RWTH Aachen University
Software Engineering Group

Internationalization in Model-Driven Development using the MaCoCo Project as an Example

Bachelor Thesis

presented by
Franken, Miguel

1st Examiner: Prof. Dr. B. Rumpe

2nd Examiner: Prof. Dr. Horst Lichter

1st Advisor: Dr. Andreas Wortmann

2nd Advisor: Simon Varga

The present work was submitted to the Chair of Software Engineering

Aachen, April 6, 2022

Abstract

Nowadays, companies want to distribute their software on the global market with the goal of generating more revenue. However, there are many language and cultural differences that make successful global distribution difficult. For this reason, companies usually internationalize and localize their software systems in order to offer each market a customized version of their system that takes into account their linguistic and cultural differences. This paper shows how an internationalization of systems created in a model-driven software development process can be achieved. In particular, it is shown in detail how MontiGem and MontiGem-based applications can be internationalized. MontiGem is an enterprise information system developed at the RWTH Aachen University and serves as a basis for many other projects such as MaCoCo, a financial services software. Large parts of these systems are generated from models within a model-driven development process, be it the backend, the frontend or the communication between both. The results of this thesis enabled the localization of MontiGem-based applications, i.e. the generated systems can now be translated into a number of desired target languages. The user can manually select the language to be displayed in the system or the system derives the most appropriate language for a user from external factors such as the language of the web browser in which MontiGem is access by the users. This paper shows how the modeling language GUIDSL, which is used to generate large parts of the user interface of MontiGem and MontiGem-based applications, and the FrontendRTE have been extended to provide internationalization functionality. In complex and large projects like MaCoCo, collaboration and exchange of units to be translated also play an important role. This paper describes how these problems can be solved by using standardized translation formats that are also understood by most external translators.

Statement of Task

Nowadays, companies want to release their software on the global market with the goal of achieving higher sales. To successfully distribute a software on the global market, it must be adapted to the linguistic and cultural specifics of each market it is targeted at. In the literature you can find many use cases that report how to achieve successful software internationalization for software systems developed using traditional software methodologies. Besides traditional software methodologies where source code plays the central role in the artifact of software development, there are alternative methodologies. Among them is the model-driven engineering methodology, which can be seen as an alternative to the traditional software development process where models instead of source code play the central artifact of software development. Models abstract most of the technical aspects of a system and often describe a certain view of the system, so that domain experts can also participate as modelers in the creation of these models. From the models created, it is then possible to generate executable systems or subsystems by using software generators within the framework of model-driven software generation. These can be integrated into existing runtime engines and thus ultimately represent an executable system. This thesis should explore how an internationalization of a system that is developed within a model-driven engineering process can be performed. As a use case MontiGem and MontiGem-based applications like MaCoCo, a generated financial services application, should be internationalized. In the end, these systems should be able to be displayed in different languages and in the models from which the majority of these systems were generated, the modelers should be able to mark units that are translated or otherwise adapted to a locale selected by the user. Another goal of this work was to make this internationalization work so that it can be used in large and complex projects and that the adaptation of the generated system to a new language can be done in a few steps. In complex systems external translators are often employed who are not part of the actual development team of the application. The goal of this work was therefore to internationalize MontiGem and MontiGem-based applications in such a way that units to be translated could be extracted and stored in a standardized translation format that is understandable by most external translators. Even if the translation is done by the actual developers and/or modelers, different procedures for efficient collaboration should be developed. The system should be implemented with three different roles of people: Translators, reviewers and releasers. While the translators should translate the actual units to be translated, it should only be possible for the unit to be used in production after a reviewer has marked it as valid and then finally marked as released by the releaser. For the internationalization of MontiGem and MontiGem-based applications, the runtime engine of MontiGem, the system of MontiGem in which the generator is integrated, should be internationalized and the generators used in MontiGem should be extended to provide these functionalities.

Contents

Chapter 1

Introduction

1.1 Enterprise Information Systems and MaCoCo

Enterprise Information Systems (EIS) are information systems that can be used to efficiently represent the operational information of a company [wika]. As a robust foundation, Enterprise Information Systems offer companies the opportunity to coordinate their business processes in a user-friendly graphical user interface (GUI) with the goal of increasing the business productivity of the company. They also enable the company to standardize its operations. Because the focus of Enterprise Information Systems is on effectiveness, data must be presented in a manner that benefits the decision-makers [cio].

In this thesis *MontiGem* and *MontiGem-based applications* like *MaCoCo*[1] were internationalized. *MaCoCo* is a financial services software which is currently used internally at the RWTH and is developed by the Software Engineering Chair of the RWTH in cooperation with the Controlling Chair. In figure 3 you can see the main dashboard page of this system where the data is already presented in different types of charts and tables.

MontiGem serves practically as a basis for *MaCoCo* and offers the most basic functionalities of an Enterprise Information System. For *MaCoCo* this system was extended by domain specific components.

1.2 Software Internationalization

Software companies today distribute their software in the global marketplace with the goal of generating more revenue. "Dealing with *internationalization* is a key aspect when building worldwide applications and most developers have to face dealing with it sooner or later", describes it [med].

For many companies *software globalization* seems a simple step at first, but they often forget that countless different languages are spoken around the world resulting in the need to adapt their software to both the language and the culture of a particular market. Software companies, however, often start developing software that is only available in the language in which the software product initially entered the market. *Software globalization*

[1] https://macoco.rwth-aachen.de/

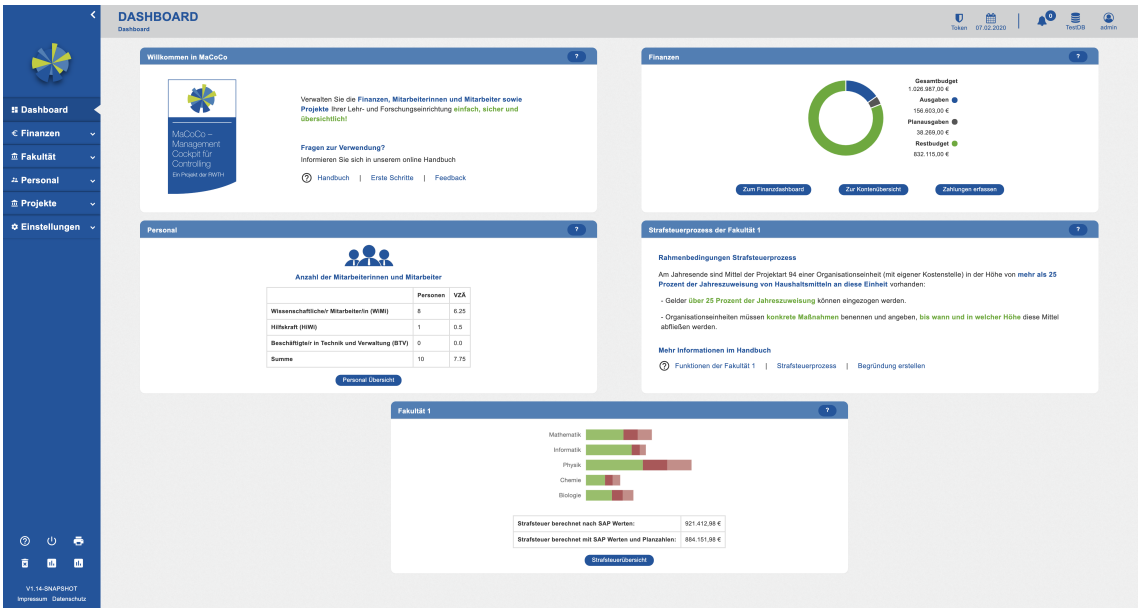


Figure 1.1: Main dashboard page of *MaCoCo*.

then means serious changes to the software architecture, which in turn results in great costs for developing these adjustments. Some companies thus decide to sell their product only in English, but users having problems with understanding various elements of the User Interface might decide to switch to a similar software product, which is available also in the language that they can understand regardless of which software product might be more suitable for them in a functional viewpoint. For this reason, it is a best practice in traditional software development already at the beginning of the development of a system to carry out an *internationalization* so as not to have to change the architecture later.

1.3 Model-Driven Engineering

Model-Driven Development of software systems is an alternative to traditional software development where models are the central artifacts of development instead of source code. Models can be of graphical or textual nature, but in the course of this thesis we will always refer to models of textual nature. *Modeling languages* (cf. section 3.2.2) can be developed to create these models. These languages describe the syntax and semantics of the models to be created, so that it is possible to transform them into an internal data structure and perform formal validations on them. Models fundamentally abstract some technical aspects of the domain and often depict a specific view on a system so that they can be understood by domain experts and enable them to play an active role as modelers in the model development. By using *software generators* (cf. section 3.2.3), executable systems can be generated from models. For this purpose the existing models are augmented with additional information, mostly of a technical nature, and then transformed into code written in a *General Purpose Languages* (GPLs) so that it is executable. The development of *modeling languages* and *software generator*s is a complex process. In practice, already existing Language Workbenches are used with which it is possible to define *modeling languages* in a simple way by specifying a grammar. In addition to the automatic generation of parsers based on the specified grammar, an infrastructure is often generated that facil-

itates the creation of *software generator tools*. In the context of this work the Language Workbench *MontiCore* was used, which is developed at the RWTH Aachen University. An overview of its functionality can be found in the course of this thesis.

Chapter 2

Structure of the Thesis

Chapter 3 first introduces some basic concepts. In section 3.1, we will first discuss in more detail what is meant by the process of *internationalization* and *localization* of software systems. This section also introduces the standardized *localization* format *XLIFF*, which standardizes the way localized data is exchanged between multiple tools. This format allows an efficient exchange of the translation units between modelers and (external) translators making it possible to use the developed *internationalization* infrastructure for *MontiGem* and *MontiGem-based applications* also in large projects. Section 3.2 introduces some basics of *model-driven software development*. Models are discussed in more detail, as well as the modelling languages in which these models are developed, and software generators are introduced with which executable systems can be generated from the models. In section 3.3 some aspects of the *MontiCore Language Workbench* are introduced, which have played a special role in the internationalization of *MontiGem* and *MontiGem-based systems* like *MaCoCo* in this thesis.

For the *internationalization* of *MontiGem-based applications* the *GUIDSL*, a *modeling language* for describing their UI, had to be extended. In chapter 4 some aspects of this *modeling language* are discussed. This chapter is accompanied by a model example based on which further aspects of this *modeling language* are explained. This chapter also gives a brief overview of the technical implementation of GUITool, which can be used to generate executable user interface code from *GUIDSL* models.

Chapter 5 is dedicated to the technical realization of the *internationalization* of *MontiGem-based applications*. The concepts of the previous chapters are used for this purpose. In particular, it shows in detail how an *internationalization* of *GUIDSL* has been realized, using the example of the previously presented model.

This work concludes in chapter 6 with an overview of the *localization tool*, which was developed in the context of this work to allow *MontiGem* developers to enter in the translations of the developed systems in *MontiGem* itself.

Chapter 3

Preliminaries

3.1 Internationalization and Localization

3.1.1 Internationalization and Localization Process

Globalized applications do not hardcode user interface elements that differ by language or culture as texts or images directly in the source code. Instead, these elements are stored as translation units in separate files and added to the project during the build process or dynamically during the runtime execution of the application. [wikb]

It is usually not sufficient to localize a software only into different languages as cultural differences within a language are often to overcome in order to provide a truly customized software version to the users. For example, the software should distinguish between British and American English if it is available in both English and American market. [wikc]

The process of making an application available in several locales can be divided into two sub-processes: The *internationalization process* and the *localization process*. The *internationalization* of software is the development of a system in such a way that in a later step it can be easily localized for different languages and cultures. The process of *internationalization* is generally abbreviated as *i18n*, where the *18* stands for the number of letters between the first and last letters of the word *internationalization*. The process of *internationalization* also includes the implementation of mechanisms so that users of the system can choose the language they want to display. Often it is also implemented that the language to be displayed is automatically determined by certain other factors without the user explicitly selecting a language. For example, the language stored in a web browser already provides information about the language in which a user wants to display a web page. Nevertheless, the user should always have the possibility to manually adjust the language to his needs.

3.1.2 XLIFF

Different languages have, among other things, different punctuation and pluralization rules. This makes the localization of software a complex and costly process during development. Especially in the development of *enterprise information systems* (cf. chapter 1), the translators of the application have to deal with domain-specific vocabulary. Often, the translation of such applications requires the use of professional translators or specialized software

tools for localizing applications. However, the use of external translators always presents some problems, as external translators in general have an aversion towards localizing data written in a language selected by the software developers. This made it necessary to standardize the way the data is passed between programmers and the tools used by translators during the localization process. [Las, VS04]

Over time, many different formats have become established for effective software localization. The following section present in details some aspects of the standardized *XLIFF localization format*, which is already widely used in the industry, and thus allows an efficient exchange of units to be translated from an application with possibly external translators who are not part of the application development team. In section 5.1 in the further course of this thesis it is presented how all the elements to be translated in *MontiGem* and *MontiGem-based applications* can be extracted into localization files in this format from the models and the source code generated from them.

Introduction to XLIFF

XLIFF is an XML-based, lossless interchange format designed by industry experts for application elements that differ by language or culture. Each element to be translated in the UI represents a translation unit, which in turn is internally saved as a key-value pair. As can be seen in the example found in figure 3.1 each translation unit is enclosed with the `<trans-unit>` and is given a unique identifier attribute. Each translation unit contains at least the two child elements `<source>` and `<target>` where the former elements represents the text written in the application's source language, whose translation must be written inside the latter element. [xli]

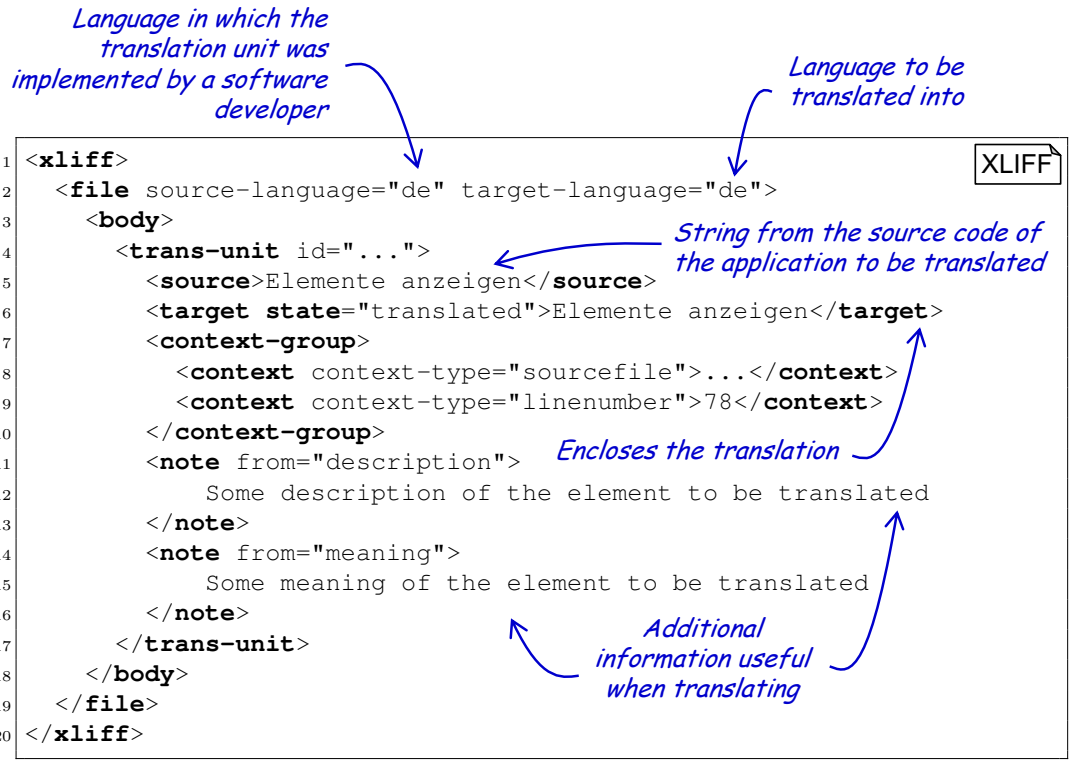


Figure 3.1: Example of a slightly simplified *XLIFF* file with exactly one translation unit

XLIFF allows via the usage of the `<note>` tag to provide translators with additional information about translation units that might facilitate the localization process by, e.g., explaining the context of the unit in more detail to the translator. In listing 3.1, e.g., an additional description and meaning is given which may enable the (external) translator to translate correctly. The optional `from` argument specifies more precisely the kind of a note whose values, however, are not standardized and may not be evaluated by some translation tools. [xli]

Typically, multiple translators work collaboratively to localize an application. This leads to further requirements for a localization format. *XLIFF* standardizes states for translation units, which should facilitate the collaboration of several translators among themselves and/or the actual software developers. By default, each translation unit is in a certain localization state. Among the ten different states defined by the standard, the two most important ones are the states `translated` and `needs-translation`, which indicate already translated translation units and units that have not yet been translated, respectively. The localization process of an application using the *XLIFF* format has therefore the overall goal to convert all translation units with state `needs-translation` into the state `translated`. In production, however, it is often necessary to have certain intermediate states available. For this purpose, translation units can also be put into the states `needs-review` and `reviewed`, which are supposed to indicate that the corresponding translation units already have a translation, but that they should not yet be used in production until a check has been carried out. If a translation unit is in one of those two states, depending on the configuration of the application, either a placeholder will be displayed in the UI that indicates a missing translation or the contents of the translation units will be displayed in the source language. [xli]

A somewhat simplified process that a translation goes through until it is used in production is shown in figure 3.2. In real uses there are further transitions possible, which are not included in this state chart. For example, it is possible that after a translation has already been used in production, it is transferred into the state `needs-translation` as certain conditions in the corresponding application make it necessary to adapt the existing translation. [xli]

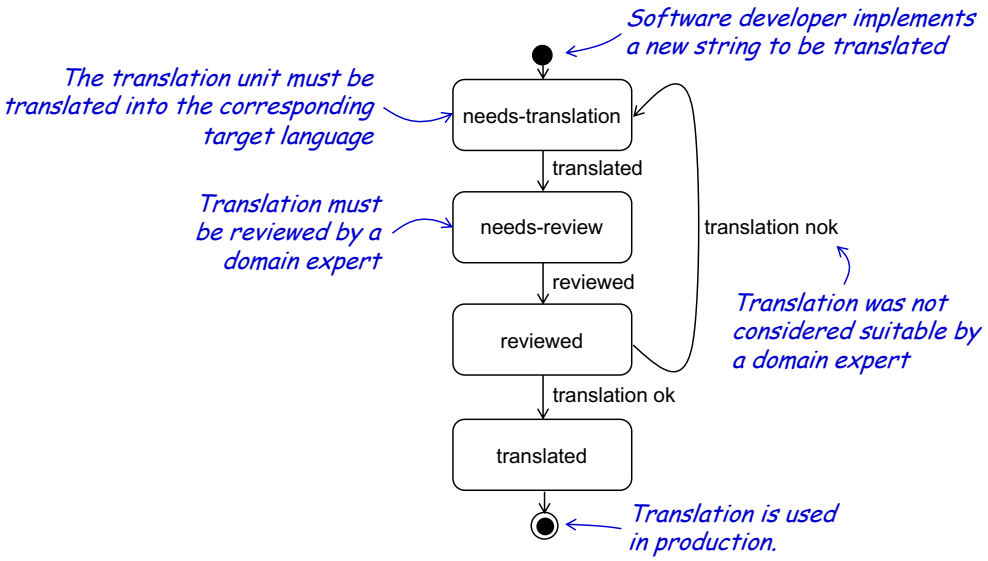


Figure 3.2: Typical process that a translation unit goes through

Variable Content in XLIFF

Interpolations Texts are often dynamic and depend on variables whose values can only be access during the runtime execution of a system, e.g., because the system waits for the result of a database query triggered by some action by the user in the UI. One possibility is to split translation units and to display dynamic content in the UI as non-internationalized strings between these internationalized units. However, this greatly increases the complexity of the translation units generated, as this results in fragmentation of the translation units to be translated from the UI. This also makes it more difficult to translate the translation units, as translators cannot see how these two distinct translation units belong together in the *XLIFF* file. It is not really possbile to use references to other translation units in *XLIFF*. [icu]

In many globalized applications that use the *XLIFF* format internally, *interpolations* can be used instead, which make it possible to use variables in translation units that are replaced at runtime. The advantage of using *interpolations* is that a unit to be translated no longer needs to be split into multiple (in XLIFF files unrelated) translation units and the translator can translate the translation unit as a whole. To use *interpolations* in *XLIFF* files the tag `<x id="..." equiv-text="..."/>` is used. Since this element is also for other *XLIFF* functionalities, which are not discusses in this thesis, it is also necessary to specify an `id` attribute that marks this element as an *interpolation*. In the attribute `equiv-text` you can specify the name of the variable as it was defined in the application by some software developer. [for, icu]

Figure 3.3 shows an example of how a variable name of a user can be dynamically inserted in the translated text of the translation unit at runtime for some internationalized element. In this example, the UI would display the text `Hello Miguel` if the user's name is `Miguel`. In the example, the *interpolation* element is replaced at runtime with the value of the variable `username`. When translating, the translator must ensure that he or she does not change the attributes of the *interpolation* element, otherwise the variable cannot be found at runtime and the content will not be inserted dynamically into the translation unit. [for, icu]

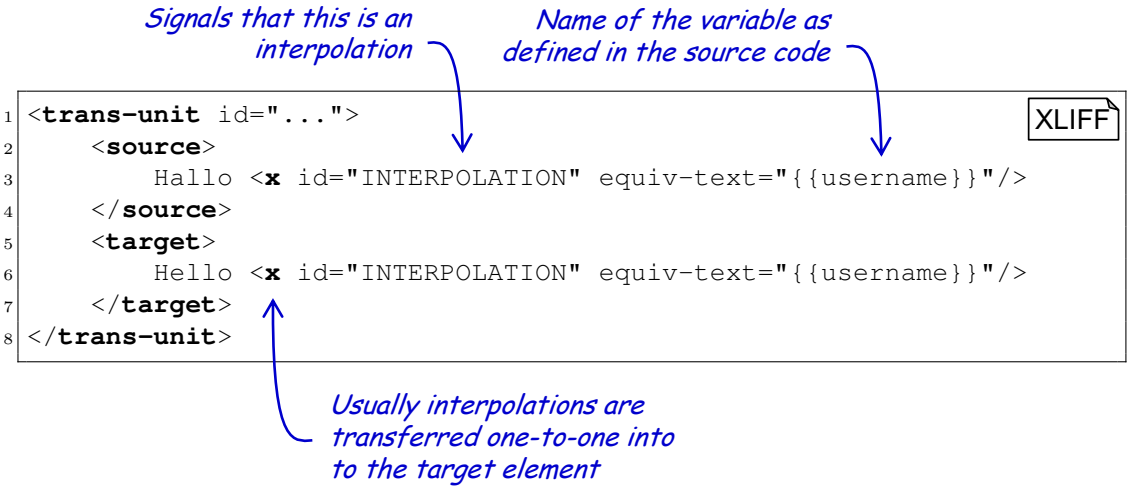


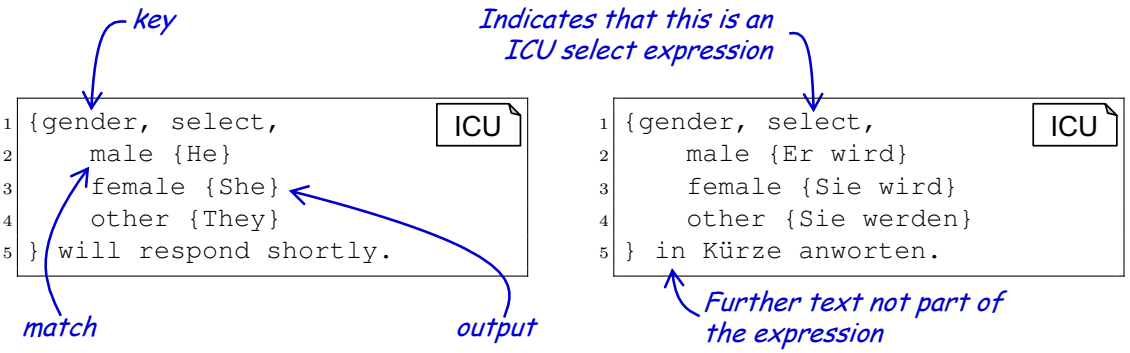
Figure 3.3: Example usage of *interpolations* in XLIFF

ICU Expressions Different languages have different pluralization rules and grammatical constructions that complicate the *internationalization* of an application. Translators have to move parts of a text to be translated to other places so that the translated text makes sense grammatically in the target language. [for]

Suppose a translator has to translate `The value for the {0} has an influence on the value of the {1}` where `{0}` and `{1}` stand for any two variables whose value is not known to the translator. This text can be easily represented in XLIFF as a translation unit by using the *interpolation* technique already introduced previously in section 3.1.2. When translating this text into German the problem occurs that the articles of the source text must be declined according to German grammar rules. The sentence "The value for the inlet opening has an influence on the value of the minimum." [mad] could, e.g., be translated into "Der Wert für die Eintrittsöffnung beeinflusst den Wert für das Minimum." [mad] and "The value for the center has an influence on the value of the average time." [mad] into "Der Wert für den Mittelpunkt beeinflusst den Wert für die Durchschnittszeit." [mad]. Without the use of *ICU Expressions*, the English text cannot be translated and stored in a file encoded in the XLIFF format. [mad]

The ideal solution would be to transform the text in its original form in such a way that no declination is necessary in the translation. However, this is not often possible given how many languages a text may be translated into. Thus, by using interpolations it is not possible to translate this text reasonably because of the dependency of the German articles on the variables that are not known during the localization process. [mad]

With *ICU Expressions*, translators can easily rearrange or replace parts of texts to translate, so that spelling, grammar, and conjugation subtleties of a language can be expressed. In the following two types of *ICU Expressions* are presented: *ICU Select Expressions* and *ICU Plural Expressions*. [for]



(a) English variant of the ICU select expression example

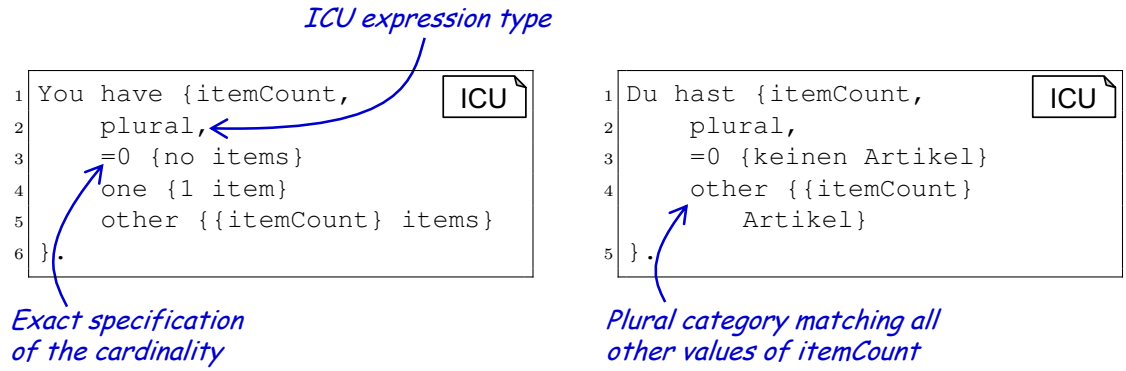
(b) German variant of the ICU select expression example

Figure 3.4: Example of an *ICU Select Expression* in English and German. Usually a software developer defines an *ICU Expression* in the source language and the translator translates it into an equivalent *ICU Expression* in the target language.

ICU Select Expressions *ICU Select Expressions* can be used similar to switch statements available in some programming languages to select sub-messages based on a fixed set of possibilities. The syntax `{key, select, matches}` is used for select arguments where the `key` represents some variable input value which is matched against one of the items of

the space-separated list of `matches`. A match is written in the form `match {output}` where `match` is a literal and `output` is the value representing this match. A match can be described as a case statement of a switch in a programming language. An example of how *ICU Select Expressions* can be used to dynamically set texts based on the gender of a person can be found in figure 3.4. [for, icu]

ICU Plural Expressions Plural *ICU Expressions* can be used to select sub-messages based on a numeric value and work similar to *ICU Select Expressions* with the difference that the key value is expected to be a number and is mapped to a plural category. The syntax for *Plural ICU Expressions* is `{key, plural, matches}`. A match is a literal value that matches one of the following plural categories: `zero`, `one`, `two`, `few`, `many`, `other`, `=value` [1].



(a) English variant of the ICU plural expression example

(b) German variant of the ICU plural expression example

Figure 3.5: Example of an *ICU Plural Expression* in English and German

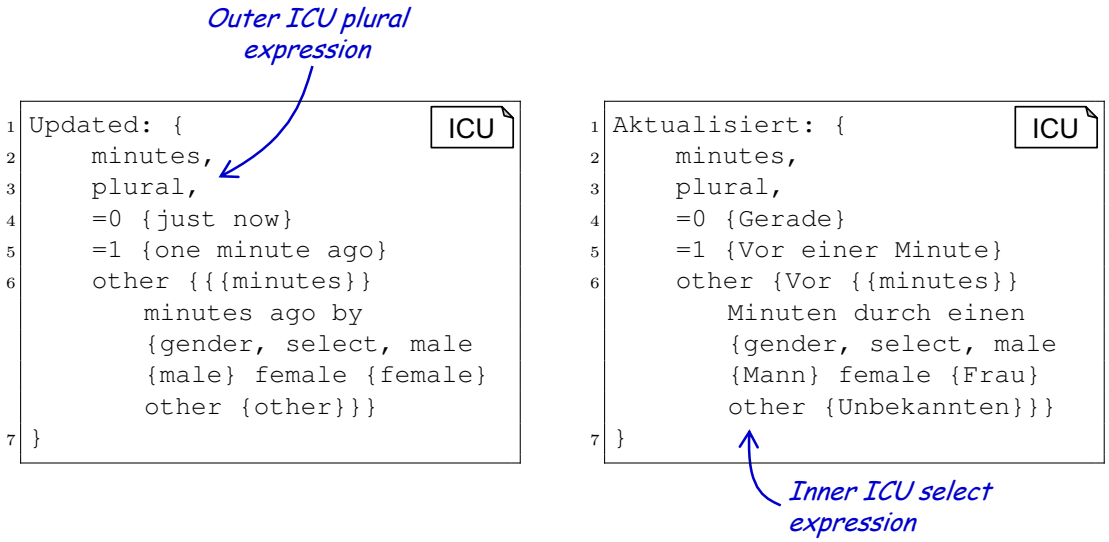
The examples in figure 3.5 show how *ICU Plural Expressions* can be used to output in an English and German expression the number of a given element in the correct grammatical structure.

Other ICU Expression Types Besides *ICU Select Expressions* and *ICU Plural Expressions*, the official ICU Guide describes further types of *ICU Expressions*. However, these are not presented in this paper, as they are not yet fully supported in the Angular Frontend Framework at the time of writing. The ICU guide[2] can be used for further information. [for, icu]

Nesting ICU Expressions It is also possible to nest several *ICU Expressions* in each other to create more complex expressions. The examples from figure 3.6 show how *ICU Select Expressions* can be nested into *ICU Plural Expressions*. [ang, for]

[1] Plural categories explained in more detail on https://formatjs.io/guides/message-syntax/.

[2] The ICU guide available on https://formatjs.io/guides/message-syntax/ gives more information on how to use the other types of *ICU Expression*.



(a) English variant of the nested ICU expressions example

(b) German variant of the nested ICU expressions example

Figure 3.6: Example of a nested *ICU Expression* in English and German [ang, for]

3.2 Model-Driven Engineering

3.2.1 Models

Model-Driven Engineering is a software engineering paradigm in which models represent the central software artifacts instead of source code such as in traditional software engineering methodologies. Models are generally more abstract than code written in a *General-Purpose Language* (GPL) like Java and thus represent a simplified version of a system or describe a specific view of it. Since complex systems in their entirety are often difficult to understand, models eliminate unnecessary components that are not important for a particular system view by reducing the number of parts involved in the system or simplifying dense interconnections between parts of it. Simplifying the complexity of a system allows the analysis of properties of the system using the model. UML class diagrams, e.g., can be used to abstractly describe the domain and their internal dependencies. The created models can be used as input for software generators, which can extract the technical information from the models and generate executable systems (cf. section 3.2.3). [RH17]

Models often encapsulate domain-specific knowledge and thus make sense for experts working in this domain. *Domain-Specific Languages* (DSLs) or in general *modeling languages* (cf. section 3.2.2) are developed to describe the models. These languages enable domain experts to create their own models using well-defined syntax and semantics. During the development life cycle different DSLs can be used for different purposes. For example, there are DSLs that are used for designing, programming and testing software systems and others that are used for describing the behavior of a system or processes. Class diagrams can be used for various purposes. The creation of class diagrams, for example, allows the documentation of a software system. The creation of DSLs therefore requires an intensive exchange of information between domain experts and the developers of the *Domain-Specific Languages* and the developers of the generator used, which uses the models described by the DSL to create executable code. [RH17]

3.2.2 Modeling Languages

Models are expressed with the use of *modeling languages*. In particular, so-called *Domain-Specific Languages* (DSLs), which are specific to one domain, are used to create models. One of the differences between *modeling languages* and GPLs like Java or Typescript is that the latter is domain-specific, whereas DSLs are.

Modeling Languages generally consist of a *concrete syntax* and a *abstract syntax*. The *concrete syntax* of a *modeling language* determines the representation of that language and can be either graphical or textual. An *abstract syntax*, on the other hand, represents the internal structure of the language, whereas semantically irrelevant aspects of the language like whitespace are omitted. By parsing the textual input models in a *Model-Driven Software Development* process, the *Abstract Syntax Tree* (AST) is obtained, which is the central structure of the model processing (cf. section 3.3). [RH17]

3.2.3 Model Processing

Generators

Code Generators play an essential role in *Model-Driven Development*. They are used to generate a set of executable output files from a set of models. Internally, software generators transform the abstract syntax of the input models. These transformations can be divided into two different categories, namely *model-to-model* (M2M) and *model-to-text* (M2T) transformations. In M2M transformations, input models of a certain *modeling language A* are transformed into models of another *modeling language B*. This makes it necessary that the generator has knowledge about the syntax of both *modeling languages*. In M2T transformations input models are transformed into a string. With M2T transformations it is not necessary that the generator has knowledge about the exact syntax of the string. Thus the complexity of developing M2T transformations can be done with less effort. [RH17]

Development Roles

The code generated from the models is usually integrated into an existing software system–the runtime engine–that provides generic components that are used by the generated code. In the context of *Model-Driven Development*, these components are in theory developed independently of the specific applications to be generated by a *component provider*. In addition to the *component provider* role in a *Model-Driven Development*, there is also the role of the *tool provider*, which is responsible for the development of the generator and the run-time system. The components of the *component provider* are integrated into the runtime system and can therefore be used by the generator as required. [RH17]

Models usually abstract technical aspects of an application, so that it is necessary to deduce these technical details from the models during the generation process and add them when transforming the AST. As one generator is potentially used for the generation of several applications with different technical requirements, software generators can be built to allow parameterization so that the generator can be adapted to the needs of a certain application. A *tool smith* adds this project-specific information and orchestrates the generator. [RH17]

Figure 3.7 shows the different roles in a *Model-Driven Development* environment.

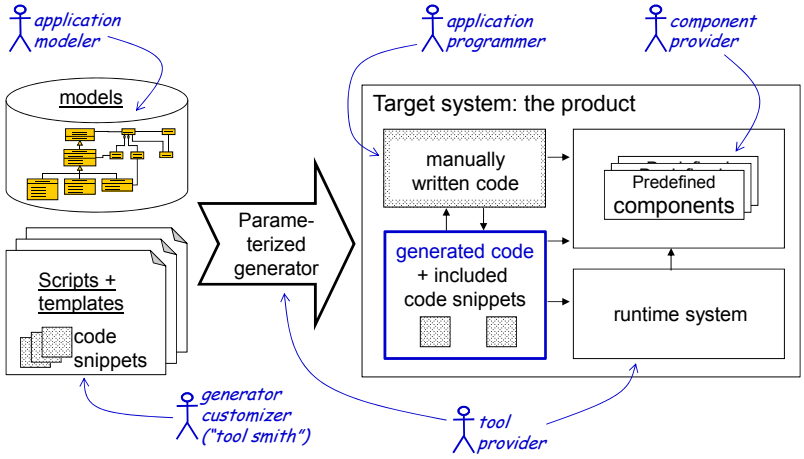


Figure 3.7: Structure of a generator [RH17].

3.3 MontiCore

In the following *MontiCore* [RH17] is presented, a language workbench developed at the Chair of Software Engineering at the RWTH Aachen University, which can be used to develop textual *modeling languages* (cf. section 3.2.2) and software generator tools that transform models written in these *modeling languages* into executable code.

3.3.1 Basic Grammar Concepts

Central artifact in the development of *modeling languages* with *MontiCore* are context-free *MontiCore* grammars, which represent the *concrete syntax* and from which *parsers* as well as the *abstract syntax* can be derived automatically. *MontiCore* allows to combine several of these grammars by composition using techniques like *language inheritance* or *language embedding*. This allows monolithic grammar artifacts to be modularized, allowing separate development. [RH17]

MontiCore grammars are an extended grammar format based on EBNF. Each grammar defined in *MontiCore* consists of a head and a body, where the head contains the name of the grammar and other information that is used to compose several grammar artifacts. By specifying *productions* in the body of a *MontiCore* grammar, both the *concrete* and *abstract syntax* of the *modeling language* can be defined. [RH17]

In this section some basic concepts of a *MontiCore* grammar are presented, which have played an extremely important role in the *internationalization* of *MontiGem-based applications*. In the following, *lexical productions*, *productions* and *tokens* are explained in more detail. In addition to these grammar concepts, there are also a number of other concepts such as *enumarations*, *interface nonterminals* or *abstract nonterminals* that do not play an important role in this work and are therefore not explained in more detail. More information on their use is available in the *MontiCore Reference Manual*. [RH17]

Lexical Productions

Lexical productions describe atomic names, values or keywords like `if`. They represent the simplest type of productions that can be in the body of a *MontiCore* grammar. A *lexer* analyses and outputs a sequence of tokens, which are passed to the *parser* in order to generate the AST from the model. *MontiCore* already contains some grammars that are intended to be reused in other grammars. Among other things, there are tokens for whitespace, comments and a number of literals such as `StringLiteral`, so that you no longer have to define these tokens yourself for every DSL. [RH17]

Productions

Productions consist of a left side that defines a *nonterminal* and a right side that describes how this *nonterminal* is defined. The main difference to lexical productions is that arbitrary other *nonterminals* can be on the right side of productions, while lexical productions cannot because they are atomic productions. [RH17]

Terminals

Terminals are atomic elements that can be on the right side of a production. Terminals are framed with quotation marks and are usually not part of the AST generated from the models, since these are not semantically relevant and are only part of the *concrete syntax* of the respective *modeling language*. However, there are some cases in which terminals are semantically relevant and therefore have to be part of the AST. For such cases, *MontiCore* terminals can also be framed with square brackets, whereby *MontiCore* generates a boolean for the associated *nonterminal*, which saves whether the terminal occurred or not in the model. [RH17]

3.3.2 Abstract Syntax Tree (AST)

From the *MontiCore* grammar, *MontiCore* generates the *lexer*, *parser* and classes for the *abstract syntax* of a *modeling language*. To generate an executable system, the generator transforms this input AST. As explained in the previous chapter, the AST of a model of a *modeling language A* can be transformed into an AST of the *modeling language B* by complex transformational mapping. On the other hand, it is also possible to transform an AST in such a way that the output AST is only augmented by relatively simple additional information after the transformation. How the transformation is done always depends on the application. In this thesis the M2M transformation is rather uninteresting. The projects shown in the course of this work use only M2T transformations and augment the models only with additional information, which is calculated based on the model but is not part of the *concrete syntax*. [RH17]

In this chapter a short overview of the generated classes for the *abstract syntax* of a MontiCore grammar is presented, which are of special interest in this thesis. Other concepts not addressed can be found in the *MontiCore Reference Manual* [RH17].

Mapping nonterminals to AST Nodes

For each *nonterminal NT*, defined by the left side of a production, *MontiCore* generates an AST class *ASTNT* whose content depends significantly on the production body: If a *nonterminal* appears on the right side of a production, this results in a composition, which behaviour depends on the cardinality with which a *nonterminal* appears. For example, an optional *nonterminal OPT*, marked with a question mark after the corresponding *nonterminal* in the grammar, causes *MontiCore* to generate a class attribute for the *nonterminal NT* on the left side of the production that has the name of the optional *nonterminal OPT* and is of the Java type *Optional*. See figure **??**

Traversing the AST

To add additional information to an AST it is necessary to traverse the AST. *MontiCore* allows to execute and specify the traversal algorithm separately from the actual operations on the individual AST nodes. In fact, *MontiCore* generates a visitor interface that contains a depth first traversal algorithm for the AST, which is used by default to traverse the AST, so that it does not need to be implemented by a language engineer for the *modeling language* to be developed. [RH17]

MontiCore generates for a *modeling language L* a visitor interface *LVisitor* that contains four methods for each *nonterminal NT*:

- `handle(ASTNT node)`
- `traverse(ASTNT node)`
- `visit(ASTNT node)`
- `endVisit(ASTNT node)`

These four methods are used in conjunction to traverse the associated AST node `NT`. The visit method is called when the node `ASTNT` is entered while traversing the AST. `endVisit(ASTNT node)` is called when the node is left while traversing and traversing is either finished or jumps to the next AST node. [RH17]

The `traverse(ASTNT node)` and `handle(ASTNT node)` methods together define the strategy for traversing the AST node for the *nonterminal* `NT`. By default the depth-first approach provided by *MontiCore* is used. Only in special cases, such as when the children of the AST node are to be processed in a special order, it is necessary to adjust the traversing algorithm for the respective node. This step was not necessary in the context of this thesis, so that this is only explained in the margin here. [RH17]

MontiCore defines standard implementations for the visitor interface, so that the language engineer must only implement generator-specific behavior by overwriting these methods in a concrete visitor class. Methods that are not of interest to the language engineer can be ignored, so that the concrete class uses the standard implementations generated by *MontiCore*. A concrete visitor implements the *MontiCore* generated visitor interface `LVisitor`. [RH17]

3.3.3 Generator Engine

After a model has been transformed into an AST by parsing it with *MontiCore*'s generated *parser* and this AST has been enriched with further information by using concrete classes that implement the generated visitor interface, in the last step of a typical software generation process textual file artifacts can be generated from the enriched AST by using the *Generator Engine* provided by *MontiCore*. *MontiCore* follows a *template-based approach*, where templates describe the general structure of the generated artifacts in which further expressions can be used to pass dynamically data to the templates. [RH17]

In this section, some aspects of the *Generator Engine* that have played a special role in this work are explained in more detail. In 3.3.3 we will first introduce the *FreeMarker Template Language* (FTL). Afterwards it is shown how to generate artifacts with the *GeneratorEngine* and the created templates. Afterwards the *TemplateController* and the *GlobalExtensionManagement* will be explained in sections 3.3.3 and 3.3.3. The *TemplateController* can be used to modularize the existing templates. Among other things, it enables the tool creator and/or tool smith (cf. section 3.2.3) to load templates as subtemplates into already existing templates. The *GlobalExtensionManagement* helps to define global variables, which can then be made available to all templates. This concept is used on a large scale, especially in the visitors used in the *GUIDSL* (cf. chapter 4). [RH17]

FreeMarker Template Language

Apache FreeMarker is a template engine that is used in *MontiCore* to generate textual output based on templates and the information from the AST. The templates are written in the *FreeMarker Template Language*, which contains powerful mechanisms that make the use of this template engine comfortable and well integratable into larger projects with little effort. [RH17, Naz17]

Expressions in FTL are similar to Java expressions and can use variables. FreeMarker's expressions are enclosed in `${...}`. For example, `${obj.name}` can be used to access the `name` attribute of the object `obj`. FTL expressions can also di method calls so that the previous expression that used a variable is equivalent to the FTL expression `{obj.getName()}`. [RH17, Naz17]

During the generation process, the templates are transformed so that the FTL expressions are replaced with data from the AST. Thus the templates contain little programming logic, because this logic is already added when traversing the input AST. This is the way how the data is calculated completely independent of the templates and while the templates remain exactly the same. This allows the parameterization of the generator. [RH17, Naz17]

In the FTL a number of different control directives can be used to control the execution flow similar to a GPL like Java. As can be seen in the listings presented in the following, directives are enclosed in tags of the form `<#directive parameters>` and `<#/directive>` or consist of a single tag. For example, variables can be declared using `<#assign name=value>`, as can be seen in listing 3.1, or Java lists can be iterated using the `<#list>` directive. [RH17, Naz17]

The FTL also includes a number of built-in functions that are especially useful for transforming dynamic strings. These built-in functions are written behind expressions (*strings*, *integers*, *booleans*, ...), with the delimiter `?` in between. For example, the built-in function

`cap_first` can be used to capitalize a string, so that, e.g., `"dataTable"?cap_first` is evaluated to `DataTable` during generation. [RH17, Naz17]

```
<#-- we assume variable ast is pointing towards an AST object
   containing class information (this is a FreeMarker comment)
--#>
package ${ast.packageName};
<#assign cname = ${ast.name}>

/* Definition of a standard ${cname}Factory (a Java comment) */
public class ${cname}Factory {
    protected static ${cname}Factory f = null;
    protected ${cname}Factory() {}
    public static ${cname} create() { if (f == null) { ... } }
}
```

FTL

Listing 3.1: Content of a template file before applying the template [RH17]

Overview of the GeneratorEngine

MontiCore's *GeneratorEngine* can be used to start the process of generating and processing the templates. For this purpose the `generate` method can be used, whose signature can be found in listing 3.2, which first opens a model file with the path `filePath` and then executes the given template `templateName` on the given AST node `node`. [RH17]

```
public class GeneratorEngine {
    ...

    void generate(
        String templateName, Path filePath,
        ASTNode node, Object... templateArguments);
}
```

Java

Listing 3.2: Signature of the `generate` method [RH17]

By setting `templateArguments` further attributes can be passed to the given template. During the execution of the template these are then available as arguments. Since FreeMarker is an untyped language, *MontiCore* offers the possibility to use the special operator `signature`, which can be used to specify in the templates which arguments are expected when processing this template. This emulates method calls like those found in Java, TypeScript or other GPLs. An example use of this operator can be found in figure 3.8. In particular, figure 3.8a shows how the generate method can be used in a tool to start the generation process with the template in figure 3.8b.

In listing 3.8b the *TemplateController* `tc` is used to specify the signature of the template, which is explained in more detail in the following section.

```java
GeneratorSetup s =
    new GeneratorSetup();

...

GeneratorEngine ge =
    new GeneratorEngine(s);

ge.generate(
    "tpl/TSPage.ftl",
    Paths.get("example-page"),
    ast,
    attr1,
    attr2,
    attr3);
```

(a) The *GeneratorEngine* must be configured with the *GeneratorSetup* beforehand. An AST node and three further attributes are transferred to the template. The signature is optional and serves mainly for a better overview.

```
${tc.signature("ast",
    "attr1", "attr2",
    "attr3")}

Value of attribute 1:
${attr1}

Value of attribute 2:
${attr2}

Value of attribute 3:
${attr3}

Access to AST node:
${ast.pageName}
```

(b) The specified AST node and the attributes are mentioned in the signature in the template. By using FTL expressions it is possible to access the attributes specified in the `generate` method.

Figure 3.8: The *GeneratorEngine* provides the `generate` method that can be used to start the generation process. [RH17]

Template Controller

This section introduces the *TemplateController* `tc` used in *MontiCore*, which offers typical template operations such as the `signature` method introduced in the previous section, which allows you to specify the expected attributes of a template. [RH17]

In contrast to the *GlobalExtensionManagement*, which is discussed in the following section and instantiates exactly one object that all templates have common access to, for each template execution a separate *TemplateController* is instantiated, with which the templates gain access to additional template-specific information such as the name of the current template or the name of the package. [RH17]

In addition to this operation, the *TemplateController* is often used to load other sub-templates in templates, so that it is possible to compose the templates in a tree-like manner. This makes the templates easier to read, since they can be kept relatively small, but on the other hand, it also promotes the reusability of individual generic templates that can be integrated into other templates as required. [RH17]

MontiCore's *TemplateController* class provides a number of different methods to load sub-templates into a template. In the context of this work, two of them were of particular importance, which will be explained in more detail in the following. Further methods can be found in the *MontiCore Reference Manual* [RH17].

The easiest way to load a sub-template in a template is to use the `include` method that requires you to specify the name of the sub-template and an AST node that is being passed to this template during template execution. If you want to pass data to a sub-template that cannot be accessed via the passed AST node, it is possible to use the `includeArgs`

method in a template instead, which similarly to the `generate` method introduced in section 3.3.3 offers the possibility to pass any other data to a template. The signature of these two methods can be found in listing 3.3. [RH17]

```java
class TemplateController {
    StringBuilder include(String templateName, ASTNode ast);
    StringBuilder includeArgs(String templateName,
                              ASTNode node,
                              List<Object> templateArguments);
}
```

Listing 3.3: Methods provided by the *TemplateController* for template inclusion [RH17]

An example of how to use the `includeArgs` method in a template is shown in figure 3.9, where the parent template in figure 3.9b loads the sub-template specified in figure 3.9a. Using the `includeArgs` method, two additional strings are passed to the template in figure 3.9a, so that a signature is introduced in the sub-template, allowing the sub-template to access the data through FTL expressions using the names of the attributes from the signature. [RH17]

```ftl
${tc.includeArgs(
    "sub-template",
    ast.getAChild(),
    "Miguel",
    "Franken")}
```

(a) Parent template that includes a sub-template

```ftl
${tc.signature(
    "firstName",
    "lastName")}
Hello ${firstName}
    ${lastName}.
```

(b) Child template included in template in listing (a)

Figure 3.9: The *GeneratorEngine* provides the `generate` method that can be used to start the generation process. [RH17]

Global Extension Management

MontiCore's GlobalExtensionManagement manages among other things global variables, which are available in each of the templates. This is another possibility for the language engineer to transfer data from Java to the templates. Global variables can be defined directly in the templates or in the Java code that calls the templates, e.g. in a concrete visitor. The first of these variants did not play a role in the context of this thesis and is therefore only mentioned here for completeness. The signature of the method `defineGlobalVariable` used to define global variables can be defined can be found in listing 3.4. [RH17]

```java
public class GlobalExtensionManagement {
    void defineGlobalVar(String name, Object value);
    Object getGlobalVar(String name);
}
```

Listing 3.4: Signature of the `defineGlobalVar` method used to define global variables using *MontiCore*'s *GlobalExtensionManagement* [RH17]

An example of how a global variable can be defined in a visitor `SomeConcreteVisitor` for

a *modeling language* `L` can be found in figure 3.5. Usually the *GlobalExtensionManagement* is initialized in the tool of the *modeling language* and then passed to the visitor via a constructor argument. Alternatively, the `glex` object can also be defined as static in the tool. The advantage of the first variant is that you can limit the scope of the global variable to a certain visitor, for example. [RH17]

Java

```
public class SomeConcreteVisitor implements LVisitor {
    private GlobalExtensionManagement glex;

    constructor(GlobalExtensionManagement glex) {
        this.glex = glex;
    }

    @Override
    public void endVisit(ASTA node) {
        glex.defineGlobalVariable("myVar", "myValue");
    }
}
```

Listing 3.5: Defining global variables in a concrete visitor for some *modeling language* `L`.

After node `A` was visited, a global variable named `myVar` is defined in the `endVisit` (cf. section 3.3.2) method. This variable is then available to all templates of the corresponding `generate` call. In a template, this variable can be accessed directly with the corresponding FTL expression as shown in figure 3.10. [RH17]

FTL

```
<#-- Get global var -->
<#assign myVar =
    glex.getGlobalVar("myVar")
    ?cap_first>
Value of myVar is: ${myVar}
```

(a) Within the templates you have access to the object `glex` and thus also to the global variables.

FTL «applied»

```
<#--
The expression ${myVar}
was replaced
-->
Value of myVar is: MyValue
```

(b) Result after the template was applied. The comment, however, is only for clarification and would not be produced.

Figure 3.10: It is possible to access global variables managed by *MontiCore's GlobalExtensionManagement* from within the FTL templates [RH17]

Chapter 4

GUIDSL

GUIDSL is a *domain-specific language* (cf. section 3.2.2) that is used in *MontiGem* and *MontiGem-based applications* such as *MaCoCo* to describe the UI of the system in a declarative manner. A *GUIDSL* model describes exactly one page of the system which can contain itself several different components. As the *GUIDSL* is specially designed to enable to model very data intensive systems whose data needs to get displayed in a lot of different ways, the *GUIDSL* provides a variety of different components such as charts and tables that can be directly built into the models. These elements abstract usually the most of the technical aspects but are also highly configurable, so that they can be adapted to the exact needs of an application.

In the following, different aspects of the *GUIDSL* that are of special interest in this thesis are explained in more detail. In particular, the grammar of some of the elements available in the *GUIDSL* is explained in detail and a brief description is given of the technical realization of the *GUITool*, which allows to generate files in *HyperText Markup Language* (HTML) and *TypeScript* (TS) from *GUIDSL* models (cf. section 3.2.3). These files can be integrated into *MontiGem* or other *MontiGem-based applications* such as *MaCoCo* and can be extended manually if needed. These explanations are accompanied by a simple example which is presented in the following section and which is referred to in other sections of this chapter.

4.1 Model Example

A minimal example of a *GUIDSL* model that represents a page containing a card with a data table can be found in figure 4.1. This figure also shows approximately how the generated UI from this model looks and how the elements of the model relate to the elements found in the UI. The model is located in the figure in the upper half and in the lower half the generated page integrated into *MontiGem* is displayed. The page contains exactly one card that consists if a table that dynamically loads data from the database. The *GUITool* automatically generates methods for the communication with the backend.

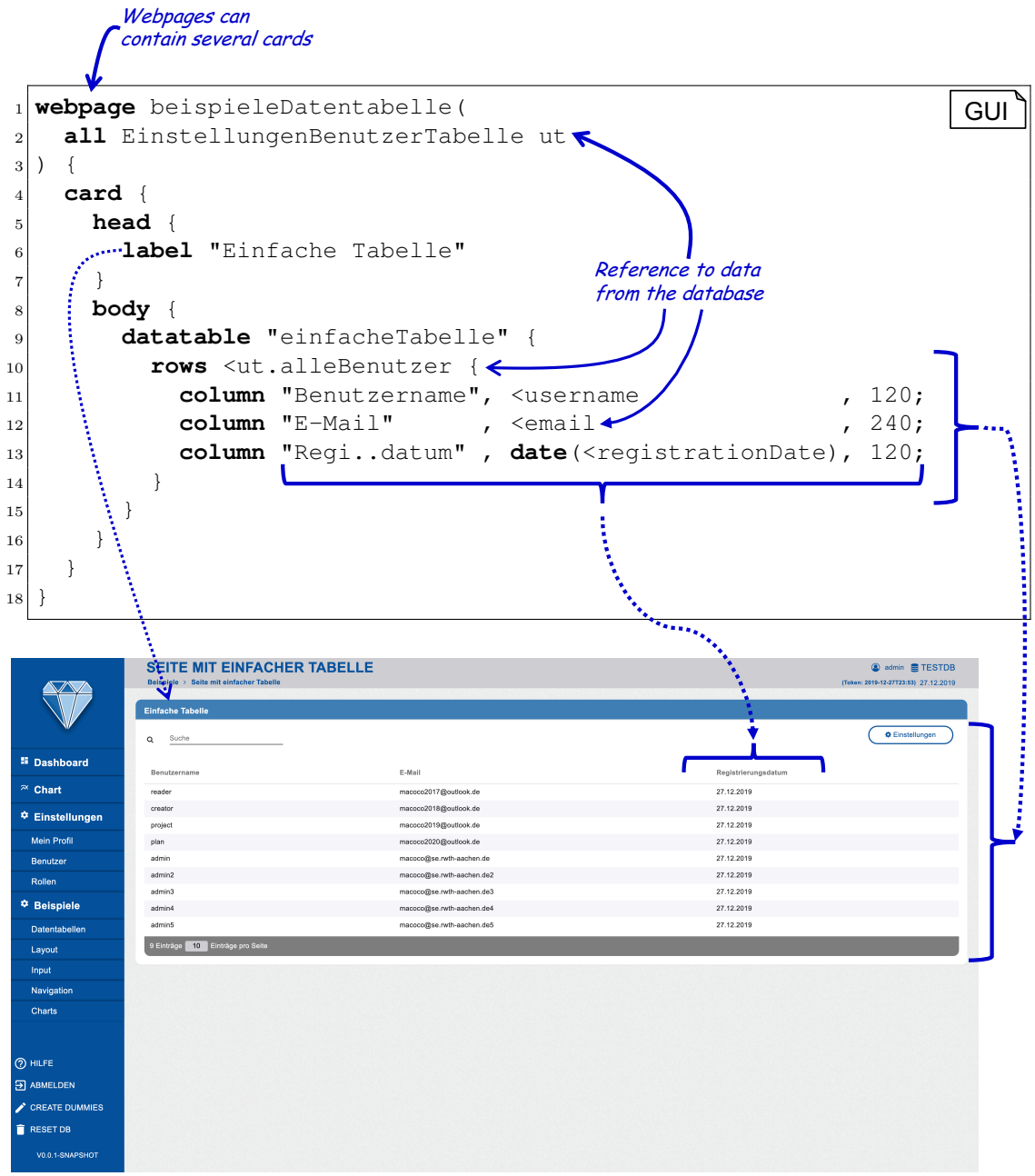


Figure 4.1: Simple example of a *GUIDSL* model and how the generated software system looks like in *MontiGem*

4.2 Simplified Grammar Explained

4.2.1 Pages

Each *GUIDSL* model has an AST node of type *ASTPage* as root element. A slightly simplified grammar for this element can be found in figure 4.2, where `Name` stands for a lexical production that is already provided by MontiCore, defined in the `MCBasics` grammar, and can be used to name elements. The naming is of utmost importance for the generation of executable code from the models, because the names given to the elements of

the *GUIDSL* are often used to distinguish between several elements of the same type. In fact, the `Name` in conjunction with other concepts is also used to create a symbol table, which will not be discussed further in this paper. For more information on the symbol table and the element `Name`, please refer to the MontiCore Reference Manual [RH17].

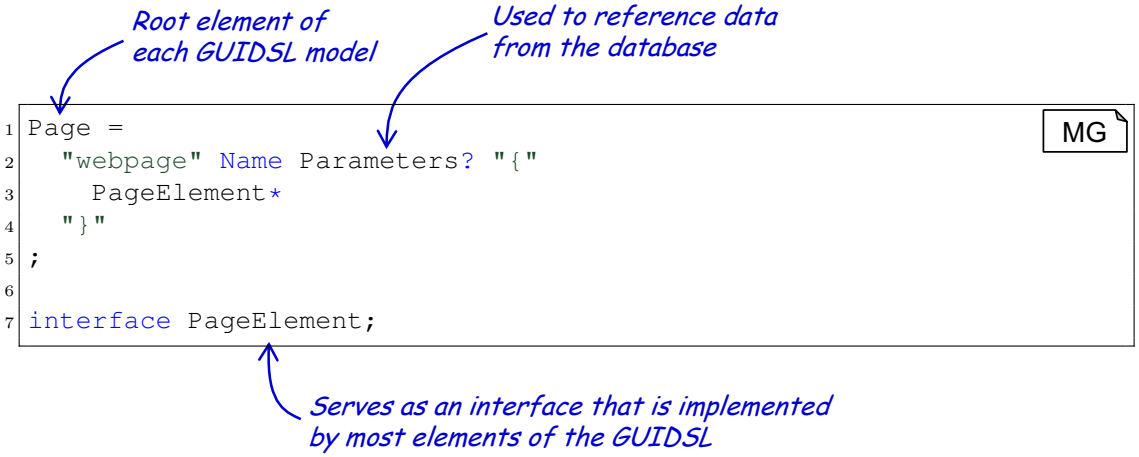


Figure 4.2: Slightly simplified grammar of a page element

`Parameters` in 4.2 stands for another *nonterminal*, which can be optionally used in a page as needed to reference data stored in the database. When parameters are specified, the *GUITool* automatically generates functions that are used to communicate with the backend of the system, so there is no need to add this functionality manually. The modeler can add an arbitrary number of elements that implement the interface `PageElement` between the curly braces of a page. Various elements of *GUIDSL* implement this interface and often use it themselves on the right side of their production. This allows to model an UI of any complexity by composition of multiple *GUIDSL* elements.

4.2.2 Cards

As can be seen in figure 4.1, for example, the layout element `Card` is an element that implements this interface. A simplified grammar for a `Card` can be found in figure 4.3. A `Card` consists of an optional header and a body. Using this element as direct descendant of a `Page` causes the *GUITool* to generate a UI as seen in figure 4.1, with the blue header and a white body. The grammar of the `Card` has been simplified for better readability, but is in fact highly configurable. It is possible to specify, for example, the width of a `Card` and how it should be displayed when using multiple cards on one page. For more information on how to configure *GUIDSL* elements, please refer to the *GUIDSL* Wiki[1].

4.2.3 Labels

As shown in the example model in figure 4.1, a `Label` element is used in the header of the `Card` element. The use of labels in cards is of particular interest when there are multiple cards on a page, as they give the users a first clue about the offered functionality of the

[1] https://git.rwth-aachen.de/macoco/gui-dsl/-/wikis/home

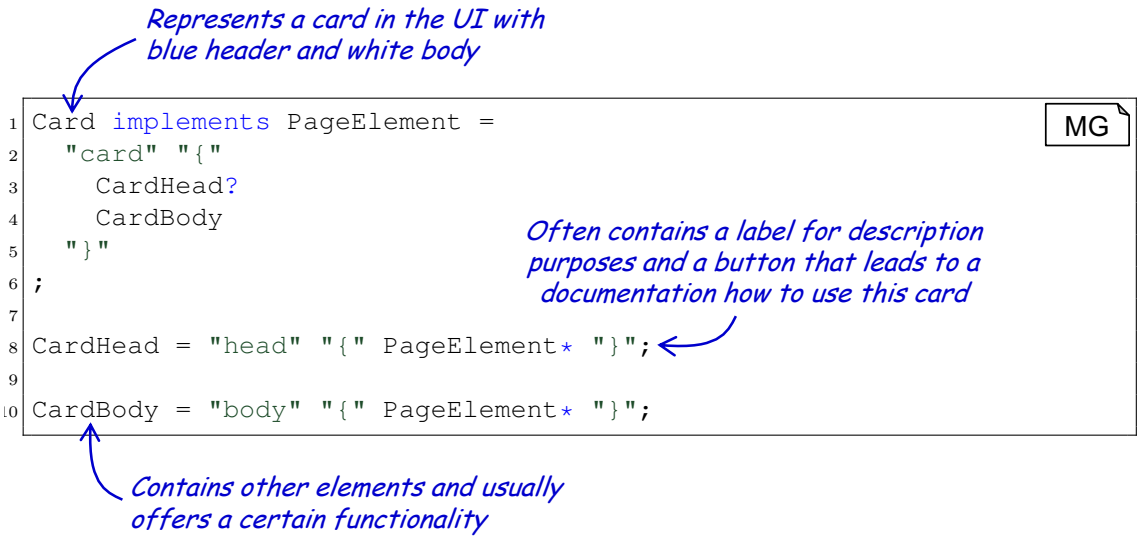


Figure 4.3: Simplified grammar of a card element used for layout purposes in *GUIDSL* models

card, and on the other hand, card labels help users quickly find the right cards on a page. By default, this text is generated on the left side of the header, but as mentioned above, the card is highly configurable and you can also where elements should be displayed inside a card header.

Each label element starts with the keyword `label` and is followed by a concatenation of several string literals, which are merged by the *GUITool* and displayed as one text that belongs together. The *nonterminal* `StringAddition` stands for this concatenation of any number of strings. For example, `"Hello" + "World" + "!"` would be a valid concatenation that could be used for labels. The simplified grammar belonging to this element can be found in figure 4.4. The actual grammar has been simplified for better readability and understanding in such a way that only concatenations of strings are possible. In fact, not only is it possible to concatenate multiple strings with each other, but it is also possible to call methods that may provide data dynamically. But this work is presents the simplified grammar as it can be found in figure 4.4 and further mentions of the `Label` element are based on this simplified grammar.

4.2.4 Tables

Within the body of a `Card` it is possible to use an arbitrary number of page elements of the *GUIDSL* to display data and to enable interaction with them. Among these page elements are data tables, which in the simplest case can be used to display data from the database of the Backend in a table in the UI. The associated simplified grammar of such a simple table can be found in figure 4.5. A table has a unique name that must be written directly after the keyword `datatable` and is used by other *GUIDSL* elements for reference purposes. A table consists of several columns, the name of which is given directly after the keyword `column`. The *GUITool* then generates a header for each column that bears this name.

A table cell is specified for each column, which references data from the database. The

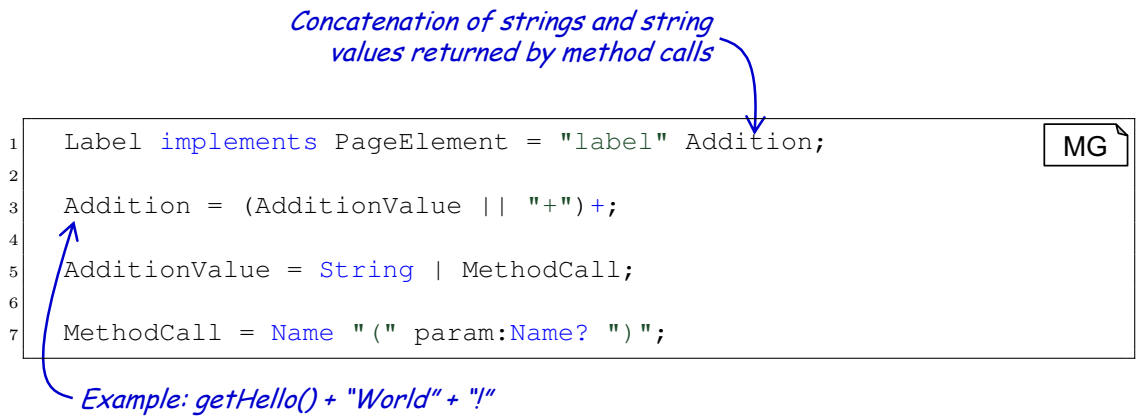


Figure 4.4: Simplified grammar of a label element

simplified grammar for this table element distinguishes between two different types of references. Normal references that have the name `ref` in the grammar refer to normal data that is interpreted as a string in the table, i.e. the value from the database is taken over one-to-one without further transformations. In *MontiGem-based applications*, however, a date is stored internally in the database as an ISO string such as `00271201T000000+0100`, making it necessary to transform the ISO date string into a readable string such as `27.12.2019` before displaying it in the UI. For the use of dates in data tables, the keyword `date` can be used in the *GUIDSL* models and the reference must be surrounded by parentheses, as can also be seen, e.g., in figure 4.1. By using a `DateReference`, the *GUITool* ensures that the ISO string received is transformed into a more user-friendly string before the data is displayed in the table. In data tables there are further such so-called `BuiltInFunctions` with which data can be transformed in various forms. For the sake of simplicity, these have been omitted here.

4.3 Technical Realization

4.3.1 Pages and the ComponentGenerator

For each `Page` node, the *GUITool* always generates two files regardless of which exact elements are used in that page. These are an empty Angular component in the programming language *TypeScript* and a template file in *HTML*, which is used by this component. For an empty page, the *GUITool* already generates the entire frontend architecture of the components, which is necessary to exchange information with the backend using the command pattern. The generation of an empty page is shown in figure 4.6.

In the following, the imports, attributes, and methods that are added to this Angular component skeleton are called elements of the Angular component. Depending on which *GUIDSL* elements are actually used in the page, further Angular component elements must be created and added to the corresponding Angular component. Creating and adding these elements is usually done in a concrete visitor–a class implementing the *GUIDSLVisitor* interface. These visitors use internally the *ComponentGenerator* class, which provide methods that pass the data from the defined component elements to templates and can thus be integrated into the component skeleton.

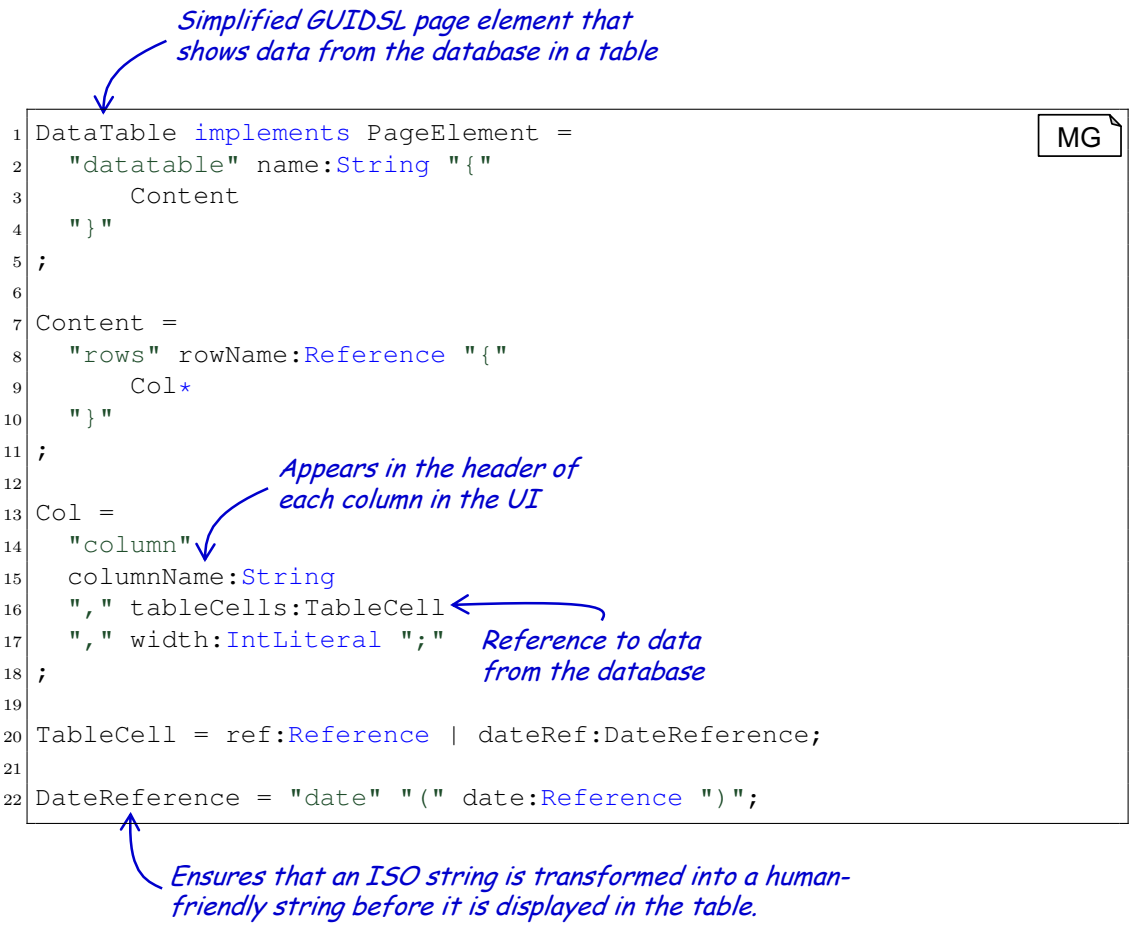


Figure 4.5: Simplified grammar of a table element that can be used in cards to display data dynamically loaded from the Backend's database

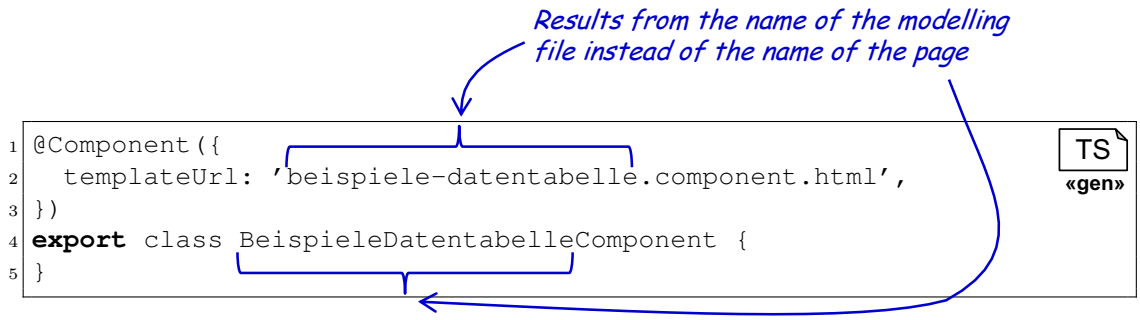


Figure 4.6: The generated Angular component for an empty `Page` node. Even empty pages already contain the architecture to communicate with the Backend.

An example of how the *ComponentGenerator* can be used to generate a method `getGreeting` that returns a string to greet a person can be found in figure 4.7. This figure also shows the generated *TypeScript* method that would be integrated into the Angular component by the component generator, so that it would also be available for other methods in this component. This is indicated in listing 4.1.

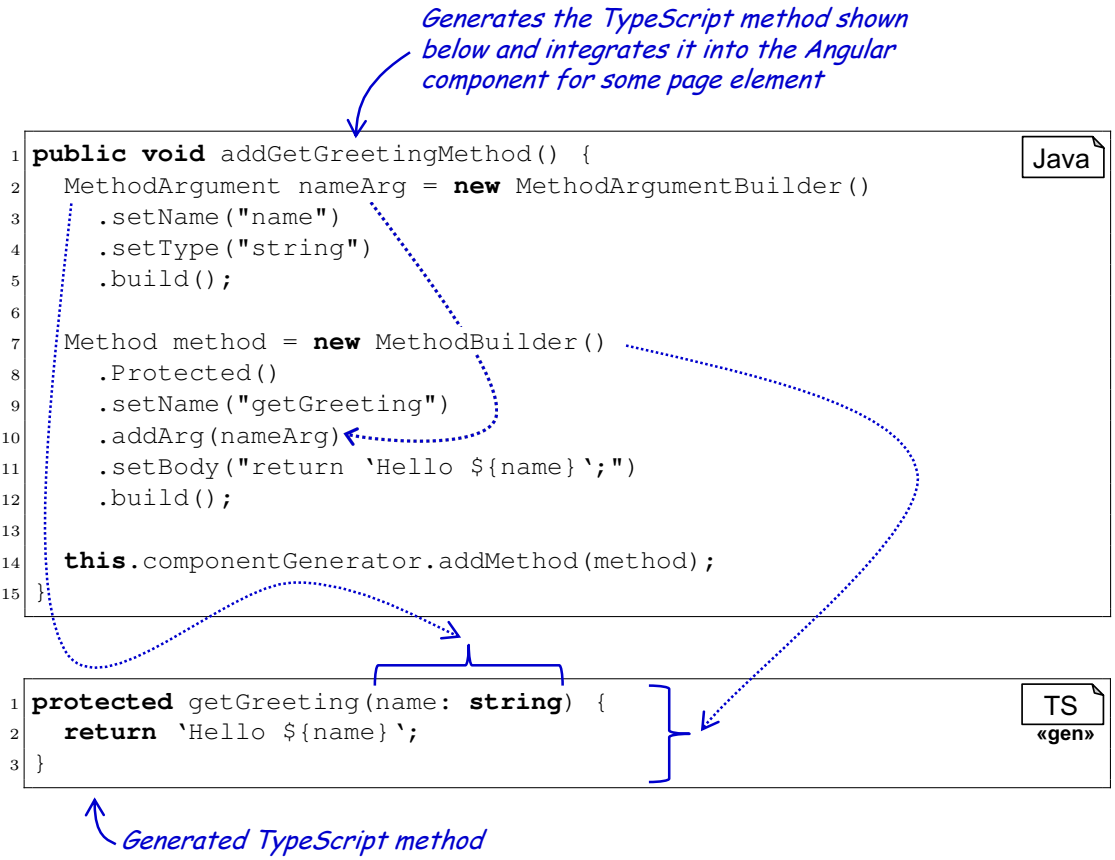


Figure 4.7: Adding methods to the Angular component that is being generated for each `Page` element and thus for each *GUIDSL* model file.

```
@Component({
  templateUrl: 'beispiele-datentabelle.component.html',
})
export class BeispieleDatentabelleComponent {
  protected getGreeting(name: string) {
    return `Hello ${name}`;
  }
}
```

TS «gen»

Listing 4.1: Component elements such as imports, attributes, and methods can be added to the actual Angular component via the ComponentGenerator.

4.3.2 Labels

A label is one of the simplest elements of the *GUIDSL*. Figure 4.3 shows a simplified variant of the template used in *GUIDSL* that is used to generate the HTML code that will be integrated into the template of the corresponding page component. The exact hook mechanism used in the HTMLVisitor, the visitor when generating the HTML template files

for a page component, will not be discussed further in this work, because it did not play a major role in the *internationalization* of *MontiGem* or *MaCoCo*. Details about template hooks, however, can be found in the MontiCore Reference Manual [RH17].

Since an `Addition` node contains an arbitrarily long list of `AdditionValue` nodes, it is necessary to iterate in the template over this list using the `<#list>` directive already introduced in section 3.3.3. Within the loop (*lines 3-8*), a distinction is made between the case that the value is a simple string or a method call, i.e. a node of type `ASTMethodCall`. This distinction can be made by the method `isPresentString()` (*line 3*) generated by MontiCore for the `ASTAdditionValue` class (cf. section 3.3.2). According to the grammar there are only these two cases, so that the first block (*line 4*) handles the strings and `else` block (*lines 6-7*) covers the case that the value is a method call.

If a value is a string, then it is simply output. If it is a method call, however, the *PrintHelper* is used, a helper class that is made available globally in the templates with the variable name `printHelper` using the *GlobalExtensionManagement* (cf. section 3.3.3) provided by MontiCore. See 4.2 for details on how to define a global variable for this helper class. The `printMethodCall` method implemented in this helper class returns a stringified version of the passed node of type `ASTMethodCall`. Although the grammar of the method call can be simplified for labels, so that the entire method call can be accessed as a single string in the templates without using the *PrintHelper*, it is necessary to have more precise control over this node in other parts of the *GUIDSL*. For example, some *GUIDSL* elements must process the method name and the passed parameters of a call separately.

```java
glex.setGlobalValue("printHelper", new PrintHelper());
```

Listing 4.2: In the *GUITool*, an instance of the *PrintHelper* is made globally availaible under the variable name `printHelper`

The whole text including the method calls is displayed side-by-side encapsulated in a HTML element `<macoco-label>`. This element is a component provided by some Component Provider (cf. section 3.2.3) and is part of the *Runtime Engine* (RTE) of *MontiGem*. This Angular component already contains all the necessary default styles, so that the label fits already well into the UI of *MontiGem*.

```ftl
<macoco-label>
  <#list ast.getAddition().getAdditionValueList() as value>
    <#if value.isPresentString()>
      ${value.getString()}
    <#else>
      <#assign methodCall = value.getMethodCall()>
      {{ ${printHelper.printMethodCall(methodCall)} }}
    </#if>
  </#list>
</macoco-label>
```

Listing 4.3: Simplified template used to generate the HTML code for a `Label` element. In this simplified version, configuration options such as setting colors are omitted for readability reasons. The exact details can be looked up in the implementation of *GUIDSL*.

4.3.3 Tables

While the previously discussed label element is one of the simplest element of the *GUIDSL*, the data table element is one of the most complex elements of this *modeling language*. Therefore, it is not possible to cover all generation aspects in this paper and the following explanations are limited to concepts that have played an important role in the *internationalization* of *MontiGem*.

Of particular interest regarding the *internationalization* of the table element was the generation of a method that is being used directly by the *FrontendRTE* of *MontiGem-based applications*. This method returns an array of `TableColumn`, an interface that is already provided by the *FrontendRTE* of *MontiGem* and therefore does not need to be generated by the *GUITool*. An excerpt of this interface can be found in listing 4.4 and the generated method using this interface can be found in figure 4.8. While the interface us already provided by the *FrontendRTE*, the *GUITool* generates for each table a method that returns an array of objects that conform to that interface. Figure 4.8 shows how the *GUITool* would generate this method for the previously presented model example found in section 4.1.

TS «RTE»

```
1 export interface TableColumn {
2     name?: string;
3     prop?: TableColumnProp;
4     displayValue?: (arg: any) => any;
5     width?: number;
6 }
```

Listing 4.4: The *FrontendRTE* provides an interface `TableColumn`. The generated code uses this interface.

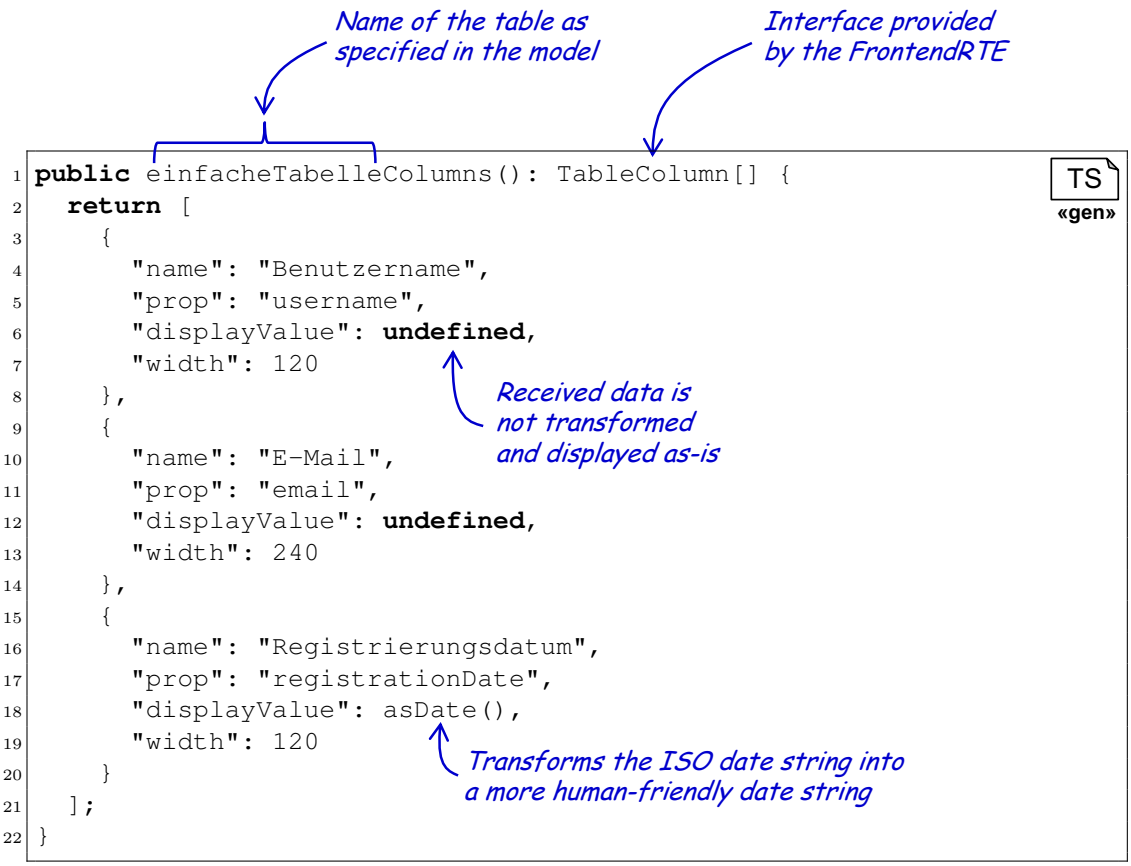
Each `TableColumn` object represents one column of the table. The displayed name of a table column in the UI is specified in these objects via the `name` property. The property `prop` is used by the table component to bind data. With the `displayValue` property a method can be specified that transform the bound data before displaying it. The importance of this functionality is discussed in the following section.

4.4 Elements that had to be internationalized

In the context of this work it is not possible to talk in detail about all elements of the *GUIDSL* that have been internationalised. Instead, based on the example already presented in section 4.1 of this chapter, we will discuss some technical details that are of particular interest for the *internationalization* of the *GUIDSL* elements contained in these model examples.

Listing 4.5 contains the model example found in section 4.1, where elements that required internationalisation are highlighted with a bluish background.

Among the elements that require internationalisation is the `Label` element, which is used in the example in the header of the card. This element obviously needs translation in a localized *MontiGem-based application* as the label content id directly displayed in the UI.




```
1  public einfacheTabelleColumns(): TableColumn[] {
2    return [
3      {
4        "name": "Benutzername",
5        "prop": "username",
6        "displayValue": undefined,
7        "width": 120
8      },
9      {
10       "name": "E-Mail",
11       "prop": "email",
12       "displayValue": undefined,
13       "width": 240
14     },
15     {
16       "name": "Registrierungsdatum",
17       "prop": "registrationDate",
18       "displayValue": asDate(),
19       "width": 120
20     }
21   ];
22 }
```

Figure 4.8: A method is generated from the table modeled in the presented model example (cf. section 4.1), which returns an array of `TableColumn` objects

In addition to the `Label` elements, the names of the columns that are defined for each column needed to get internationalized as these names are displayed in the header of each table column in the UI in order to distinguish them from other columns of the same table.

Mostly all strings in a GUI model had to be internationalized, because they are in some way related to content in the UI. However, interestingly, it is not really necessary to localize all the used strings in a GUI model. This can also be seen in the example in listing 4.5 as not every string has a blue background. The table name, for example, is not displayed in the UI and is only used internally by the *GUITool* during generation to uniquely identify multiple tables and configure the generation process based on this table name. This makes it possible to model several similar tables on one page. Using the table name as unique identifier of a table in a model, the *GUITool* can create code that doesn't include duplicates that would otherwise result in compile-time errors when building the generated *MontiGem* application.

Not only strings had to be internationalized but also other elements of *GUIDSL*, which were not directly defined as strings in the models. As you can see in the example in listing 4.5, a `DateReference` is used, which transforms a date stored as ISO 8601 date format[2] string from the database into a human-friendly string before displaying it in the UI. This transformation is done using a corresponding date format, which can vary greatly depending on the locale a user has selected or was automatically assigned to based on

[2] https://www.w3.org/TR/NOTE-datetime

factors such as the language of his browser or the location in which he is located. For example, the date format `MM/DD/YYYY` is unique to the United States and most countries in Europe use a format similar to `DD/MM/YYYY`. The separators may even vary and may be slashes, dashes or periods. For example, in Germany the format `DD.MM.YYYY` is usually used. A localized version of *MontiGem* has to transform, for example, the ISO 8601 string `00271201T00000000+0100` received from the database into the more human-friendly string `27.12.2019` for the German locale before displaying it in the UI. When selecting the US locale of the application it would be necessary to transform the ISO string into `12/27/2019`. Depending on which local has been selected, *MontiGem* must perform the correct transformation.

GUI

```
webpage beispieleDatentabelle(
  all EinstellungenBenutzerTabelle ut
) {
  card {
    head {
      label "Einfache Tabelle"
    }
    body {
      datatable "einfacheTabelle" {
        rows <ut.alleBenutzer {
          column "Benutzername", <username                , 120;
          column "E-Mail"      , <email                   , 240;
          column "Regi..datum" , date(<registrationDate), 120;
        }
      }
    }
  }
}
```

Listing 4.5: *GUIDSL* Model example from section 4.1, where elements that required internationalisation are highlighted with a bluish background

Chapter 5

Technical Realization of the Internationalizaion of MaCoCo

This chapter discusses some details about the technical realization of the internationalization of *MontiGem* and *MontiGem-based applications* like *MaCoCo*. In the course of this work, both the *GUIDSL* and the Runtime Engines (RTEs) of *MontiGem* were extended in such a way that it became possible to display the system in several locales. Due to the limited length of this thesis details of the extensions of the *BackendRTE* of *MontiGem* are omitted and only extensions of the *FrontendRTE* of *MontiGem* are discussed. Most of the enhancements and changes had to be done on the *FrontendRTE* anyway, since only a small part of the backend had to be internationalized.

This chapter in particular does not deal with every single detail of the technical implementation. Instead, it first presents changes that are of a generic nature and are reused in many other places. Among them is, for example, a new production of the *GUIDSL* grammar, which is reused in most of the other elements to be internationalized. Afterwards, the methods that are generated with the *GUITool* from these elements and how they are used elsewhere will be discussed. The last sections of this chapter shows how the elements from the *GUIDSL* model example already presented in section 4.1 can be internationalized.

Many of the mechanisms discussed in this chapter have been implemented analogously or very similarly in other parts of the system, be it in the *GUIDSL* or the *FrontendRTE* of *MontiGem*. For further details please refer to the respective implementations.

The *GUIDSL* Wiki[1] in the GitLab project has also been extended to include information on internationalization, so that any details on how to model internationalized *GUIDSL* models can be found there.

It should also be mentioned that the code examples presented in this chapter, as in the previous chapters, have been simplified to provide better readability. In addition, details that were not of interest for internationalization have been left out completely, such as the different configuration options of the *GUIDSL* elements mentioned later in this chapter.

[1] https://git.rwth-aachen.de/macoco/gui-dsl/-/wikis/home

5.1 Extraction of translatable content

In the context of this work a *npm script*[2] was made available with which the translatable content could be extracted from the entire source code including the generated part. During the extraction process, the translatable content is added to the translation files whose XLIFF format was already presented in section 3.1.2 as new or modified translation units.

The extraction script can be started with `npm run extract-i18n` and runs in three phases: In the first phase all HTML files are run through and all elements to be translated are collected. In the second phase this is done analogously for all TypeScript files. In the last phase the found units are merged with the already translated units, so that the translation files do not have to be translated every time after executing this script.

If necessary, this tool will also update the additional information intended for translators if there are any changes. Additional information modeled in the *GUIDSL* models (cf. section 5.2.1) is mapped to `note` elements in the *XLIFF* files (cf. section 3.1.2). In the last step, units that were present in the translation files but are no longer used are also deleted. Changes to a text in the source code usually result in completely new translation units. But if these changes only add or delete whitespace, the extraction tool recognizes this and does not create a new translation unit but modifies the existing ones.

5.2 Generating Localization Methods

5.2.1 I18nText Grammar

Most of the strings that can be modeled using the *GUIDSL* had to be internationalized in the course of this work. This has already been pointed out in chapter 4. In the grammar of the *GUIDSL*, a new *nonterminal* `I18nText` has therefore been introduced, which stands for a string that requires translation in the localized version of the system. This production is used by many other *nonterminals* such as the corresponding *nonterminal* for tables on the right side of the production to provide *i18n* functionality. Details on this are given later in this chapter, for example in section 5.3.1. First of all, the corresponding grammar of the `I18nText` *nonterminal* is discussed in more detail, the production of which can be found in figure 5.1.

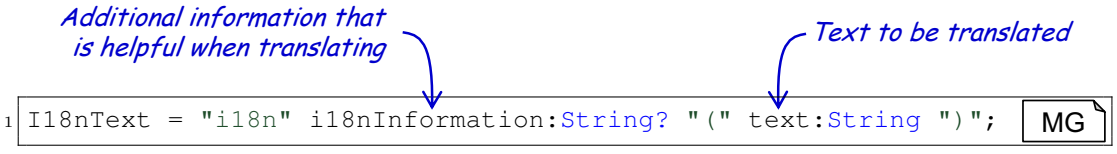


Figure 5.1: Grammar of the `I18nText` *nonterminal*

An internationalized string always starts with the keyword `i18n`. The modeler can then provide additional information, such as a description or a meaning of the text to be translated. This `i18nInformation`, as it was called in the production, might be helpful when translating the text and can be integrated into the decision process of the translator. This information has been marked as optional in the grammar, so it can only be used when

[2] https://docs.npmjs.com/misc/scripts

needed. Details on how to make certain parts of a grammar optional have already been discussed in section 3.3.1. Basically the modeler has to decide if it seems necessary to pass on further information to the translator. He adds this additional information when it seems to him that without this information a decent translation without further context is difficult. Finally, the text to be translated stands after this additional information or after the keyword `i18n` if this has been omitted. The text must be enclosed in round brackets. Two simple examples of how this *nonterminal* can be used in the models can be found in listing 5.1.

```
1 i18n("Benutzername")                                                    GUI
2 i18n "description of the context" ("text to be translated")
```

Listing 5.1: Two simple examples of how the *nonterminal* `I18nText` could be used in the models in appropriate places.

It is also worth mentioning that `i18nInformation` is subject to a more precise syntax, which is not yet part of the grammar in the current implementation and is only checked in a concrete visitor after parsing. The syntax of an information is `description|meaning@@id` where each of these partial information is optional and can be omitted. In future versions of the *GUIDSL* this should be incorporated into the grammar, allowing the mechanism to be more robust.

5.2.2 I18nHelper and Localization Methods

The *GUITool* has been extended in such a way that for each of these `I18nText` elements in the model, methods are generated and integrated in the Angular component of the respective page that allow other parts of the page component to access the localized value of the corresponding source texts. The method names of the generated methods always start with `getLocalized` as shown in the example in figure 5.2. In order to use several of these methods in one component and don't generate code duplicates that would result in compile-time errors, the method name is given a unique suffix, which is marked with `_myIdentifier` in the example. Often this identifier consists not only of the text to be translated for which special characters are replaced or omitted as these are not allowed characters for TypeScript method names but also of the scope in which the translation unit appears.

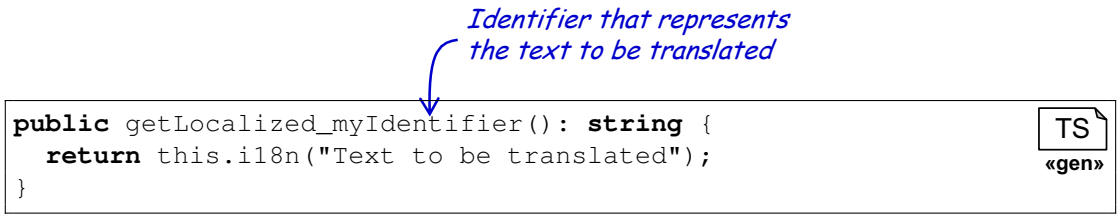


Figure 5.2: Example of a generated getLocalized method that returns the translated string in the language selected by the user

As already explained in section 4.3.1, the *ComponentGenerator* class is used in the *GUITool* to integrate the created methods into the Angular component. This is used in the

I18nHelper, a helper class that encapsulates generic i18n concepts, as shown in listing 5.2. Basically this helper class is used according to the following scheme: In the *TSVisitor*, which is responsible for generating the TypeScript code, the *i18nHelper* class is first initialized and is given the *ComponentGenerator* that has already been made available for the *TSVisitor*. The AST is traversed by the *TSVisitor* and, in the case of occurrences of `I18nText` elements, this is passed on to the *I18nHelper* together with a calculated identifier for this text. The `addElement` method provided by the *I18nHelper* can be used for this. After all `I18nText` nodes have been traversed, the `generate` method provided by the *I18nHelper* can be used, which first creates the localization methods based on the added *i18n* elements and then integrates them into the corresponding Angular component.

Java

```java
public class I18nHelper {
  private HashMap<String, ASTI18nText> i18nElements = new
      HashMap<>();

  private ComponentGenerator componentGenerator;
  public I18nHelper(ComponentGenerator componentGenerator) {
    this.componentGenerator = componentGenerator;
  }

  public void generate() {
    for (String id : i18nElements.keySet()) {
      Method getLocalized = new MethodBuilder()
          .setName("getLocalized" + id)
          .setReturnType("string")
          .setTemplate("gui.ts.i18n.Element")
          .addTemplateArgs(i18nElements.get(id), id)
          .build();
      componentGenerator.addMethod(getLocalized);
    }
  }

  public void addElement(String identifier, ASTI18nText text) {
    this.i18nElements.put(identifier, text);
  }
}
```

Listing 5.2: Implementation of the *I18nHelper*

It is important that the `generate` method is only called when all elements of the model have already been traversed. Otherwise the *GUITool* would generate code duplicates. In order to avoid such duplicates, concepts related to AST traversing were used which were already explained in section 3.3.2. The `endVisit` method presented therein which is also generated for the `ASTPage` node is of particular interest for the integration of the `getLocalized` methods into the Angular page component. Since the page node is the root node of the AST the `visit` method of the `ASTPage` node is always executed first in a concrete visitor and the corresponding `endVisit` method is ececuted after all other elements have already been visited. The `endVisit` method is therefore a good candidate for executing the `generate` method provided by the *I18nHelper*, because at this point you can be sure that all other AST nodes have already been traversed for the corresponding model and therefore already all `ASTI18nText` nodes have been handed over. Figure 5.3 illustrates this again with a code example.

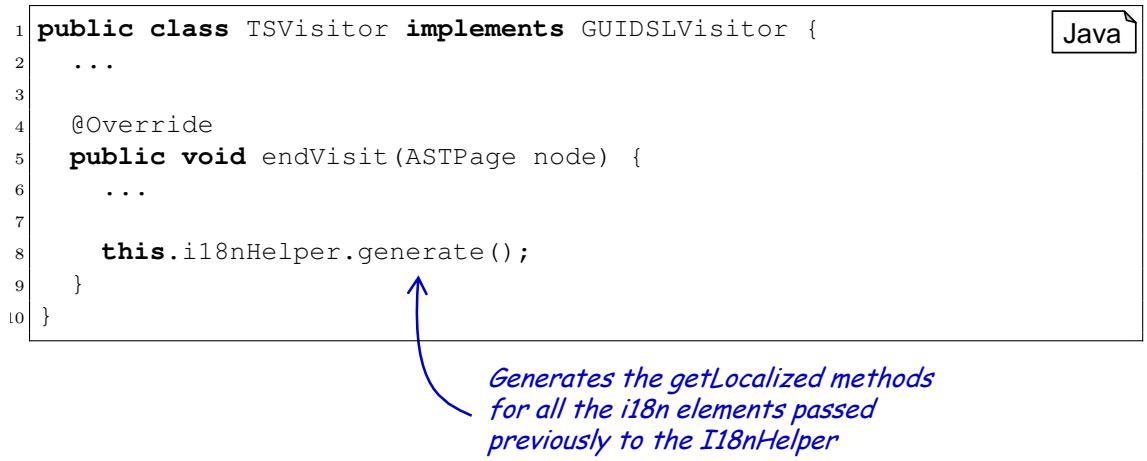


Figure 5.3: `getLocalized` generation after all AST nodes have been visited

The generated `getLocalized` methods can then be used elsewhere in the Angular page component to get access to the translated content in the language selected by the user. Listing 5.3 shows, e.g., for the method generated for the table from the model example (cf. section 4.1) how these `getLocalized` methods replace the old hardcoded strings. As can be seen in the listing the name property now uses the `getLocalized` method instead of a hardcoded column name string (cf. 4.8) that was taken over one-to-one from the model example, so the UI now displays the translated column name instead of the untranslated one.

```
public einfacheTabelleColumns(): TableColumn[] {
  return [
    {
      "name": this.getLocalized_ut_alleBenutzer_benutzername(),
      "prop": "username",
      "displayValue": undefined,
      "width": 120
    },
    {
      "name": this.getLocalized_ut_alleBenutzer_e_mail(),
      "prop": "email",
      "displayValue": undefined,
      "width": 240
    },
    {
      "name": this.getLocalized_ut_alleBenutzer_regi..datum(),
      "prop": "registrationDate",
      "displayValue": asDate(),
      "width": 120
    }
  ];
}
```

TS «gen»

Listing 5.3: Internationalized version of the method that is generated for the modeled table in the internationalized GUIDSL model example (cf. section 4.1 and 5.3.3)

5.2.3 Template for Localization Methods

```ftl
${tc.signature("i18nElement", "id")}
<#assign list = guiHelper.decodeI18nInformation(i18nElement)>

return this.i18n(
  <#if i18nElement.isPresentI18nInformation()>
    {
      value:  "${i18nElement.getText()}",
      <#list list as value>
        <#if value_index = 0>
          meaning:
        <#elseif value_index = 1>
          description:
        <#else>
          id:
        </#if>
        "${value}",
      </#list>
    }
  <#else>
    "${i18nElement.getText()}"
  </#if>
);
```

Listing 5.4: FTL template for the `getLocalized` methods

```ts
return this.i18n({
  value: "Benutzername",
  meaning: "myMeaning",
  description: "myDescription",
  id: "myId",
});
```

Listing 5.5: Example of generated method body for some `getLocalized` method

In this section we will take a closer look at the FTL template used in the *GUITool* that describes how the method bodies of the generated `getLocalized` methods look like. As can be seen in the listing 5.2, when generating this method, two additional arguments are passed to the template named `gui.ts.i18n.Element`. This is the `i18nElement` over which the loop is currently iterating and the unique identifier associated with this element. By using the `signature` method provided by the `TemplateController`, these passed objects can be accessed in the template (cf. section 3.3.3). In the template, whose code can be found in listing 5.4, it is checked whether the GUI modeler has specified additional information for the translator. If this is the case, then this information is returned by the generated method as object together with the original text as prop-

erty, as in the generated example in listing 5.5. As discussed in the previous section, the `i18nInformation` is a string. Using the `decodeI18nInformation` method provided by the `GUIGeneratorHelper`, the information can be transformed into this object. This method always returns an object that is supported by the i18n polyfill.

5.2.4 Injecting I18n Service

For the internationalization of *MontiGem* whose *FrontendRTE* is based on Angular, the Angular Internationalization Tools[3] could be used, which are already built natively into Angular and already allowed to read and display translations stored in the *XLIFF* files (cf. section 3.1.2) from the *HTML* files. Also these tools already made it possible to extract the texts marked with `i18n` from the *HTML* files, so that they could be automatically inserted into the translation files. However, it is not possible to read translations from the TypeScript files with these tools. Therefore a polyfill is used in addition to these tools, which makes this functionality available in the TypeScript files. This polyfill is the package `ngx-polyfill`[4], which must be injected into the Angular components as a service.

The *ComponentGenerator* has been extended by a method `useI18n`, which ensures that the polyfill is first imported into the corresponding component and then adds a corresponding constructor argument so that it can be injected into the component by the dependency injection mechanism used in Angular. The implementation can be found in figure 5.4. This method is called in the *TSVisitor* when an internationalized element is traversed. See listing 5.6 for details. This ensures that the polyfill is only used and imported if the model actually contains internationalized elements.

```java
public class TSVisitor implements GUIDSLVisitor {
    ...

    @Override
    public void visit(ASTI18nText node) {
      this.componentGenerator.useI18n();
    }

    @Override
    public void visit(ASTLocalizedAdditionValues node) {
      this.componentGenerator.useI18n();
    }
}
```

Listing 5.6: Only provide additional *i18n* functionalities from external libraries when the modeler marked elements in the model with the `i18n` keyword.

[3] https://angular.io/guide/i18n
[4] https://github.com/ngx-translate/i18n-polyfill

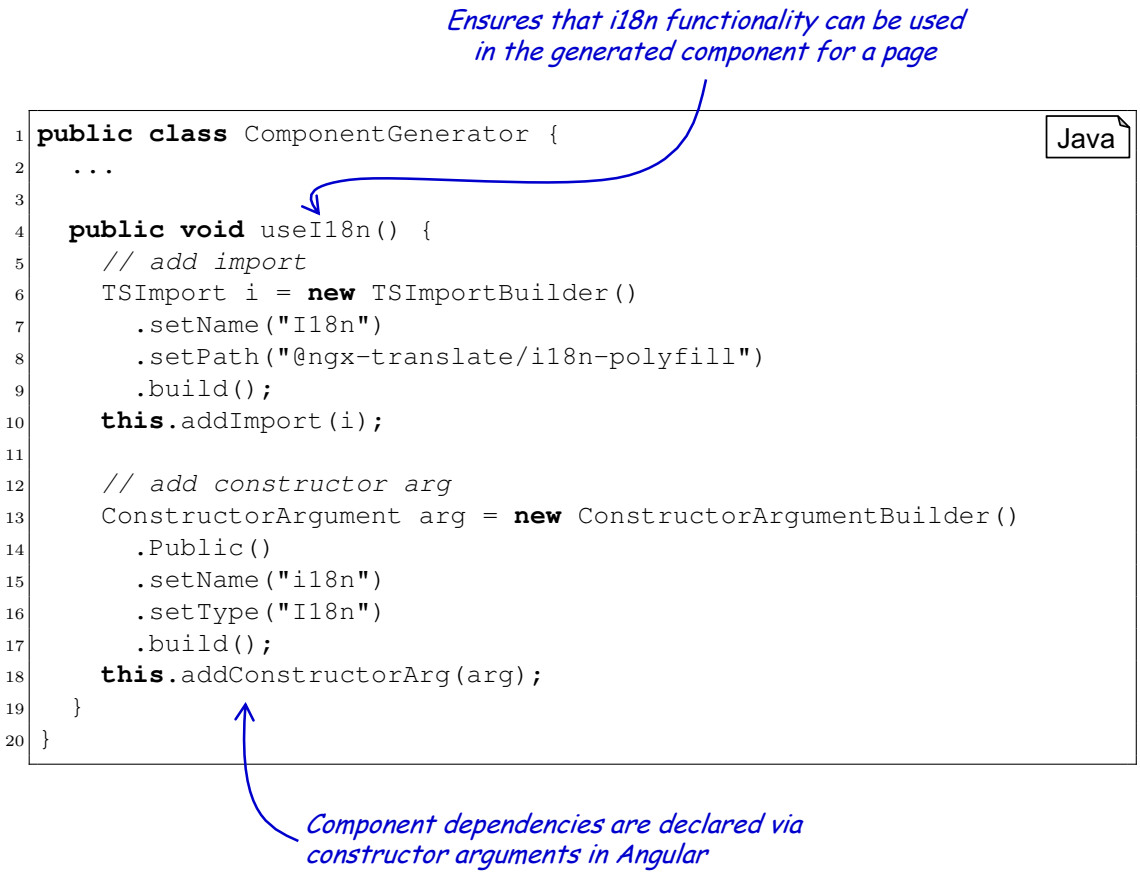


Figure 5.4: The implementation of the `useI18n` method used to make additional *i18n* functionalities from external libraries available in the generated Angular page component.

5.3 Internationalization of Tables

5.3.1 Internationalized Column Name

While the previous sections explained concepts of generic nature, this section gives an example of how these concepts can be used to implement an internationalized variant of the table element that has been previously introduced in the GUI model example found in section 4.1.

First of all, the grammar of the table has been extended in such a way that the modeler now has the possibility to use the `I18nText` element for the column names of the table. This has been implemented as an alternative to the normal text that should not be displayed translated in the application, so that the modeler can decide whether it is necessary to display a translated column name in the generated system or not. A simplified grammar of the corresponding extension of the *nonterminal* `Col` used in tables is given in listing 5.7.

```
Col =
  "column"
  (columnName:String | i18nColumnName:I18nText)
  "," tableCells:TableCell
  "," width:IntLiteral ";"
;
```
MG

Listing 5.7: Simplified internationalized grammar of a table column.

In the `TSVisitor` the `visit` method of the `ASTMacocoDataTable` node had to be extended so that every `I18nText` node used in the table together with its asssociated identifier is passed to the `I18nHelper` instance so that it can generate the `getLocalized` methods from it. By doing so, other places in the system can access the localized value of the original text using the generated `getLocalized` methods. The exact implementation can be found in listing 5.8, where the existing `getIdForColumn` method provided by a helper class is used to get an identifier from the column name to be translated.

```java
@Override
public void visit(ASTCol node) {
  ...

  if (col.isPresentI18nColumnName()) {
    this.i18nHelper.addElement(
      helper.getIdForColumn(col, node),
      col.getI18nColumnName());
  }
}
```
Java

Listing 5.8: By traversing the table columns the units to be translated are passed to the I18nHelper, so that it generates the getLocalized methods.

The corresponding HTML template was then adapted to use the generated getLocalized method if the modeler decided to use the `I18nText` element in the model.

5.3.2 Internationalized DisplayValue Methods

As mentioned in section 4.4, it is also necessary to transform certain data before displaying a table. This transformation depends on the language that a user has selected in his preferences or that is automatically assigned to him. The transformations are carried out via the methods that are transferred as `displayValue`. To provide the internationalization functionality, the corresponding transformations in the *FrontendRTE* have been modified in such a way that the locale selected by the user is taken into account during the transformation. A modification of the `asDate` transformation method can be found in listing 5.9, which was implemented analogously or similarly for the other transformation methods. With the global variable `CurrentLocale`, which was set in the frontend's main module, the `AppModule`, the current locale of the user can be accessed in the transformation methods. Usually it suffices to pass the currently active locale to the `momentJS` [5] library,

[5] https://momentjs.com/

which is used in *MontiGem* for formatting date objects. Based on this locale `momentJS` executes the correct transformation.

```
import { CurrentLocale } from "@root/app.module";

export function asDate(): (value: any) => string {
  return function (date: Date | null): string {
    if (data && ((date + '') !== 'Invalid Date')) {
      return moment(date)
        .locale(CurrentLocale)
        .format(DEFAULT_DATE_FORMAT);
    }
    return '';
  }
}
```

TS «RTE»

Listing 5.9: Internationalized transformation method used in tables to convert ISO date strings into an readable date string format that is adapted to the currently active locale

5.3.3 Internationalized Example

Figure 5.5 shows the version of the example from section 4.1 where the table will be displayed translated in the UI of *MontiGem*.

```
webpage beispieleDatentabelle(
  all EinstellungenBenutzerTabelle ut
) {
  card {
    head {
      label "Einfache Tabelle"
    }
    body {
      datatable "einfacheTabelle" {
        rows <ut.alleBenutzer {
          column i18n("Benutzername"), <username              , 120;
          column i18n("E-Mail")      , <email                 , 240;
          column i18n("Regi..datum") , date(<registrationDate), 120;
        }
      }
    }
  }
}
```

GUI

Column names marked with "i18n" are shown translated

Figure 5.5: Model example presented in section 4.1 in which the table is internationalized

Chapter 6

Localization Tool

6.1 Usage

As part of this work, a localization tool was developed that can be used by translators to translate the units of the frontend. The tool was integrated into *MontiGem* and *MaCoCo* as a separate page and consists of several cards, which provide valuable statistics or allow the input of translations. The whole page can be seen in figure 6.1. In the following, individual functions of this tool are described in more detail.

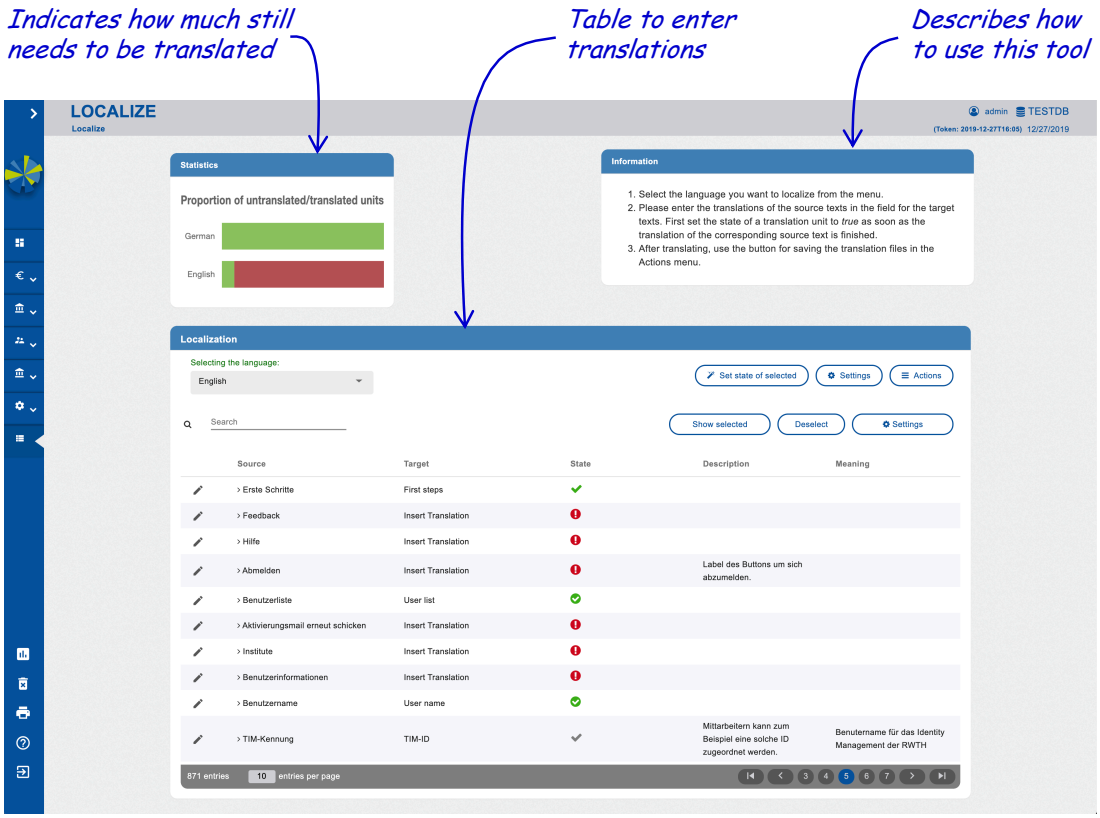


Figure 6.1: Overview of the translation tool

6.1.1 Usage of the Translation Table

The main card contains an editable table of all translation units and is shown as an example in figure 6.2. To start localizing, the translator must first select the language for which he wants to enter translations using the drop-down menu in the upper left corner of the card. The translator can then start entering the translations by clicking on the corresponding pencil in the table row to switch to edit mode. The unit can then be stored temporarily by pressing the disk symbol. By pressing this button the translations are not yet saved in the actual translation files. Only when the translator actively notifies the system that he has finished his translation session, the translations are actually taken over to avoid constant reloads when translating the frontend. Translators can end their translation session by using the save actions that can be found in the actions menu in the upper right corner of the translation card. If *MontiGem* or *MaCoCo* has been started in live development mode, the page will then be reloaded with the new translations. This allows the translator to quickly check how the translations actually look like in the application.

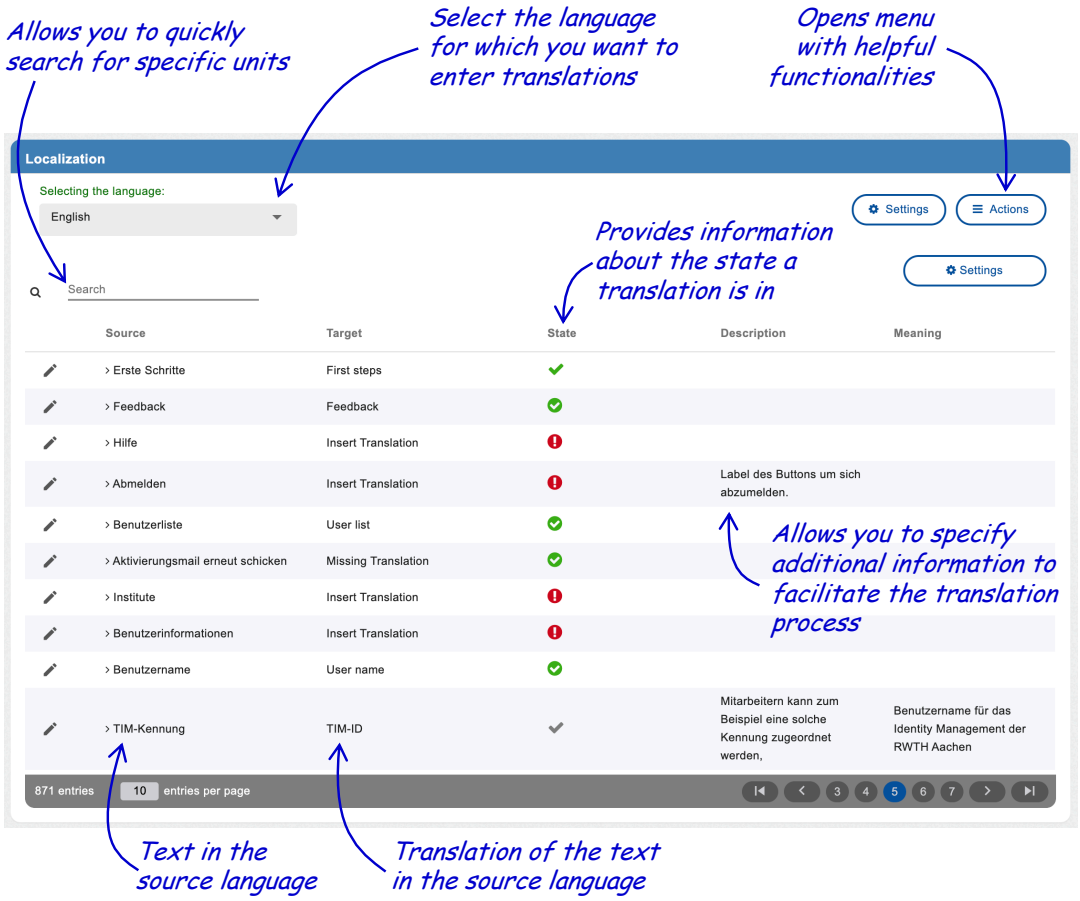


Figure 6.2: Translation card of the localization tool page of *MontiGem* and *MaCoCo*

6.1.2 Usage in collaboration environments

One of the goals in developing this tool was to make it easy to use in a collaborative environment. Therefore, the tool provides the possibility to specify the different translation

states, which were already presented in section 3.1.2. The use of these translation states enables a good collaboration between software developers and translators. Besides the columns of the translation table, which display the text in the original language and the column in which translations can be entered, the table also has an additional column which, in the displaying mode, provides information about the current state of the corresponding unit and, in edit mode, allows this state to be changed. The actual identifiers of the states have been replaced with symbols and colors have been given so that the user of this tool can easily distinguish between these different translation states. Which states the symbols belong to can also be seen in the system. See for example figure 6.5.

As already explained in section 3, it is possible to provide a translation unit with additional information that may be useful when translating. If a software developer makes use of this functionality, this information is displayed in the translation table in the columns `Description` and `Meaning`. If this information is not provided by the software developer in the source code, these fields remain empty. In the example in figure 6.2, the translation unit with the source text `Abmelden` in German has the description attached that this is the label of a button with which a user can log out.

6.1.3 Draft Mode

In the settings menu you can select how untranslated units should be displayed in the application. Figure 6.3 shows this menu in detail. Currently there are two options: Either untranslated texts are displayed in the source language or all untranslated units are replaced with the placeholder `Translation Missing`. In production you should usually use the first option, because by using a placeholder it is no longer possible for the user to get a grasp what this text in the context really means. However, the use of placeholders has enormous advantages in the development process of the app, as they allow you to quickly identify untranslated units in the application. But the disadvantage of this variant is that you have to check the source code to see which string it actually replaces. To improve this, in the future tooltips could be implemented for the placeholders that show the text in the source language. So it would no longer be necessary to look into the source code if you find one of these placeholders in the application.

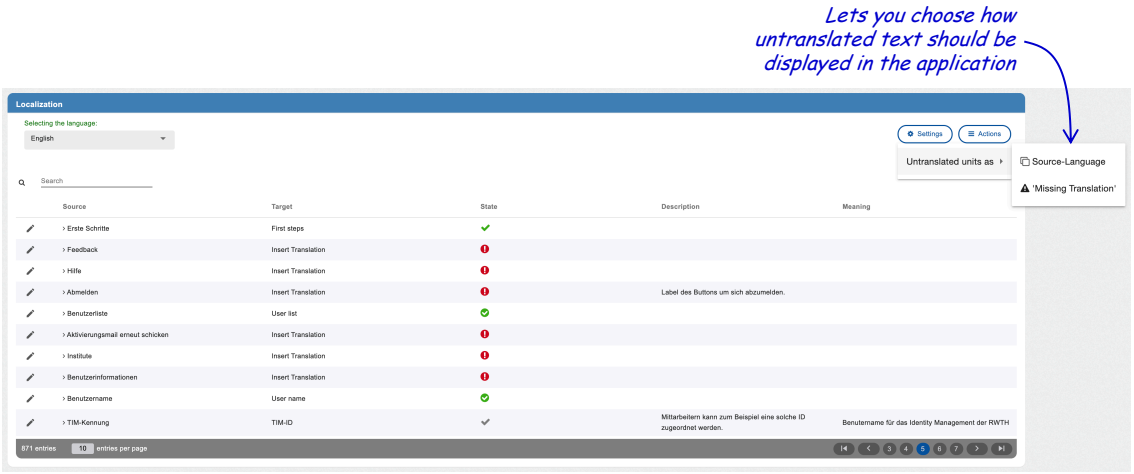
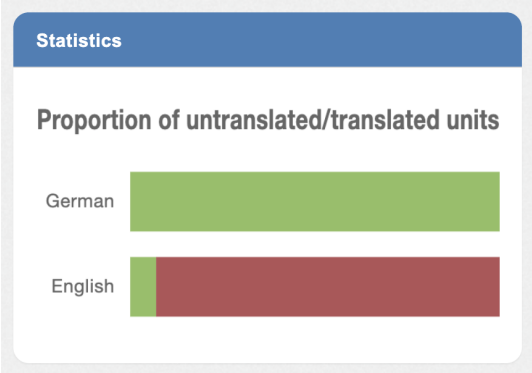


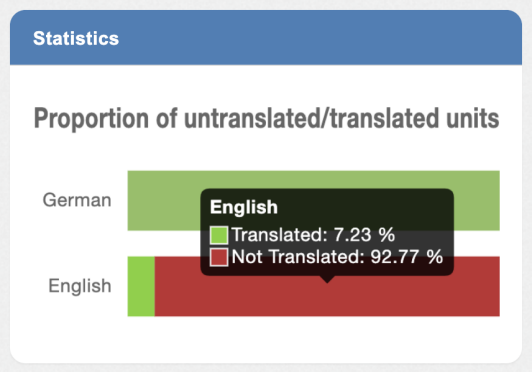
Figure 6.3: In the settings you can choose how untranslated texts should be displayed in the application.

6.1.4 Statistics

In the tool there is also a card that displays useful statistics about the translation status of the application. In this card, as shown in figure 6.4a as an exemplary figure, a bar chart is displayed showing the percentage of translated and untranslated units for each of the supported languages of the application. In this statistic, units are only considered translated when they are actually in the translated state and thus displayed in production. In all other cases the statistics summarize the units under the category of untranslated units. The bar chart already gives a rough overview of how many units still need to be translated. If you hover the mouse over one of the bars of the barchart, as shown in figure 6.4b, a tooltip is displayed with detailed information about the ratio of translated and untranslated units.



(a) The chart gives a rough overview of how many units still need to be translated. *Green* represent already translated units and *red* represents untranslated units.



(b) Detailed information is displayed when the translator hovers one of the bar charts. In this example, the bar chart for the English language is hovered.

Figure 6.4: Exemplary statistics of the translation status

6.1.5 Additional useful functionalities

In the translation table the functionality to select rows by clicking on them has been added. After the selection, these rows get a bluish background to distinguish them from the unselected rows. If the user has selected multiple rows in the translation table, a new button appears above the translation table with which he or she can open a menu to change the status of multiple units at once. This is especially useful in a collaborative environment where a reviewer checks one page of the translation table and then, with a few mouse clicks, can quickly change the status of several of these translation units after checking without having to switch to edit mode for each of them. So this functionality saves valuable time, especially for reviewers. The menu button can be seen in figure 6.5.

6.2 Technical Realization

This section covers some of the technical details of how the localization tool was implemented.

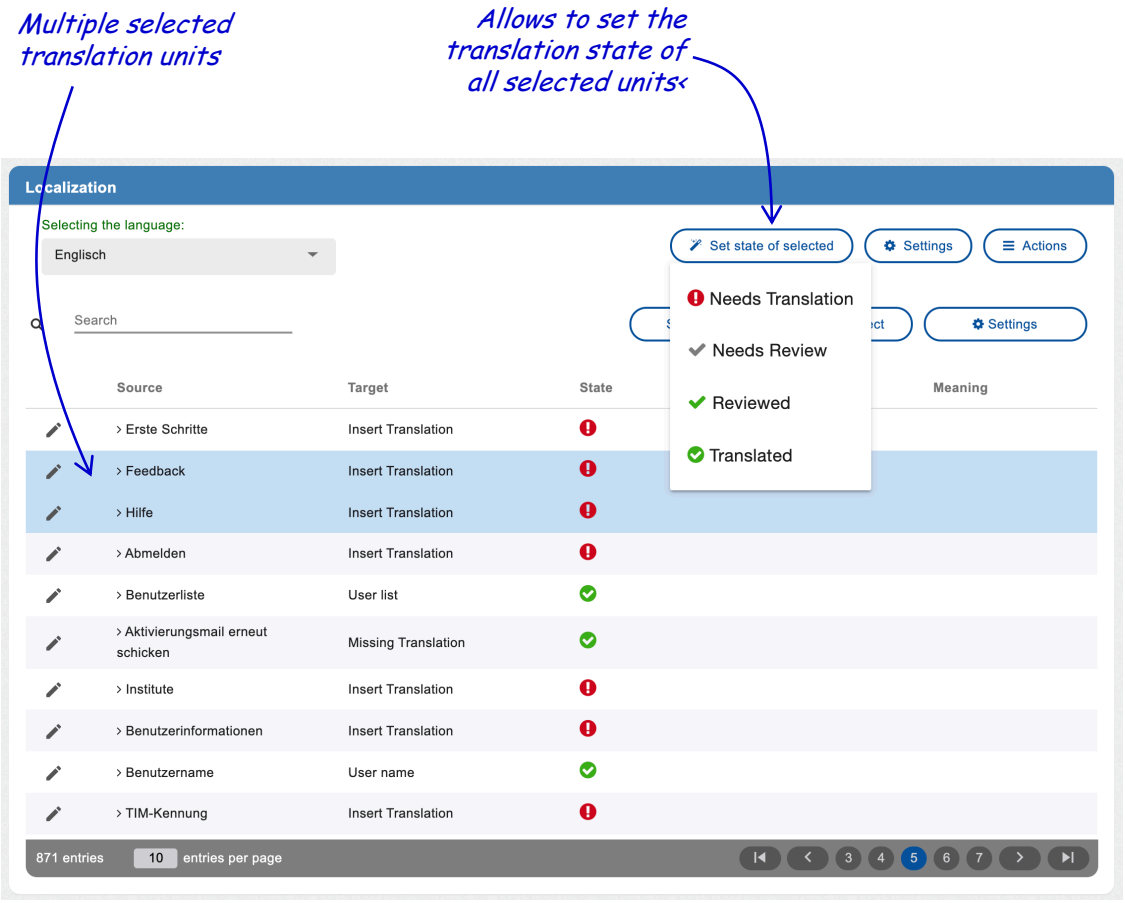


Figure 6.5: The translation tool allows you to change the state of multiple translation units at once

As explained in the previous sections of this chapter, the localization tool can be used to edit translations for the frontend. Internally, these translation units are managed and stored in *XLIFF* files (cf. section 3.1.2). Depending on which language the translator has selected via the drop-down menu, the corresponding *XLIFF* file is loaded into the frontend. This is done using *WebPack's RawLoader* which allows to import whole files as strings. This is possible without any security concerns, as this tool is usually not taken over to the production version of the app as this tool is only useful for the development of the application. Therefore no checks have been implemented to verify that the *XLIFF* file loaded is a valid file. After the Frontend has access to the *XLIFF* file as a string, *fast-xml-parser*'s [1] parser functionality is used to transform this string into an equivalent tree structure representing the *XLIFF* file allowing it to get traversed. The use of this XML parser is possible because *XLIFF* is an XML based file format, as explained in section 3.1.2. A difficulty with parsing of the *XLIFF* file was that the `<source>` and `<target>` elements can contain additional elements, which are used for interpolations or other HTML tags. These elements had to be interpreted as strings so that they could be displayed correctly and edited in the data table. Therefore the parser process was configured in a way that the whole content of the `<source>` and `<target>` elements was interpreted as string.

[1] https://www.npmjs.com/package/fast-xml-parser

After the *XLIFF* file was available as a structure, it was sufficient to traverse over it in order to offer the functionality described above. Internally, all the logic that made it necessary to traverse this tree structure was encapsulated in a separate Angular service called `LocalizeService`. The Angular component responsible for the *Localization Tool* could then use this service and call the correct method of the service depending on the functionality the user executed, so that only a few times in the component itself the *XLIFF* structure needed to get traversed.

Most of the logic for the table was already built into *MontiGem* and could be reused in this work to develop this tool. Also the *MontiGem's* BarChart component could be reused. In developing this tool it was only necessary to traverse the *XLIFF* structure and collect the meta data in a variable that implements the `ITranslationMetaData` interface. This interface can be found in listing 6.1. After collecting the meta data, all that was left to do was to transform the data into the format defined in *MontiGem's FrontendRTE* that is used to display data in the bar chart.

TS

```
export interface ITranslationMetaData {
  sourceLanguage: String,
  targetLanguage: String,
  numberOfUnits: Number,
  numberOfTranslatedUnits: Number,
  numberOfUntranslatedUnits: Number,
}
```

Listing 6.1: Interface implemented by a variable in the `LocalizeService` to store the metadata of an *XLIFF* file.

Chapter 7

Summary

In this thesis it was worked out how to internationalize software systems that are developed model-driven (cf. section 3.2). The thesis first introduces some basics of modeling languages (cf. section 3.2.2) and software generators (cf. section 3.2.3) and shows how such tools can be created using MontiCore. The remaining chapters show in detail how the GUIDSL (cf. chapter 4), a modeling language used for the generation of the UI of MontiGem (cf. chapter 1) and MontiGem-based applications such as MaCoCo, was internationalized (cf. chapter 5) and which further changes to the runtime engines of MontiGem were necessary. With the results of this work it is now possible to translate MontiGem or have it translated by external professional translators. A tool was developed to work on the translations of a MontiGem application in a collaborative environment (cf. chapter 6). Depending on the language the user has selected, the system will display the system in the correct language.

Bibliography

[ang] Angular, internationalization (i18n). https://angular.io/guide/i18n#nesting-plural-and-select-icu-expressions. [Online; Accessed: 2020-01-07].

[cio] Cio wiki, enterprise information systems. https://cio-wiki.org/wiki/Enterprise_Information_System_(EIS). [Online; Accessed: 2020-01-07].

[for] Formatjs message syntax. https://formatjs.io/guides/message-syntax/. [Online; Accessed: 2020-01-07].

[icu] Icu user guide. http://userguide.icu-project.org/formatparse/messages. [Online; Accessed: 2020-01-07].

[Las] Alex Lashkov. How-to: Software internationalization done right. https://bit.ly/375PQ7x. [Online; Accessed: 2020-01-07].

[mad] Madcap software blog, what to keep in mind when using variables in your translation workflow. https://bit.ly/2StcXUe. [Online; Accessed: 2020-01-07].

[med] Medium.com, introduction to internationalization (i18n). https://bit.ly/2ujikgZ. [Online; Accessed: 2020-01-07].

[Naz17] Pedram Mir Seyed Nazari. *MontiCore: Efficient Development of Composed Modeling Language Essentials*. Shaker Verlag, 2017.

[RH17] Bernhard Rumpe and Katrin Hölldobler. Monticore 5 language workbench. edition 2017. 2017.

[VS04] R Viswanadha and M Scherer. Localizing with xliff & icu. In *26th Internationalization and Unicode Conference*, 2004.

[wika] Wikipedia, betriebliches informationssystem. https://de.wikipedia.org/wiki/Betriebliches_Informationssystem. [Online; Accessed: 2020-01-07].

[wikb] Wikipedia, internationalization and localization. https://en.wikipedia.org/wiki/Internationalization_and_localization. [Online; Accessed: 2020-01-07].

[wikc] Wikipedia, locale (computer software). https://en.wikipedia.org/wiki/Locale_(computer_software). [Online; Accessed: 2020-01-07].

[xli] Xliff 1.2 specification. http://docs.oasis-open.org/xliff/v1.2/os/xliff-core.html. [Online; Accessed: 2020-01-07].