cases a map as input. Listing 3.9 shows the use of iterations over a sequence in FreeMarker.

```
1 <#list fields as field>
2     private ${field.type} ${field.name};
3 </#list>
```

Listing 3.9: FreeMarker list directive

It also allows iterating maps with `<\#list persons as id, person>` or ranges such as `<\#list 1..10 as x>`. These templates introduce variables that are only locally available inside the *list* directive scope.

For the TMM we are mainly interested in the data element that is iterated, rather than the iteration itself. However, with a variable size that is only determined on runtime when the data model is created, the TMM has to work for any size of sequence. To guarantee that, the sequence node has to establish a pattern that all elements of that sequence need to follow.

```
1 <sequence key=KEY [elementKey=ELEMENT_KEY]>
2     ...
3 </sequence>
```

Listing 3.10: Sequence node

We will map iterations over lists or similar collections to the sequence node, as shown in Listing 3.10. It needs a *key* attribute like the scalar node, but also allows an element attribute, which is needed depending on the Template Engine. The *elementKey* attribute defines the local namespace under which dependent variables can be addressed with. For the previous FreeMarker template example, the TMM needs to include the sequence node, shown in Listing 3.11.

```
1 <sequence key="fields" elementKey="field">
2     <scalar key="type"/>
3     <scalar key="name"/>
4 </sequence>
```

Listing 3.11: Sequence node example

`fields` is the name of the list and is set as the key, while `field` is the local namespace under which nested variables will be addressed in the template. So the nested variable `name` in the list is addressed in the template as `field.name`. Some *Template Engines* don't use this kind of local addressing, but instead would address similar to Mustache, as shown in Listing 3.12. In contrast to FreeMarker, the Mustache template itself does not imply a sequence or map uniquely, but instead the type is determined based on the data model. Listing 3.13 shows an according data model including a sequence called `fields`.

```
1 {{#fields}}
2     private {{type}} {{name}}
3 {{/fields}}
```

Listing 3.12: Mustache Sequence Template

```
1  {
2    "fields":[
3      {
4          "type": "String",
5          "name": "firstName"
6      },
7      {
8          "type": "String",
9          "name": "lastName"
10     }
11   ]
12 }
```

Listing 3.13: Mustache Sequence Data Model

Special Cases

Maps are iterable in some engines as well, an example template is shown in Listing 3.14. However, this is only an abbreviation by skipping the local namespace definition, when there are only two needed nested variables. The extended version is shown in Listing 3.15.

```
1 <#list products as name, price>
2 <p>${name}: ${price}
3 </#list>
```

Listing 3.14: Map Iteration FreeMarker Template

```
1 <#list products as product>
2 <p>${product.name}: ${product.price}
3 </#list>
```

Listing 3.15: Map Iteration FreeMarker Template

Therefore, we can do the same to model this, which means leaving out the local namespace, which is defined by the `elementKey`, and including only two sub data elements, resulting in Listing 3.16.

```
1 <sequence key="products">
2     <scalar key="name"/>
3     <scalar key="price"/>
4 </sequence>
```

Listing 3.16: Modeling Map Iteration

Locally defined sequences, such as *<#list 1..10 as x>* don't need to be mapped. They do not hold any placeholder that needs external data, therefore we can ignore anything related to `x`.

Up-shifting

Still, there may be further directives or global variables inside the iteration directive, that are not in the iteration scope. This allows the mapped element to shift level upwards, meaning outside the sequence element within the TMM. The variable `fieldvis` in Listing 3.17 is global and thus not related to `field`. Figure 3.1a would create the impression, that `fieldvis` would be dependent on `field`, because the TMM does not use the full dotted path as the key. Therefore, the variable has to be shifted upwards outside the sequence node, as shown in Figure 3.1b. This process might be repeated, if there is multiple nesting, shifting the variable upwards multiple times. Because `fieldvis` is a global variable, it has to end up on top-level. *Up-shifiting* is a *reduction transformation rule* that needs to be applied after mapping an element. The same can be applied for sequences and maps. The purpose of nesting is to show dependencies between directives. After applying the *mapping transformation rules* there might be cases, where there is a nesting that does not express a dependency. *Up-shifting* is a means to achieve the true purpose of nesting.

```
1     <#list fields as field>
2         ${fieldvis} ${field.type} ${field.name};
3     </list>
```

Listing 3.17: List directive including a global variable

```
1 <sequence key="fields"
      elementKey="field">
2   <scalar key="fieldvis"/>
3   <scalar key="type"/>
4   <scalar key="name"/>
5 </sequence>
```

(a) Without variable up-shifting

```
1 <sequence key="fields"
      elementKey="field">
2   <scalar key="fieldvis"/>
3   <scalar key="type"/>
4   <scalar key="name"/>
5 </sequence>
```

(b) After variable up-shifting

Figure 3.1: Variable shifting in a list

3.1.4 Conditionals

Conditionals allow multiple branches depending on the evaluation of a condition. Conditionals are structure elements, which are different to iteration and have to be modeled directly. As mentioned, they establish a special context, unlike a scope. Placeholders used uniquely within a templates branch have to be nested within the according TMM branch as well. The context denies elements to be shifted upwards, unless certain criteria are fulfilled, which will be covered in the Subsection **Further Branch-Collapsing Rules**. There are four cases that need to be modelled:

- if
- if-else
- if-(elseif)+
- if-(elseif)+-else

Combined they result in the regular expression: $if - (elseif)^* - (else)^{0,1}$, which the TMM needs to depict. *If* and *else* can only occur once in a conditional context, while *elseif* can occur any number of times with different conditions.

Listing 3.18 shows, how Freemarker implements conditionals. Each if or elseif directive has a condition, that is also should be mapped into the TMM.

```
1 <#if condition1>
2     ...
3 <#elseif condition2>
4     ...
5 <#else>
6     ...
7 </#if>
```

Listing 3.18: FreeMarker Conditional structure

Conditional structures are mapped to a conditional node, as shown in Listing 3.19, that allows three subnode types: if, elseif and else. As always, the square brackets symbolize optionality, which means that every conditional node needs at least a nested if node.

```
1  <conditional>
2      <if condition=CONDITION1>
3          ...
4      </if>
5      [<elseif condition=CONDITION2>
6          ...
7      </elseif>
8      ...
9      [<elseif condition=CONDITIONn>
10         ...
11     </elseif>]...]
12     [<else>
13         ...
14     </else>]
15 </conditional>
```

Listing 3.19: Conditional Node Structure

The condition will be simply copied from the template as a string, as an identifier to link the conditional branch with its origin. However, the condition can include placeholders too, which need to be stored as scalar nodes outside the conditional node. See Listing 3.20 for example, which uses a conditional part depending on the value of `createGetterAndSetter` within a list-directive. If excluding line 2, `field.type` and `field.name` would only be used within the if-branch and not outside of it. So the according TMM part could look as depicted in Listing 3.21. This means the condition `createGetterAndSetter` needs to be evaluated for each element of the sequence `fields`. The condition is a variable itself, to be precise a scalar, and therefore need to be mapped

to a scalar node. Because it is not dependent on the local scope of `field` it can be shifted upwards, outside of the sequence node, resulting in in line 1 of Listing 3.21.

```
1 <#list fields as field>
2     private ${field.type} ${field.name};
3     <#if createGetterAndSetter>
4     public ${field.type} ${"get" + field.name?
      cap_first}() {
5         return ${field.name};
6     }
7
8     public void ${"set" + field.name?cap_first}(${
      field.type} ${field.name}) {
9         this.${field.name} = ${field.name};
10    }
11    </#if>
12 </#list>
```

Listing 3.20: FreeMarker Template example with conditional context

```
1 <scalar key="createGetterAndSetter"/>
2 <sequence key="fields" elementKey="field">
3     <conditional>
4         <if condition="createGetterAndSetter">
5             <scalar key="type"/>
6             <scalar key="name"/>
7         </if>
8     </conditional>
9 </sequence>
```

Listing 3.21: TMM of conditional structure within sequence

However, this leads to an issue when conditional and sequence nodes are combined. `type` and `name` are related to `field`, which is not cleary recognisable in Listing 3.21. Those variables are not allowed to shift outwards, due to the special context the conditonal structure introduces. Global variables might be nested on the same level inside this conditional branch. Therefore, the variables defined in a local scope need to be distinguishable from the global ones. Nesting them in a map node with the `key` equal to the `elementKey` attribute of the sequence node, allows local variables to be uniquely identified in this special case, such as shown in Listing 3.22 from line 4-6.

```
1 <sequence key="fields" elementKey="field">
2     <conditional>
3         <if condition="createGetterAndSetter">
```

```
4             <map key="field">
5                 <scalar key="type"/>
6                 <scalar key="name"/>
7             </map>
8         </if>
9     </conditional>
10 </sequence>
```

Listing 3.22: TMM of conditional structure within sequence (fixed)

With line 2 in Listing 3.20 however, this would look a lot different, because the two variables are used outside of a conditional branch as well, therefore making it unnecessary to cover them inside the if-branch node. This means that before mapping, it needs to be checked, if the variable in focus was already mapped in the same context on a higher level. If already mapped, such as in this case `field.type`, it does not need to be mapped a second time. Therefore, the map and its nested scalars in line 4-7 of Listing 3.22 do not need to be mapped. Having no nested elements, the if-branch loses its purpose, which is why it can be removed. Still the condition in this example refers to global variable, which we can't ignore. Therefore the scalar node with the condition variable has to remain. We refer to this *reduction transformation rule* as *branch-collapsing*, the result shown in Listing 3.23.

```
1 <scalar key="createGetterAndSetter"/>
2 <sequence key="fields" elementKey="field">
3     <scalar key="type"/>
4     <scalar key="name"/>
5 </sequence>
```

Listing 3.23: TMM with collapsed branch

The conditional nodes main purpose is to depict variables or directives that only exist within a condition-based context. Ideally, only the required context needs to be pre-processed. Unfortunately, this proves to be difficult to implement, because this would mean the condition has to be pre-evaluated the same way as the actual template evaluation to find the right branch. Otherwise, on comparison, every branch needs to be covered, which might require values to be requested, that aren't actually needed.

An idea is to create a temporary template only consisting of the conditionals, where at the end of each path lies a unique result, such as a numeric value for unique identification. The temporary template is then to be evaluated by the same *Template Evaluator* that is used for the original template and based on the output, it can be determined which

path has to be compared. Though, the actual solution is to be future work, because it would mainly improve the user experience, which has a lower priority at this point, while adding a lot of complexity to the process.

Further Branch-Collapsing Rules

To this point we only addressed, what has to happen, if only an if-branch exist, that does not include new variables. When dealing with multiple branches, new cases come up. If multiple branches exist, completely collapsing the conditional structure is only allowed, if every branch is empty. If more than one branch exists, where at least one includes previously unused variables, there are two distinguishable cases.

First, when the if-branch node or elseif-branch nodes have nested data or structure elements, any empty branch might be allowed to be collapsed. The else-branch, if empty, can be completely ignored. This is shown in Figure 3.3a and b, where the second elseif branch and the else branch are collapsed.

When an else-branch node or elseif-branch node has nested elements. Every empty branch node before that has to be transformed to a one-line node, such as `<if condition="CONDITION"/>`. This is necessary for the case that conditions are pre-evaluated, so that it is defined under which condition the else- or elseif-branch is used, where for elseif-branches the order of conditions is important. This is shown in an example in Figure 3.3c and d, where the if-branch of line 4-5 and the elseif-branch of line 6-7 are reduced to a one-line node each in line 4 and 5 of Figure 3.3d.

```
1 <conditional>
2     <if condition="condA">
3     </if>
4     <elseif condition="condB">
5     </elseif>
6     <elseif condition="condC">
7         <scalar key="x"/>
8     </elseif>
9     <else>
10    </else>
11 </conditional>
```

Listing (3.24) TMM representation of multiple branches before Collapsing

⟶

```
1 <conditional>
2     <if condition="condA"/>
3     <elseif condition="condB
      "/>
4     <elseif condition="condC">
5         <scalar key="x"/>
6     </elseif>
7 </conditional>
```

Listing (3.25) TMM representation of multiple branches after Collapsing

Listing 3.24 shows multiple branches, where only the elseif-branch with the condition `condC` has a nested element. According to the branch collapsing rule, previous empty branches need to be converted to one-line node, while following empty branches to the last branch with an element can be collapsed completly, therefore resulting in Listing 3.25, where the if- and the first elseif-branch nodes(lines 2-5) are changed to one-line nodes and the else-branch(line 9-10) node is collapsed completly.

There is a special case, such as the one shown in Listing 2.2, where the template uses built-in *null* checking. In this case the template already makes sure that the variable `name` is only used when not *null*.

Given `name` is available in the data model the condition evaluates to true and the placeholder always has a value and never needs to be requested from the user. Given `name` is not available in the data model the condition evaluates to false and the placeholder never has a value and never needs to be requested from the user, because the branch is ignored. So no matter the actual evaluation result, the placeholder is guaranteed to never be requested, thus we can ignore `name` and the conditional statement, for this specific case and collapse it.

As mentioned, conditionals create a context where variables and other directives are only allowed to be shifted upwards under one condition. This is only if the same element is used within every branch of a conditional node. So the actual branch on evaluation does not matter, because the placeholder is needed in any case. If the upshifting leads to empty branches, they need to be collapsed following previous branch-collapsing rules. This full collapsing is shown in Figure 3.3e and f, where lines 3-10 of Figure 3.3e are reduced to line 3 of Figure 3.3f.

```
<scalar key="boolA"/>
<scalar key="boolB"/>
<scalar key="boolC"/>
<conditional>
    <if condition="boolA">
        <scalar key="a"/>
    </if>
    <elseif condition="boolB">
        <scalar key="b"/>
    </elseif>
    <elseif condition="boolC">
    </elseif>
    <else>
    </else>
</conditional>
```

(a) Pre Branch-Collapsing (*Else* empty)

```
<scalar key="boolA"/>
<scalar key="boolB"/>
<scalar key="boolC"/>
<conditional>
    <if condition="boolA">
        <scalar key="a"/>
    </if>
    <elseif condition="boolB">
        <scalar key="b"/>
    </elseif>
</conditional>
```

(b) Post Branch-Collapsing (*Else* empty)

```
<scalar key="boolA"/>
<scalar key="boolB"/>
<conditional>
    <if condition="boolA">
    </if>
    <elseif condition="boolB">
    </elseif>
    <else>
        <scalar key="x"/>
    </else>
</conditional>
```

(c) Pre Branch-Collapsing (*Else* not empty)

```
<scalar key="boolA"/>
<scalar key="boolB"/>
<conditional>
    <if condition="boolA"/>
    <elseif condition="boolB"/>
    <else>
        <scalar key="x"/>
    </else>
</conditional>
```

(d) Post Branch-Collapsing (*Else* not empty)

```
<scalar key="boolA"/>
<scalar key="boolB"/>
<conditional>
    <if condition="boolA&boolB">
        <scalar key="x"/>
    </if>
    <else>
        <scalar key="x"/>
    </else>
</conditional>
```

(e) Pre Branch-Collapsing (Full Collapse)

```
<scalar key="boolA"/>
<scalar key="boolB"/>
<scalar key="x"/>
```

(f) Post Branch-Collapsing (Full Collapse)

Figure 3.3: Branch-Collapsing Examples

3.1.5 Functions and Macros

Functions and Macros are directives, that allow to fix redundancies in a template. They work similar with the difference that functions can return a value and that object code within them is ignored and not end up in the output, whereas macros do the opposite. Both have an identifying name and allow input parameters, that define a new namespace. Still, global variables can be accessed within, if there is no need for an abstract use.

Functions

Listing 3.26 show the usage of Functions in Freemarker. Allowing any number of parameters and multiple return statements anywhere within the nesting. If there is no `<#return ...>` statement, then an undefined value is returned. As mentioned, object language, such as line 3, will be ignored. Functions can be called in a template with `${FunctionName(Parameter1, ..., ParameterN)}`, where the `FunctionName` equals the name of the function definition, i.e., in line 1 of Listing 3.26.

```
1 <#function FunctionName Parameter1 ... ParameterN>
2     ...
3     This text will not be printed out
4     ...
5     <#return ReturnValue>
6     ...
7 </#function>
```

Listing 3.26: FreeMarker Functions

Functions can contain any type of directives, such as lists, conditionals, or variables from the global namespace. Therefore, it is not sufficient enough to only cover the signature, which means that the full functions content needs to be mapped to a TMM.

Macros

The Freemarker signature for macros is almost the same as for functions. Also, there can be any number of return statements, but with the difference, that they can't return a value and only work as a breakpoint, as the example in Listing 3.27 shows. `<#return>` will end the flow of the content of this macro, therefore will not print line 5. Object code can be used and will be printed to output.

```
1 <#macro printError errorcode message>
2     Error ${errorcode}:
3        ${message}
4     <#return>
5     Will not be printed.
6 </#macro>
```

Listing 3.27: Freemarker Macro Example

This macro can be called with `<@printError errorcode message/>`, which differs from the function call. Freemarker Macros also allow a nested subdirective within the macro, which we will not go into detail, because it does not introduce a new problem for us, so it will still be transformable with the presented rules.

Macros also support default values, either in definition or call, shown in Listing 3.28. So if neither `a` or `b` have a value in the data model, the template fragment would still evaluate, resulting in `1 + 10`. This allows us in theory to ignore those variables in the TMM transformation. However, we did not focus enough on the consequences, this decision may have, which is why we ignore defined default values of macros for now.

```
1 <#macro test x=1 y>
2     ${x} + ${y}
3 </#macro>
4
5 <@test a b=10/>
```

Listing 3.28: Freemarker Macro Default Values

Usage in Template Meta Model

Because they are similar, and their distinctive behavior is not relevant for us, we might model functions and macros the same way. Still, we require the distinction between function and macro, simply because they might have the same identifying name. Also, we gave an outlook, that a configuration can define which directives are allowed, therefore adopting the distinction from the template is necessary. Velocity, for example, only supports the macro directives but does not feature functions. There are two approaches to model this. Either, it can be orientated at the original template, giving the function or macro a defined space and whenever it is called within the template, we need a node that references the function node. Or else, the mapped meta code inside the function can be directly integrated where it would be called otherwise. This implies, that there is no

separate namespace dependent on the parameters. Both offer advantages over the other. The second option allows it, to use *reduction transformation rules* on the content of the function when inserted, which is not possible with the first approach. However, when a function is used multiple times in a template, this can result in more lines than in a separate non-redundant space. Mainly, it leads to a more complex transformation, that require many iterations of the *reduction rules*, which is an error-prone task for manual creation, conflicting with *ReqF2*. Therefore, we will use the first approach, outsourcing the function, like it is done in the template itself, exemplary shown in Listing 3.29. Here `ARG1` and `ARG2` are scalar variables defined in the local function scope, while `GLOBAL_VAR` is defined in the global scope.

```
1 <function key=FUNCTION_NAME args="ARG1, ARG2 ...">
2     <scalar key=ARG1/>
3     <map key=ARG2>
4         ...
5     </map>
6     <scalar key=GLOBAL_VAR/>
7 </function>
```

Listing 3.29: Function Representation in a TMM

It's necessary that each argument, separated by a comma, needs to match exactly one key of a nested directive (scalar, map, or sequence). If there are no arguments for a function, the `args` attribute can be spared. The call in the TMM will be done with `< function key=FUNCTION_NAME args="ARG1_ACTUAL, ARG2_ACTUAL"/>` The `args` names do not need to match between call and function, only the `FUNCTION_NAME` needs to match uniquely.

For Macros, the only difference is that `function` will be replaced by `macro` in the definition- and the calling-node.

Reduction transformation rules can only be applied within the context of the function. This can lead to redundancies, that need to be cleared when using the TMM in the further process, which is when a global variable used in the function as well as in the context of the call. An example is shown in Listing 3.30. The scalar with the key `GLOBAL_VAR` would be a duplicate in the context of the function call `<function key=FUNCTION_NAME/>`, therefore needs to be sorted out when processing the TMM.

```
1 <tmm ...>
2     <scalar key=GLOBAL_VAR/>
3     <function key=FUNCTION_NAME/>
```

```
4
5     <function key=FUNCTION_NAME>
6         <scalar key=GLOBAL_VAR/>
7     </function>
8 </tmm>
```

Listing 3.30: TMM Function with redundant Variable in context

Because functions or macros hold a unique name inside a template and are available everywhere inside of it, they should be nested directly inside the `tmm` node.

3.1.6 Include and Import

Include and *Import* are the last directives we will cover in the TMM.

The *include* directive takes a relative or absolute path and allows inserting another template at the point where the directive is used. Therefore, allowing outsourcing of redundant template fragments to different files. The Freemarker syntax for that is `<# include path>`. An included sub-template uses the scope of where it is inserted, all variables used in the sub-template have to be available in the source template.

The *import* directive is used to import macros and functions outsourced to a different file, which then can be used as a library in multiple templates. Different to the *include* directive, it does not matter as much where it is placed in the template, but preferably as early as possible, because functions can only be used after import. It is used in Freemarker as `<#import path as lib>`, where `path` is used same as with `include` and `lib` is defining the namespace under which the *macros* and *functions* can be accessed. Given the macro `printError` of Listing 3.27 is defined in `lib`, it can be called in Freemarker by `<@lib. printError errorcode message"/>`. Imported libraries work with the same namespace as macros or functions defined in the actual template, therefore using local namespaces defined by the parameters and the global namespace.

Both allow the outsourcing of redundant structures, to different files, so they can be used in multiple template, with the difference that imports allow parameterized used, while includes are static. For transforming them into the TMM, it is not as much of a problem how to model them, because we can use everything we already introduced. Sub-templates are not different in their structure than normal templates, therefore we can model their content to a separate tmm node. The reference in a template to a subtemplate, could be

done with `<include path=SUB_TEMPLATE_PATH/>`. Multiple calls to the same subtemplate, should not result in duplicate TMMs. Listing 3.31 shows a `tmmConfiguration` with two `tmm` nodes. The first tmm node(line 3-4) references the second tmm node(line 5-7) with `<include...>`. Thus, when the first TMM is pre-processed, the second one needs to be pre-processed as well.

```
1 <tmmConfiguration ...>
2     <tmm path=TEMPLATE_PATH templateEngine="FreeMarker
      ">
3         <include path=SUB_TEMPLATE_PATH/>
4     </tmm>
5     <tmm path=SUB_TEMPLATE_PATH templateEngine="
      FreeMarker">
6         <scalar key="test"/>
7     </tmm>
8 </tmm>
```

Listing 3.31: Referencing a Subtemplate in a TMM

Handling *imports* is not as much of a problem either. Declaring macros or functions global is done by nesting them not inside the tmm node, such as in Listing 3.30, but instead directly inside the tmmConfiguration node, as shown in Listing 3.32. The scalar `GLOBAL_VAR` is modelled three times, in the function context, the macro context and the TMM context. We defined function and macro contexts as an absolute boundary, meaning elements aren't allowed to up-shift outside of them, in contrast to conditional nodes, where *up-shifing* is allowed, if every branch includes the same element.

```
1 <tmmConfiguration ...>
2     <tmm path=TEMPLATE_PATH templateEngine="FreeMarker
      ">
3         <scalar key=GLOBAL_VAR/>
4         <function key=FUNCTION_NAME/>
5         <macro key=MACRO_NAME/>
6     </tmm>
7     <function key=FUNCTION_NAME>
8         <scalar key=GLOBAL_VAR/>
9     </function>
10    <macro key=MACRO_NAME>
11        <scalar key=GLOBAL_VAR/>
12    </macro>
13 </tmmConfiguration>
```

Listing 3.32: TMM Representation of Imports

Comparing this with Velocity, some differences can be noticed. It, too, supports an *include* directive, but with the difference, that this does not evaluate the file specified, but simply inserts it as it is, which is primarily used to insert text files, pictures, et cetera. To insert other templates that are evaluated, Velocity offers the *parse* directive. Macros can be defined within a template, but also in a global macro environment, which is defined in Velocity configuration and is always loaded. Macros defined in a template used can be treated as previously described, while global macros need a special transformation rule. In the mapping phase, all global macros should be loaded into memory and when they are used anywhere within the target template network, they need to be mapped as a macro element nested in the top-level tmm node. Therefore, guaranteeing that all required macros are available in the TMM.

3.1.7 Appliance of Mapping and Reduction Transformation Rules

Now that we described both Mapping and Reduction Transformation Rules, the only aspect we did not cover is in which order they have to be applied. A prioritization is important to guarantee an unambiguous result.

Generally, templates will be parsed and mapped from top to bottom. However, the meta code and its defined context need to be mapped as a whole. To demonstrate the transformation, we will use the example FreeMarker template in Listing 3.33, which references a `header.ftl`, shown in Listing 3.34 and imports `macros.ftl`, shown in Listing 3.35. The Template generates a simple Java class with getter and setter methods for a dynamical amount of `fields`. The `header.ftl` file includes a header description for a class, that can be customized, given `description` is not *null*, otherwise inserts a generic description.

```
1 <#include "header.ftl">
2 <#import "macros.ftl" as macros>
3
4 class ${name} {
5     ${vis} ${name}() {}
6
7     <#list fields as field>
8         private ${field.type} ${field.name};
9         <@macros.getterAndSetter field/>
10    </#list>
11 }
```

Listing 3.33: Advanced Template Example

```
1 <#if description??>
2   /**
3      ${description}
4     */
5 <#else>
6     /**
7     This class represents a ${name}.
8     */
9 </#if>
```

Listing 3.34: Content of header.ftl

```
1 #macro getterAndSetter field>
2 public ${field.type} ${"get" + field.name?cap_first}()
       {
3     return ${field.name};
4     <#if log>
5     System.out.println("get" + ${"\""+ field.name
      +"\""}+"() is called.");
6     </#if>
7 }
8
9 public void ${"set" + field.name?cap_first}(${field.
      type} ${field.name}) {
10    this.${field.name} = ${field.name};
11    <#if log>
12    System.out.println("set" + ${"\""+ field.name +
      "\""} + "("+ ${field.name} +") is called.");
13    </#if>
14 }
15 </#macro>
```

Listing 3.35: Content of macros.ftl

We can enumerate all the use of meta language, ignoring the content of directives for now, such as the `list`, only interested in the area it encloses, in this case line 7 to 10 in Listing 3.33 and ignoring duplicates on the same context level, such as the duplicate use of `name` in line 4 and 5. This enumeration is shown in Table 3.1, listing five identified uses of meta language. The next step will be to check, if each element can be expanded. M1 references another template, therefore we will investigate this next. M1.X references an element in the context of M1, followed by M1.X.X referencing an element in the context of M1.X and so forth. Before M2 is transformed M1 has to be fully transformed, therefore requiring for every element X in M1.X to be transformed before. Simple variable

references such as M3 cannot be expanded, therefore continuing with the next element in the same context, which in the example is M4. Table 3.2 already includes all fully expandend elements of the `header.ftl` file. Note that we defined M1.1a and M1.1b, to express that M1.1 is to be mapped as whole but creates two seperate contexts for each branch.

#	meta code	area of definition
M1	`<#include "header.ftl">`	line 1
M2	`<#import "macros.ftl" as macros>`	line 2
M3	`${name}`	line 4
M4	`${vis}`	line 5
M5	`<#list fields as field>...</#list>`	line 7-10

Table 3.1: Identifying Top-Level context

#	meta code	area of context
M1.1a	`<#if description??>...<#else>`	3.34 line 1-5
M1.1a.1	`${description}`	3.34 line 3
M1.1b	`<#else>...</#if>`	3.34 line 5-9
M1.1b.1	`${name}`	3.34 line 7

Table 3.2: Identifying expanded include context

M2 is the import of a macro library, we do not need to expand, because unless an actual macro is called from this library, we do not need to map any macro. For the mapping we only need to consider the scope and file specified. As mentioned, M3 and M4 can be skipped aswell, because they cannot be expanded, therefore only requiring M5 to be expanded. Note that `field.type` and `field.name` are listed multiple times, however always in a different context. Those duplicates have to be removed by transformation. After full expansion, the order in those tables defines the order in which those elements are transformed, therefore always preferring nested elements over neighboring elements. So the order starts with $M1 \rightarrow M1.1a \rightarrow M1.1a.1 \rightarrow M1.1b \cdots \rightarrow M5.3.4.2$.

Now that we defined the order in which templates need to be transformed, we need to define the order in which *transformation rules* are applied. Generally speaking, *Mapping Transformation Rules* are always applied first on an element. *Reduction Transformation Rules* however, are neither applied after mapping all elements, nor directly after mapping each element, but whenever a context is left. For example, see the result of mapping M1.1a.1 in Listing 3.36. M1.1 was mapped beforehand, therefore already creating the if and else branch node in the TMM. The mapping of M1.1b.1 however was not done yet.

#	meta code	area of context
M5.1	`${field.type}`	line 8
M5.2	`${field.name}`	line 8
M5.3	`<@macros.getterAndSetter field/>`	line 9, ref. 3.35 line 1-15
M5.3.1	`${field.type}`	line 2
M5.3.2	`${field.name}`	line 2
M5.3.3	`<#if log>...</#if>`	line 4-6
M5.3.3.1	`${field.type}`	line 5
M5.3.3.2	`${field.name}`	line 5
M5.3.4	`<#if log>...</#if>`	line 11-13
M5.3.4.1	`${field.type}`	line 12
M5.3.4.2	`${field.name}`	line 12

Table 3.3: Identifying expanded list context

```
1 ...
2 <conditional>
3     <if condition="description??">
4         <scalar key="description">
5     </if>
6     <else>
7     </else>
8 </conditional>
9 ...
```

Listing 3.36: Mapping M1.1a.1

Note, that there is no `<scalar key="description">` outside the conditional node, because the condition uses the special value checking syntax described in Section 2.4, which is the only case where no scalar has to be created for the condition, as mentioned in Section 3.1.4. Because there is no such element identified as M1.1a.1.1 or M1.1a.2, meaning a nested or neighbor element, the workflow returns to the context of M1.1. Before mapping M1.1b.1 however, *Reduction Transformation Rules* have to be applied to M1.1a, if possible. As specified, this branch needs to be collapsed, resulting in `<if condition ="description??"/>`. Only then the mapping of M1.1b.1 is allowed, which result in `<scalar key="name"/>` inside the else branch. Reduction rules could be applied on the else branch after that, however in this case, none can be applied. In conditionals, we might apply branch collapsing reduction on a branch, as with the if branch. After mapping and reducing every branch separately, the whole conditional structure might apply reduction rules, for example removing empty branches, up-shifting elements available in

every branch or collapsing the full conditional structure, if no branch has any nested elements.

Following this order of Mapping and Reduction Transformation Rules, the example template of Listing 3.33 will result in Listing 3.37.

```
<tmmConfiguration version="1.0">
    <tmm file="..." templateEngine="FreeMarker">
        <include path="header.ftl"/>
        <scalar key="name"/>
        <scalar key="vis"/>
        <sequence key="fields" elementKey="field">
            <scalar key="type"/>
            <scalar key="name"/>
            <macro key="getterAndSetter" args="field"/>
        </sequence>
    </tmm>
    <tmm file="header.ftl" templateEngine="FreeMarker">
        <conditional>
            <if condition="description??"/>
            <else>
                <scalar key="name"/>
            </else>
        </conditional>
    </tmm>
    <macro key="getterAndSetter" args="field">
        <map key="field">
            <scalar key="type"/>
            <scalar key="name"/>
        </map>
        <scalar key="log"/>
    </macro>
</tmmConfiguration>
```

Listing 3.37: Result of Transformation

3.2 The Template Pre-Processor

Now that we defined an initial version of the TMM and the rules to create it, we can use it to achieve our actual goal, which is the comparison of the TMM and the input data, to find missing values and add them manually, before evaluation of the template. To achieve that, we introduce a new component, the Template Pre-Processor (TPP).

Following the Separation of Concerns (SoC)[8] principle, we need multiple subcomponents in the TPP. One for the comparison, one for the user interaction handling the requests of missing values, and one for the enrichment of the original data model.

To achieve *ReqF6* and *ReqNF3*, those subcomponents should be decoupled from *Template Engine* specifics. We can use small interfaces instead, following the Interface Segregation Principle (ISP)[11], that can be adapted to access the *Template Engine* specific behaviors, such as the data model access or the engine's configuration. As declared in *ReqF9*, it should only be possible to add data to or query data from the data-model, not remove or reorganize existing data, the process can work with a facade for each *Template Engine*, which combines controlled access to each adapter. Figure 3.4 shows an exemplary combination of adapter and facade patterns, with implementations for FreeMarker.

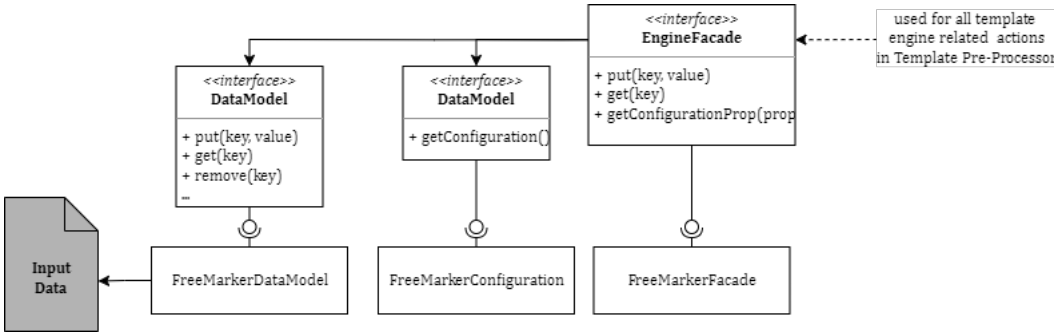

Figure 3.4: Template Engine specifics accessible over Facade

From this concept, we can add the pre-processing component to the basic model of Figure 1.1. As shown in Figure 3.5 this component takes the input data and all TMMs of the templates, that should be evaluated and creates an updated version of the input data structure which is given instead of the base input data to the Template Evaluator. Ideally, the *Enriched Input Data* should include all data necessary to deny the generator from failing, because a placeholder can't be replaced. Though, this will not ensure that the generation with any *Template Evaluator* will never interrupt, because there are can be other reasons in which a *Template Engine* can interrupt the generation. The Input Data artifact here refers to the abstract data model, accessed through the *Template Engine* facade of Figure 3.4. The TMMs have to be available at the start of the process. Because they have to be created manually for now, they need to be available before starting the application.

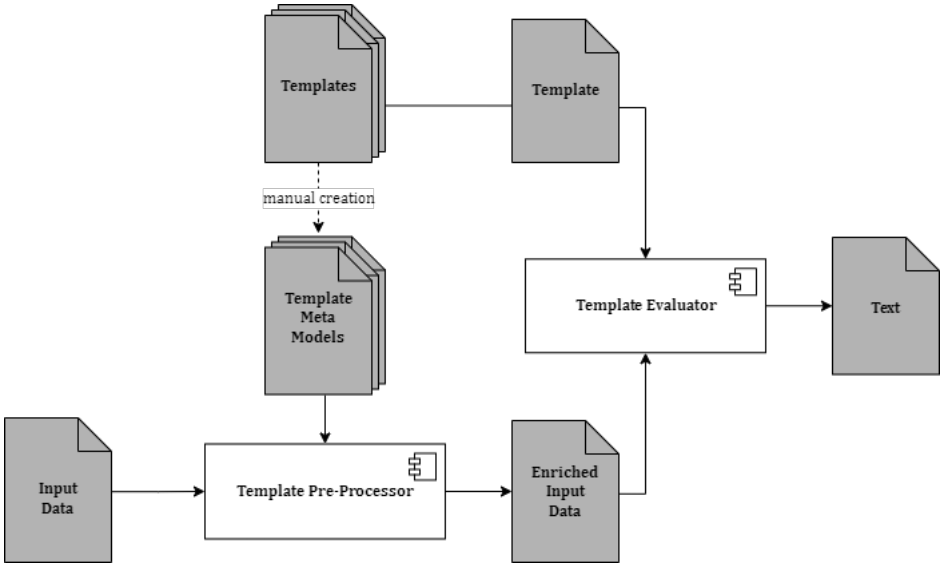

Figure 3.5: Code Generator with Template Pre-Processor

In White box view (see Figure 3.6) the TPP includes multiple subcomponents. The figure shows the flow of data in the process. The core components are the *Model Comparator* and the *Data Enricher*.

3.2.1 The Model Comparator

The Model Comparator takes the TMM and the input data model as input and compares them to find missing values. The TMM defines the required values, the data model defines the given values of the current instantiation, and the model comparator matches them, resulting in a collection of mismatches, if any are found.

The comparison searches for the keys of the TMM in the data model top-down. Because of up-shifting, those keys can be found on the same level, given that there are no conditionals. With conditionals, there can be boundaries on certain levels, in which case the key in the TMM is on a lower level than in the data model. Because the data model only has key-value pairs on lower levels if they are dependent on a different key and there are no other context defining structures such as conditionals, it is safe to say that the level of a key in the data model is always equal or higher than the level of the same key in the TMM.

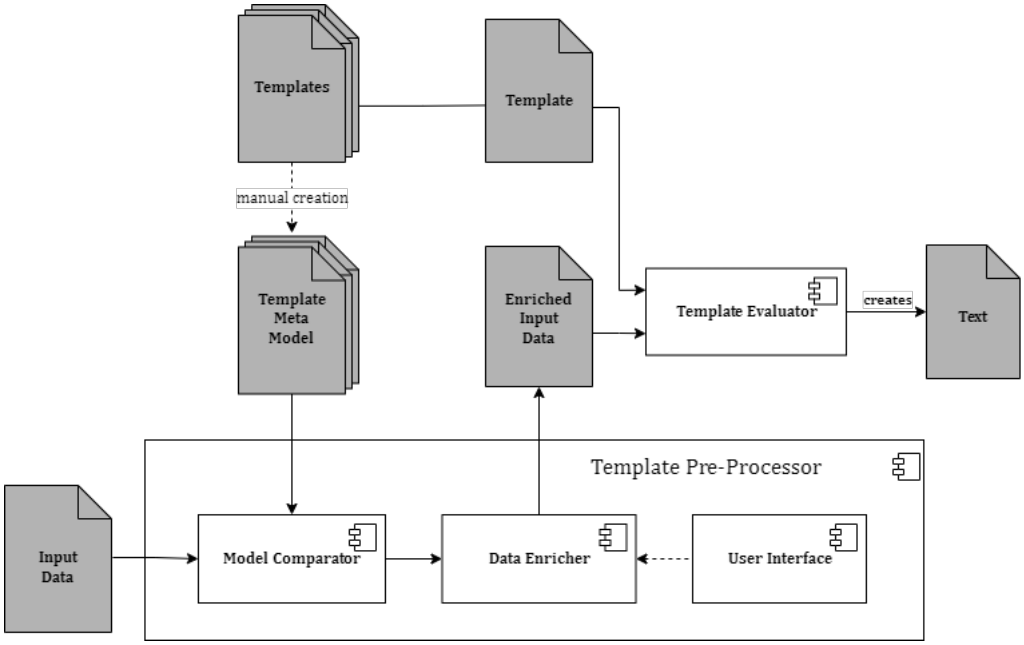

Figure 3.6: Code Generator with Template Pre-Processor(Whitebox)

In the problem statement, we listed an example template generation with an example data model (Listing 1.1, Listing 1.2 & Listing 1.3). The according TMM is shown in Listing 3.38

```
1 <tmm file="..." templateEngine="FreeMarker">
2     <scalar key="name"/>
3     <scalar key="vis"/>
4     <scalar key="log"/>
5     <sequence key="fields" elementKey="field">
6         <scalar key="type"/>
7         <scalar key="name"/>
8     </sequence>
9 </tmm>
```

Listing 3.38: TMM example

There is no conditional node, due to *branch-collapsing* all that remains is the condition `log` as a scalar node. Because of that, the two models are very similar in terms of their key placement. The comparison is relatively straight-forward. The order is as listed in the TMM, $name \rightarrow vis \rightarrow log \rightarrow fields$.

Because `name` is on top level in the TMM, this exact key has to be looked up in the data model. The same works for the next two scalars. `fields` is supposed to be a sequence

however, where each element needs to follow the same pattern, in this case the scalars `type` and `name`. To compare sequence, first the key `fields` will be looked up in the data model. It contains two elements. Each element is compared to the pattern. Both elements match the pattern, therefore the check is successful, and no value is missing for generation.

When the Model Comparator finds missing data, it is important to store the essential information for the later enrichment. This includes the full path along the data model to the position where the value is missing, as well as the type of value missing. The full path is needed mainly to remember the sequence index, where there is a missing value. Though, it is unlikely that a value is missing only in one element. Currently, there is no complex type system implemented, we still can differentiate between the abstract data types sequences, maps, or scalars. Maps or sequences should always be requested as a whole. Therefore, only the highest missing element structure should be in missing values set D_M. Given a sequence named `fields` that includes a map with the keys `field.type` and `field.name`. When `fields` is missing, then only `fields` should be in the set of missing values D_M, but not transitive objects such as `field` or `field.type` and `field.name`, because they are missing anyway.

3.2.2 The Data Enricher

The Data Enricher is the component with the responsibility to request all items of the missing value collection from the user and put those retroactively added values to the data model at the designated spot.

The actual requests should be outsourced to a user interface component. Offering an interface, i.e., *MissingValueRequestInterface*, enables multiple implementations for user interactions. Requesting missing variables is not much of an issue, because they represent a scalar, which is easy to parse from manual input, and can be requested and added to the model one after one. Maps and sequences however are a bit more complex, because as mentioned, they have to be requested as a whole. The complexity is even higher the deeper they are nested.

Listing 3.39 shows an Example TMM with multiple nested maps and sequences. Given that the input data model, does not include the map *zoo*, the whole structure has to be requested from the user. Enriching the data model with the map *zoo* is only possible after requesting all sub elements. Figure 3.7 shows the order of requests and creation of

the full data structure. The scalar with the key `name` can be requested directly, while the sequence with the key `animals` first needs to request nested elements, such as `name` and `description` in the scope of animals, as well as another nested sequence `inhabitants`, which needs to follow the same steps, therefore request `name` and `age` in the scope of inhabitants. After all scalars of a local scope have been requested and added to the respective container, it can return and continue with the next element. Only at the last step, when *zoo* is fully constructed, it can be added to the data model. Note, that this integrates the option to request multiple items for newly created sequences, as defined in *ReqNF1*, by using loops, that should ask the user on every iteration if he wants to add another item. As shown, only scalars need to be requested, which then can be organized into the right container type.

```
1 <map key="zoo">
2     <scalar key="name"/>
3     <sequence key="animals" elementKey="animal">
4         <scalar key="name"/>
5         <scalar key="description"/>
6         <sequence key="inhabitants" elementKey="
      inhabitant">
7             <scalar key="name"/>
8             <scalar key="age"/>
9         </sequence>
10    </sequence>
11 </map>
```

Listing 3.39: TMM Zoo Example

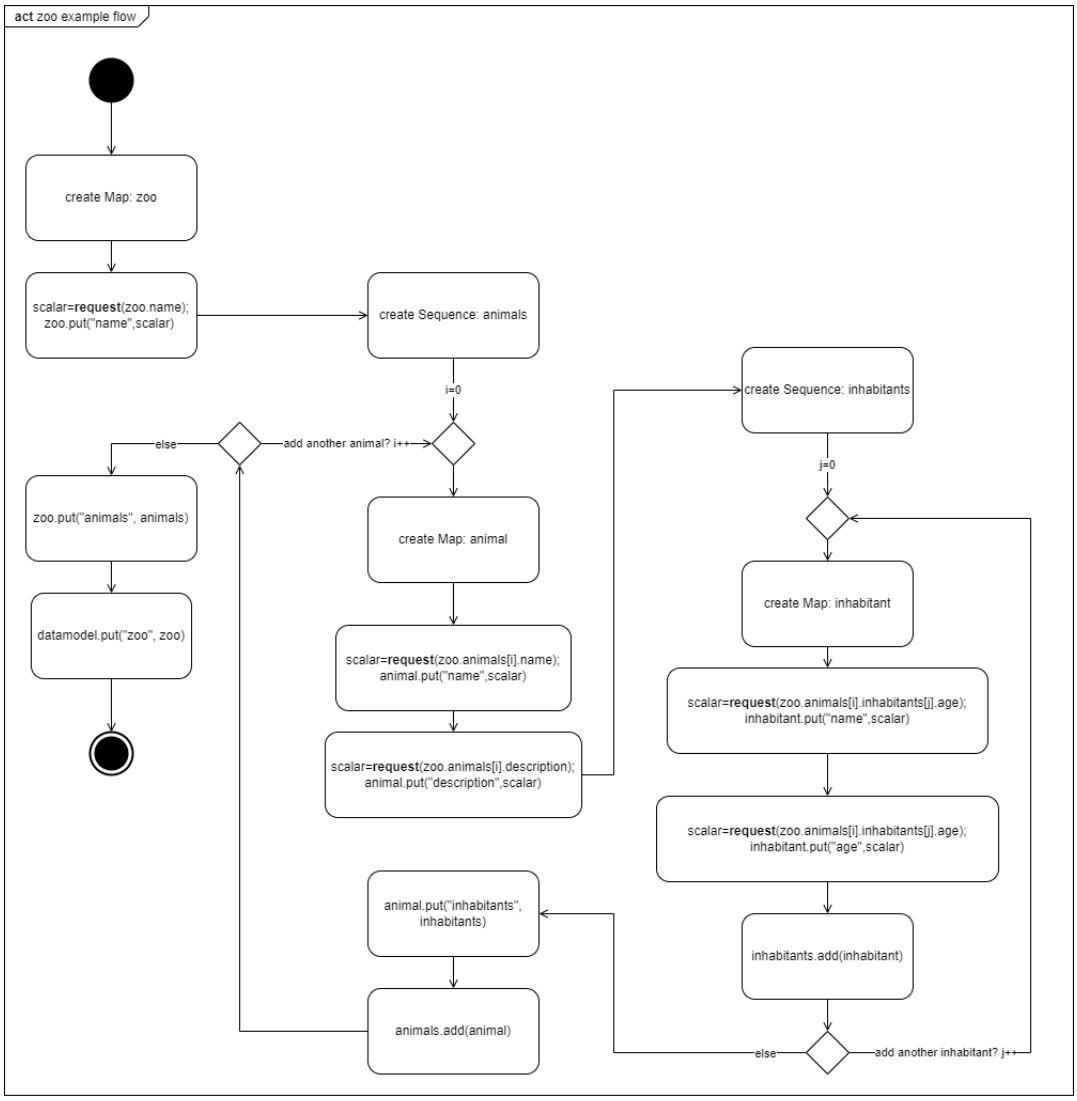

Figure 3.7: Zoo example flow

4 Template Pre-Processing in Advanced Code Generation

In this chapter, we will discuss the use of pre-processing in an advanced and industrial context, with focus on the systems, previously presented in Chapter 2. We are motivated to find problems or difficulties that might occur in practical application of our approach.

4.1 Pre-processing with CobiGen

In Section 2.3, we presented and explained the important terms and features of CobiGen. These need to be addressed and possibly adapted, so we will discuss the extensibility of the CobiGen framework, the two configurations and the whole generation process itself. The main question we try to answer is, if pre-processing is achievable in CobiGen despite dynamic template selection.

The most important issue, originates from the architecture of CobiGen (see Figure 2.2) and its workflow. CobiGen runs as a mostly isolated process, which means after selecting a compilation unit, the process runs from reading the input and creating an internal model, over evaluating templates, to merging the generated code to the source code base, as one process, with the only interruption being the user prompt to select the increments to generate. Figure 4.1 shows an example of the CobiGen User Interface from the Eclipse IDE integration. On the left there are all possible use-cases, which can be generated based on the input, listed on root level (i.e., CRUD devon4ng Angular App), which can hold nested increments (i.e., Angular devon4ng Component). Based on the selection, the files that will be generated are shown on the right side. If the file has no conflicting file in the source code base, they will be marked as *(new)*.

The problem is, that we can't just build the TPP to run before the CobiGen process, because it requires a data model, which is only created by the Input Reader, therefore

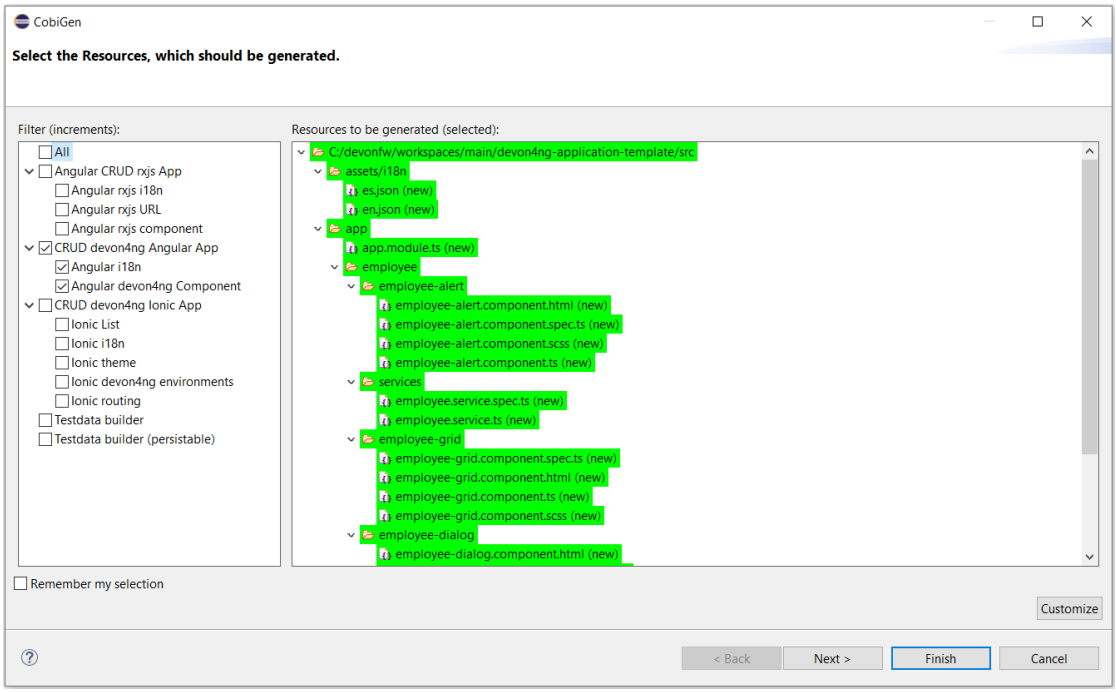

Figure 4.1: CobiGen User Interface

the TPP needs to run at least after the Input Reader. Furthermore, we require the target templates, not directly, but as a list of template paths, so it is clear which TMMs need to be checked. However, the required templates are only selected by the user, therefore the TPP needs to be set after user selection and before template evaluation. There are two ways, how the TPP might be integrated into CobiGen.

The first idea is to give the responsibility to CobiGen, which means pre-processing needs to be actively integrated into the Framework. This means, the TPP is added to the CobiGen Framework (Figure 2.2), as shown in Figure 4.2.

The second possibility is to add the TPP externally as an extension to CobiGen, therefore not relying on a CobiGen integration and shifting the responsibility to the Framework consumer instead. This can be done through the *Template Engine* extension point. For example, instead of using the *FreeMarker Template Engine*, one could implement a *FreeMarker Pre-Processing Template Engine*, which can utilize the existing *Template Engine* implementation but with upstream pre-processing, such as shown in Figure 4.3.

The second approach has the benefit of practicability. All components needed can be used as they are, without requiring many changes, while the first approach requires the

CobiGen framework to be changed a lot more internally. The first approach would require that each supported *Template Engine* had to be adapted by the TPP as well, or at least requires a setting to enable or disable pre-processing.

There are actually not many advantages the first approach has over the other, except that it could access information that is only used internally and that implementations, such as the facade of Figure 3.4 could be reused. The fact that CobiGen developers would be in debt to implement and adapt the TPP to work with its components, is a major inconvenience.

In contrast, the second option does not suffer from these issues. Firstly, the pre-processing can be used if needed by the consumer and is not fundamentally integrated into the process. CobiGen developers could still be involved and offer pre-processing *Template Engine* implementations for the currently implemented *Template Engines*. CobiGen values extensibility most, which ideology we can adopt. Using the extension point does not change the level of abstraction used within the CobiGen framework, but allows the consumer to add pre-processing when it is actually needed instead of offering it based only on the assumption that it may be used. Therefore, the second approach utilizing the *Template Engine* extension should be preferred.

In Section 2.3, we also presented context and template configurations and their usage within template directories. From this, we can derive, where to best place TMM definitions in a CobiGen directory. While it is of course possible to define one `tmm.xml` per template, a centralized solution, would be more comprehensible and would give a better overview over required data.

The context configuration defines which template root directories are generally generateable and defines triggers which work as filters, to define which use-cases are allowed in the user selection afterwards. Therefore, it might be possible to define one TMM configuration per use-case, directly in the root template directory on the same level as the template configuration file. This has the advantage, that only one file per template root directory includes all required data for the whole directory and outsourced subdirectories and allows a simple lookup. Additionally, function libraries and sub-templates are defined once in the same configuration as the TMM that uses them. Having multiple files, would mean that those libraries or outsourced sub-templates might be defined multiple times, if they are referenced in more than one TMM configuration. It is important to note that the context configuration can assign further variables, specific to the use-case, while the Input Reader defines variables that are generally available. We have to pay attention

to that distinction when creating the TMM. Foremost, variable assignments, which are defined as `<variableAssignment type="" key="" value="" />` are stored in the namespace `variables.<key>`, therefore requiring a special mapping. Those mappings can be based on the given `type` attribute. One might be a *constant*, which defines a fixed value. Those can actually be ignored the same as default values in templates, as described in Section 2.4, while others such as regex types need to be checked as well.

The second option can be derived from the template configurations. As mentioned, they define all increments that can be generated within a template root directory. Increments are the smallest fraction that can be chosen as a generation unit. By separating TMM configurations based on increments, one would not run into the problem that unnecessary TMM's are loaded into program memory. This means that instead of creating one TMM configuration per use-case, we create one for each increment, which can either be located on the same level as the template configuration file and then be named according to the increment, or be located within the according subdirectory of each increment with a generic name. However, this can lead to the described redundancies.

Most importantly, this approach allows going even one step further, in terms of increasing the usability of CobiGen. By arranging TMMs based on increments, it can be possible to pre-process all available increments instead, so that it can be displayed dynamically, i.e, in the user interface (Figure 4.1), how many and which variables are missing for successful generation of each increment. Pre-filtering can still be applied, for example, based on a percentage threshold of missing data that needs to be exceeded.

From this we can conclude that, given that the presented theoretical solutions can be implemented, pre-processing is achievable in CobiGen despite dynamic template selection.

4.2 Pre-processing with Repleo

Next, we will discuss the application of pre-processing for Repleo. As we introduced in Section 2.1, Repleo is a system to build syntax-safe template generators. First, we can determine the major difference between CobiGen and Repleo.

While CobiGen is a full Framework, which implements the entire process from creating an input model to generating and merging the output to the source base, allowing multiple *Template Engines*, Repleo is only one *Template Engine* implementation. It expects a

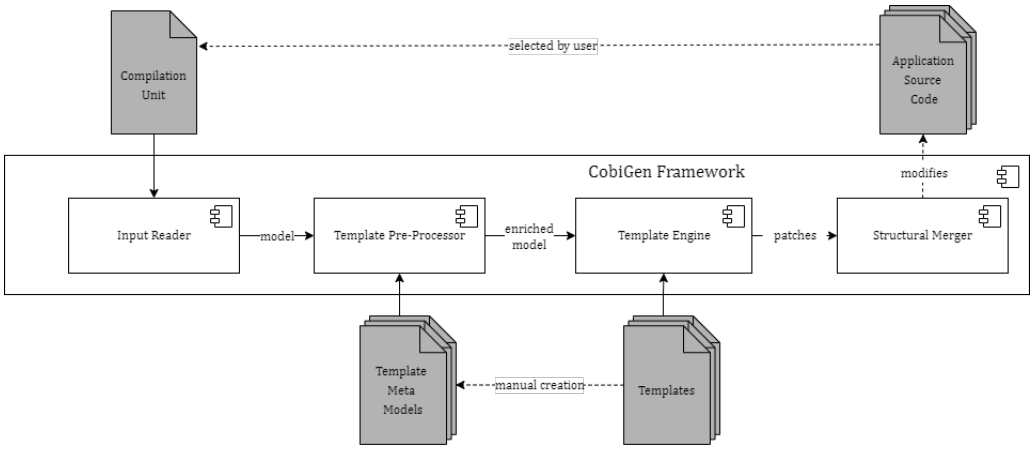

Figure 4.2: CobiGen with integrated Template Pre-Processing

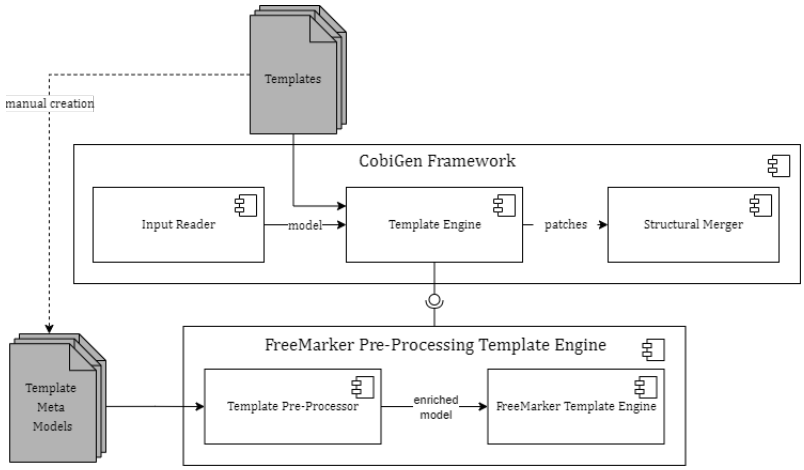

Figure 4.3: CobiGen extension with Template Pre-Processing

created input model, and only outputs the generated file, which needs to be managed by the consumer. Therefore, we do not have the same problems as with CobiGen and can extend Repleo with the TPP relatively straightforward. Figure 4.4 shows the interaction between the TPP and the Repleo process shown in Figure 2.1. To achieve this, it only requires few adaptations that are similar to any other *Template Engine*. We need to implement the data model facade (Figure 3.4) for Repleo to handle the access, and specify the abstract transformation rules for the Repleo template language to a TMM. For example, Repleo uses '/' instead of '.' for nested namespaces.

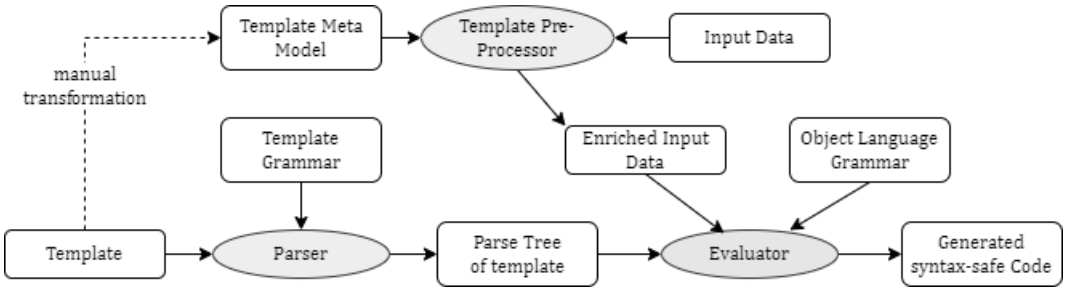

Figure 4.4: Repleo with Template Pre-Processing

The main issue, when applying pre-processing on Repleo originates from the aspect of syntax-safety. The manual input of missing values might lead to a syntax error in the generated output, therefore an exception will be thrown by the Evaluator, terminating the generation. The TPP has no knowledge over the object language grammar and can't guarantee that every manual input does not conflict with the object syntax. However, this is not a problem regarding the soundness of the *Template Engine*, but rather a usability issue. Syntax errors will still be detected when evaluating, including those caused by manual input, and then terminate the generation.

Instead of creating the TMM from a template, one could try to transform it from the parse tree of the template. This has the advantage that placeholders are assigned a type based on the given syntax. This specific type system can then be used in the TPP for enrichment to guarantee that the manual input is in compliance with the output syntax and does not lead to an error in the evaluator.

However, according to *ReqF6* and therefore *ReqF4*, this data should not be used directly in the TMM. A TMM should be created only once per template, but the template could be used by multiple engines. For example, a template written in FreeMarker can be used both by the normal *FreeMarker Template Engine* and the *Syntax Safe Java FreeMarker Template Engine* (Section 2.3). This can be bypassed by outsourcing the type system

to a separate file, which can be as simple as just mapping the template variables to the types assigned by the template parser, such as `{"key": "type"}`. Where for each key in the TMM there is a key-type pair in this separate file. These files can then be uniquely used for Repleo pre-processing.

Though, it can't be used in the TPP directly, because the components also should work on a non-engine-specific level. The only extension point is offered by the data enricher, which we called *MissingValueRequestInterface*. This can be implemented to connect requested scalars with the defined type, based on their matching key. Figure 4.5 depicts, how this process can work around the TPP. The manual input can then try to be parsed to the required type and then be returned to the Data Enricher.

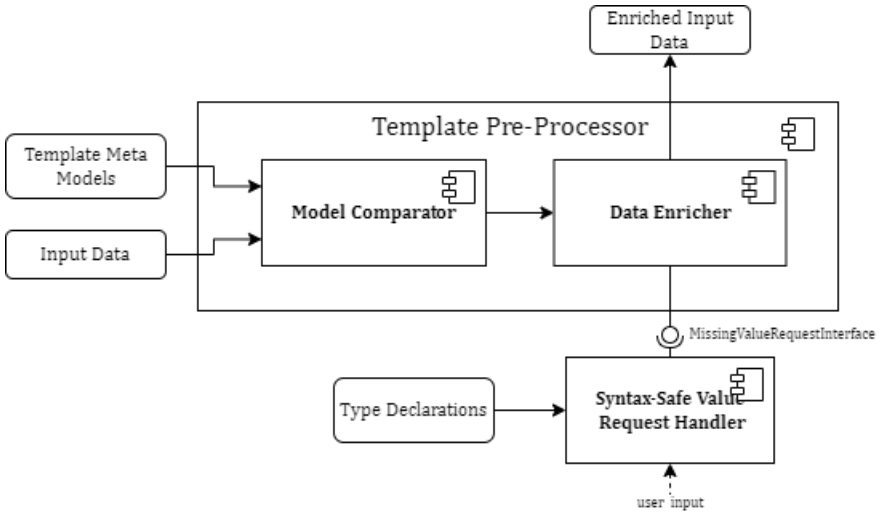

Figure 4.5: TPP with syntax-safe Input

There are some requirements that need to be fulfilled, to allow this second more complex approach to work. First, it needs access to the internally processed parse tree of a template. Because Repleo is mainly a theoretical work[3][4], it is not specified enough of how the engine can be configured. Nevertheless, it can be stated, that the transformation does not work manually, because the parse tree is internally processed, meaning it does not result in a file, that can be accessed by a human and therefore the TMM can't be manually created. Given that it can be automatically transformed, which will be addressed in Section 4.4, the last requirement is that the transformation from the parse tree results in the same TMM as created by the original template. Analogous to the

Equation (3.1), we define g as:

$$g\colon parseTree \xrightarrow[TR]{} tmm \tag{4.1}$$

We use the *parse* function defined by Arnoldus as h to describe the relation between template and parse tree as:

$$h\colon template \xrightarrow[parse]{} parseTree \tag{4.2}$$

Given Equation (3.1) and Equation (4.1) and $parseTree = h(template)$, we require Equation (4.3) to be valid for this approach to work. This means that the TMM transformed from a template and from its parse tree have to be equal to each other for pre-processing to be used from a parse tree.

$$f(template) = g(parseTree) \tag{4.3}$$

Given all these requirements can be fulfilled, the TPP integration can be changed from the non-syntax-safe input version, shown in Figure 4.4 to a process that guarantees syntax-safe manual user input, shown in Figure 4.6.

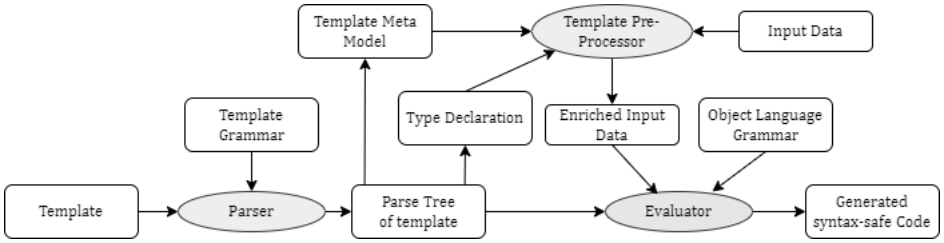

Figure 4.6: Repleo with syntax-safe TPP

4.3 Pre-processing with SafeGen

The integration of pre-processing into SafeGen has similar difficulties as with Repleo. We can choose an analogue approach to Repleo, either ignoring type-safety of manual input, or using a system as specified in the previous chapter, where the type system is transformed to a separate file and the combination of the type system and missing values is applied in a request interface implementation. However, the TMM has to be created

as originally defined through Equation (3.1), because in contrast to Repleo where types are assigned only in the parse tree, SafeGen defines types in the template directly.

The main problem and biggest difference to Repleo is the data with which the template is filled. SafeGen templates use only what is given to them as input in `#defgen` declaration, which might be a Java class, interface or method. The TMM only depicts abstract types: sequences, maps and scalars. Transforming a java class to a TMM might be possible, but it is far too complex to be covered in this thesis and to be applied for general use. For SafeGen it may be a better approach to create SafeGen specific TMM, which instead of using a combination of sequences, maps and scalars, uses the java structure of classes, methods, field, etc. and is built specific for that. SafeGen is specifically built for Java, creating templates based on Java syntax. Trying to forcefully apply our abstract approach to SafeGen is simply far from reality, because we did not collect requirements from SafeGen specifically and the fundamental usage of SafeGen and classic template-based code generators is too different.

We can conclude that our approach for template pre-processing on SafeGen is practically not applicable under its current specification. However, SafeGen specific pre-processing should be implementable using our approach as a basis. Although, it is a question of the actual necessity. SafeGen inputs are limited to legal Java programs and its application is also limited to specific cases, such as generating an interface, wrapper, or delegator from class. The differences to other *Template Engines* are simply too different for our approach to work.

4.4 Automatic Creation of the Template Meta Model

For the most part, we described transformation rules abstracted from any template language, which have to be specified for each template language. While some *Template Engines* aren't very complex and the transformation should be relatively straightforward, there are other languages or templates that can lead to complex transformation scenarios.

Especially the reduction rules are the biggest source for that. Complex templates with deep nesting, results in a complex transformation with multiple iterations of applying reductions, that can be too extensive and error-prone for a human doing manual transformation. Considering that one TMM for each template has to be created, CobiGen

for example has template root directories which include dozens of templates, resulting in much manual work.

Another big issue would be the maintenance of templates and their TMMs. If a template is changed, that will make the according TMM outdated, unless it is updated correctly as well.

Combining these to problems regarding complex template structures, such as CobiGen's, will prove to be unusable in industrial applications. This can be solved by automation. Currently, the pre-processing relies on two asynchronous separate steps: the creation of the TMMs and the actual pre-processing using the TMMs. For the TPP to work, it has to be ensured that the required TMMs are available at run-time. When the transformation is automated, this results in one synchronous process, where the TMMs can be created and processed in one continuous flow. In Figure 4.7 the manual creation is replaced by a new component, referred to as *Template Transformer*, which need to be specified for each *Template Engine* that needs to be supported. This will specify all transformation rules, and if implemented will result in the correct TMM under any input, no matter how complex. This can be optimized by caching the TMMs. This means that TMMs will be transformed on the first execution, and then reused in following executions. This, requires that certain conditions are fulfilled. Most importantly, the original template wasn't changed, which would require the TMM to be updated.

Therefore, we can conclude, that the automation of creating a TMM is essential for appliance in industrial applications, however leaving the exact specification for future work.

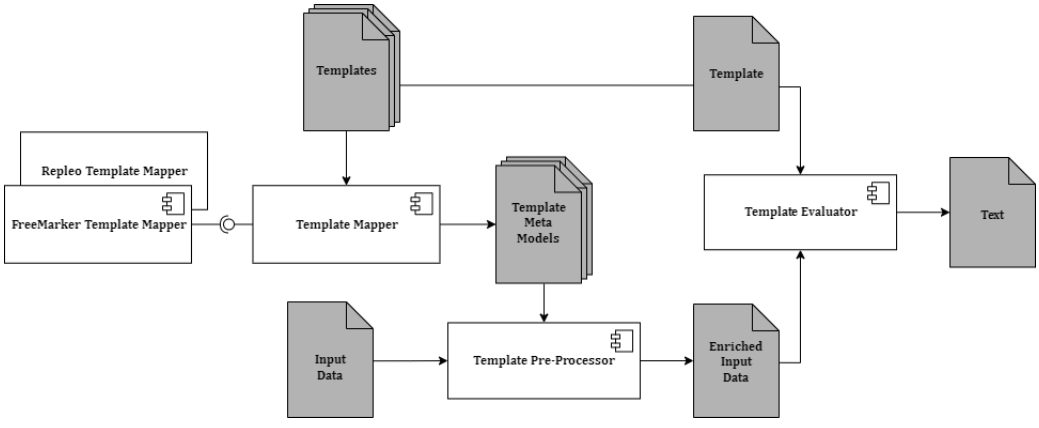

Figure 4.7: Automation of Template to TMM Transformation

5 Conclusion & Future Work

In conclusion, this thesis focused on creating a process that enables to find missing data in an input data model by comparing it to the required data of selected templates. In industrial applications, such as CobiGen, template-based source code generation is used in a fully automated process. Currently, if an exception is caused by missing input data which is required in a template, the process will terminate, given an interrupting *Template Engine* such as FreeMarker is used. While this guarantees that only desired output is produced, therefore guaranteeing soundness, the usefulness suffers from it. Instead of starting the generation on the off chance of success, we aim to check given and required data before evaluation time, to find causes for interruption.

We introduced the Template Meta Model, a representation of a templates meta language. It derives from a Template through *transformation rules*, which define abstract instructions, that need to be specified for a *Template Engine*. We described *transformation rules* for the most common template directives, such as conditionals and iteration. Manual TMM creation has to happen asynchronously before the TMM processing. We discussed the problems caused by manual creation regarding the industrial context, where many complex templates imply too much manual work and high error probability to be used. However, developing an automatic process for template transformation, will increase the usability by removing the repetitive work of creating and updating a TMM for each new or changed template. The TMM grants the ability to do a comparison of templates and the input data model, to collect missing values. These values can be requested from the user, and are then sorted into the original data model, therefore resulting in a data model which is guaranteed to include any data required for template evaluation, given that the original data model can be processed. By applying general software engineering principles and architecture patterns, we suggested an internal abstract design for the Template Pre-Processor component. Then, only an initial work is required to adapt the TPP for a specific *Template Engine* and template language in use.

While keeping the TPP and TMM mostly *Template Engine* independent and only requiring engine specifications of certain interfaces, therefore reducing the amount of adaptions needed for extending the process to different engines, we discovered that in context of syntax and type-safe *Template Engines*, we are not able to request the required object from the user only by the TMM data. Data models that include custom objects, for example Java classes, can't be processed with the current TMM version. Only a combination of sequences, maps and primitive types is allowed. However, we described a workaround for the syntax-safe *Template Engine* Repleo and our presented approach, by collecting type information in a separate file used in the enrichment process. While a similar process might work for the type-safe code generator SafeGen, we found a conflicting problem in its fundamental use. While most *Template Engines* work with a tree-like data model, which can be created from an input, such as through CobiGen's Input Reader and do not imply a fixed object language, SafeGen only works for Java and uses Java objects as input directly. Therefore, it would be more effective to let this information be used in a TMM as well. The SafeGen templates and input is simply too different to what our approach is designed for.

Another issue that we determined is how to handle meta language conditionals, while comparing. We included conditionals in the TMM to establish different contexts, based on how the conditions are evaluated. However, evaluating the condition before the actual template evaluation is a process that we did not focus on in detail. Unless a solution is specified on how to evaluate template language specific conditions in a *Template Engine* independent context, all possible cases have to be covered in comparison and enrichment. This means that there are cases, where values are requested from the user, that will not be used in template evaluation.

Lastly, it is important to note that at the time of writing, the TPP is only implemented as a prototype, lacking many features presented in this thesis.

Bibliography

[1] *What is Agile?*. – URL https://www.atlassian.com/agile#:~:text=Agile%20is%20an%20iterative%20approach,small%2C%20but%20consumable%2C%20increments.. – last accessed 28.09.2022

[2] Arnoldus, B.J.: *An illumination of the template enigma : software code generation with templates*, Mathematics and Computer Science, Dissertation, 2011

[3] Arnoldus, J. ; Brand, M. van den ; Serebrenik, A. ; Brunekreef, J.J.: *Code Generation with Templates*. Atlantis Press, 2012 (Atlantis Studies in Computing). – URL https://books.google.com/books?id=UvC0MJHSqjkC. – ISBN 9789491216565

[4] Arnoldus, Jeroen ; Bijpost, Jeanot ; Brand, Mark van den: Repleo: A Syntax-Safe Template Engine. In: *Proceedings of the 6th International Conference on Generative Programming and Component Engineering*. New York, NY, USA : Association for Computing Machinery, 2007 (GPCE '07), S. 25–32. – URL https://doi.org/10.1145/1289971.1289977. – ISBN 9781595938558

[5] Brunnlieb, Malte: *Source Code Transformation based on Architecture Implementation Patterns*, Technische Universität Kaiserslautern, Dissertation, 2018

[6] Brunnlieb, Malte: Codegenerierung mit CobiGen. In: *JavaMagazin* (2021). – URL https://www.capgemini.com/de-de/wp-content/uploads/sites/5/2021/03/JavaMagazin-03-CobiGen.pdf

[7] Brunnlieb, Malte ; Poetzsch-Heffter, Arnd: Architecture-driven incremental code generation for increased developer efficiency. In: Hasselbring, Wilhelm (Hrsg.) ; Ehmke, Nils C. (Hrsg.): *Software Engineering 2014*. Bonn : Gesellschaft für Informatik, 2014, S. 143–148

[8] Dijkstra, Edsger W.: *Selected writings on computing: a personal perspective*. New York, NY, USA : Springer-Verlag New York, Inc., 1982. – ISBN 0-387-90652-5

[9] Huang, Shan S. ; Zook, David ; Smaragdakis, Yannis: Statically Safe Program Generation with SafeGen. In: Glück, Robert (Hrsg.) ; Lowry, Michael (Hrsg.): *Generative Programming and Component Engineering*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2005, S. 309–326. – ISBN 978-3-540-31977-1

[10] Johner, Christian: Code Generation: The Magic Formula for Faster and Better Code? (2021). – URL https://www.johner-institute.com/articles/software-iec-62304/and-more/code-generation-the-magic-formula-for-faster-and-better-code/

[11] Martin, Robert C.: *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003. – URL http://dl.acm.org/citation.cfm?id=515230

[12] Renze, Matthew: Composite Data Types in Data Science. (2019). – URL https://matthewrenze.com/articles/composite-data-types-in-data-science/

[13] Renze, Matthew: Scalar Data Types in Data Science. (2019). – URL https://matthewrenze.com/articles/scalar-data-types-in-data-science/

[14] Shore, Jim: Fail Fast. (2004). – URL https://www.martinfowler.com/ieeeSoftware/failFast.pdf

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original