Techniques For Lightweight Generator Refactoring

Holger Krahn and Bernhard Rumpe

Institute for Software Systems Engineering
Technische Universität Braunschweig, Braunschweig, Germany
http://www.sse.cs.tu-bs.de

Abstract. This paper presents an exercise to facilitate refactoring techniques not only on generated code, but also on generator templates by reusing existing refactoring techniques from the target language. Refactoring is particularly useful if not only the generated classes but also the template defining the result of the code generator can be adapted in a uniform treatment. We describe a simple demonstration prototype that illustrates our approach. The demonstration is based on the idea to define the templates for code generation themselves as compilable and therefore refactorable source code. However, this limits the directives embedded in the template used for code generation, as we have to embed those as comments. We explore how far this approach carries and where its limits are.

1 Introduction and Problem Statement

Code generation avoids repetitive and tedious programming tasks and helps to improve code quality as it "reuses" code from templates [13]. When code generators are used in agile projects, a subtle problem occurs: hand-coded source code is frequently changed using existing refactoring [7, 11] tools. To keep all existing code consistent, usually tools like Eclipse [6] also refactor the generated code. However, template and generated code are then not consistent anymore. So far two ways to handle this exist: either the generation is regarded a non-repeatable one-shot and the template is never reused, or the changes are manually applied to the template. In the latter case, it is important to ensure that generated code is in sync again, which forces re-generation and manual checks if the result is consistent. However, this approach is time consuming and therefore hinders an agile refactoring of software. This problem is illustrated further in Figure 1, where the following steps are applied within a software system that makes use of code generation.

1. A generator takes a model, in the following text also called *data*, and a given template file to generate the code. The results of this process are generated source files that typically interact with handwritten source code e.g. comprising technical interfaces, specific algorithms or reusable framework components.

[KR06c] H. Krahn, B. Rumpe.
Techniques For Lightweight Generator Refactoring.
Proceedings of Summer School on Generative and Transformational Techniques in Software Engineering
(LNCS 4143)
Springer 2006.
www.se-rwth.de/publications

2. The source code is refactored. If the generated source code contains references like method calls or variable instantiations that are defined in the hand-written code, changes are adopted automatically by the refactoring engine. This process is usually called *updating references* [11] in the source code that is not directly refactored.
3. The model or data is changed manually, which results in the necessity to generate the classes again. One efficiency problem is that now the changes done by the refactoring engine in step 2 are discarded and the resulting source code is not necessarily interacting correctly with the generated code anymore.

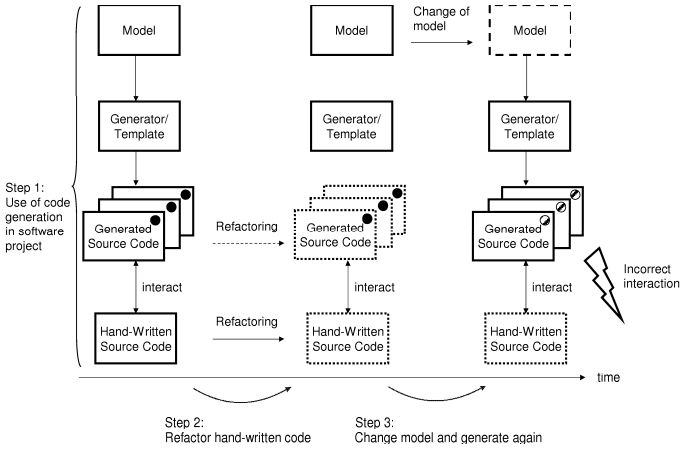

Fig. 1. Risk of incompatibility when refactoring generated source code.

The problem described hadn't occurred if the template would have been refactored in the same fashion as the refactoring engine updates the generated classes in step 1. Our approach allows to write templates in a form that allows for automatic refactoring by any code refactoring engine that updates references and preserves comments. Our experiments show that e.g. the built-in Eclipse refactoring engine is sufficient for this purpose.

Various ways of code generation are already published. For a survey of the most common approaches see for example [10]. Most similar to our approach are template-based code generators like Velocity [1] which is e.g. also used in Poseidon [2]. The mainstream of these approaches uses a template that is a combination of pieces of the target language that includes "holes" where pieces of the code generator language are included. For example, if one wants to generate Java classes with Velocity, the template contains a number of Velocity tags embedded in a Java frame. Such a file is in usually not compilable, because the Velocity tags are not valid parts of the Java language. From a practical point of view, a refactoring engine thus simply ignores the template files and is therefore incapable of updating references in them. A heavyweight solution would be to enhance the refactoring engine to actually understand and transform a template

as well. This is sophisticated work to do and even though the number of used templates will probably increasing a lot in future software projects, it may not be worth the effort. So the key idea of our lightweight approach is to use templates that are already compilable source files and hence allow a refactoring engine to recognize templates and update references by that.

Therefore we call our approach "language preserving" as from a point of view of the target language, the template is already syntactically valid, even though a semantically useless file. To our knowledge, there was so far no other experiment that facilitates this idea and actually built a working code generator on that idea. However, *model templates* which are written in the target modeling language and decorated with stereotypes are a similar idea [4].

The rest of this paper is organized as follows. Section 2 describes the template engine and its technical properties. In Section 3 a longer example demonstrates the applicability of the approach. Some refactorings are listed that can be applied to hand-written source code and change the template file automatically. Section 4 gives an overview of the whole process of code generation which supports the usage of the proposed template engine. Section 5 concludes this paper.

2 A "Language Preserving" Template Engine

The main idea of this template engine is not to generate code through completing the template by inserting data in the template holes, but to replace marked exemplary data with real data. The template consists of two types of elements. The main part is code of the target language which is basically copied to the output. Embedded are then special comments of the target language, so-called tags, that are interpreted as commands by the template engine to guide the code generation process. The template engine accesses the model class[1] to retrieve data and writes it to the output. In total, comments plus the exemplary data, which is usually a single word, are transformed into the real piece of code. An example of this behavior can be found in Figure 2.

A tag in this template language looks like a special Java multiline comment (`/*C ... */`). As described above, the word following the tag is replaced by real data. Which data is chosen depends on the tokens which are the words within the special comment. Our engine allows two possible types of tokens which are all concatenated directly without spaces to replace the word after the comment. The first type of token is surrounded by `%`-characters and serves as substitutable parameters. They are substituted by the value of the model attribute with the same name as shown in Figure 2. The second type of token is regarded as plain text and is copied directly to the output. This feature can be used for example to form valid strings.

In our template engine at each generation step one model object (that is an instance of a model class) has the "focus". This concept stems from traditional object-orientation, where exactly the *this*-object is active. This means that all

[1] Please note that a "model class" belongs to the abstract syntax describing the model and thus to the meta-model.

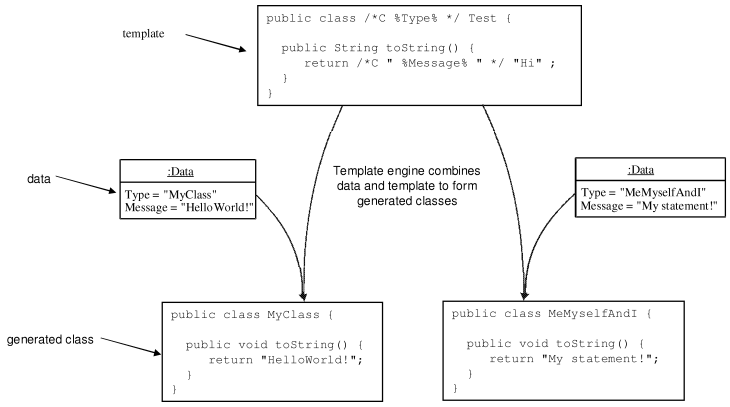

Fig. 2. Example for code generation with proposed template engine

tokens surrounded by `%` access attributes of the model object. In this point it differs from Velocity which extends to the definition of variables to access different model objects at the same time. In order to change the active model object, our template engine supports the following additional tags introducing the usual control structures:

`/*for %X% */ ... /*end*/` The prerequisite for this tag is that the active model object resp. its class provides a method `getX()` which will be accessed by the template engine. The return value of this method becomes the active model object until the end comment (`/*end*/`) is reached. Then the original model object becomes the active model object again.

`/*foreach %X% */ ... /*end*/` The prerequisite for this tag is that the active model object has got a method `getX()` which will be accessed by the template engine. The return value must be of the type `java.util.List`. The first entry in this list becomes the active model object and is used for code generation until the end comment (`/*end*/`) is reached. The resulting behavior is repeated for every entry in the list.

`/*if %X% */ ... /*else*/ ... /*end*/` The prerequisite for this tag is that the active model object has a method `isX()` which will be accessed by the template engine and returns a boolean value. If the return value is true, the template engine uses the code written in the first clause for code generation, otherwise the else clause is used. However, this control tag does not change the focus of the used model object.

The above mentioned control structures can be nested arbitrarily to access the model tree (See Section 4 for an explanation why we make use of trees and not graphs here). Within a `for` or `foreach` environment the nodes upwards in the tree can be accessed via `%number%name%` where `number` is a natural number counting the steps upwards in the model tree (`1` for direct parent) and `name` is the name of the attribute to be accessed.

In certain situations it is easier to directly output data without having example data after the comment that will be replaced. This resembles the usual behavior of a template engine and is supported via the tag `/*O ... */`.

With these mechanisms, a reasonable code generator engine is given to sufficiently demonstrate and explore our concept.

3 Example

To explore the properties of our approach, we use a comparative template engine for generating code. Martin Fowler shows in [8] different ways to realize code generation. In the following we concentrate on his solution using the template engine Velocity. The example is taken from the article, but translated from C# to Java and slightly adopted to make the concepts clearer and shorter without losing the crucial points. Especially note that while both the template and the resulting Java code are full class files, in the figures only the class body is shown.

The main idea of the example is to customize a so called reader by objects of the type `ReaderStrategy`. These are used to parse files in a line-oriented file format, where keywords at the beginning of each line determine the structure of the following data. Depending on the keyword, characters between a start and an end position have a meaning and should be extracted. For further details on the example see [8]. The example in Figure 3 shows that it is possible to invent APIs to split the code into two parts, one containing basic functions and one containing code using these basic functions in a way that is specific for a given problem.

The code from Figure 3 can be used as it is, but for complex files, one would rather like to use a form of description that just contains the information needed and not that much extra technical code. This form of description is usually called

```
public void Configure(Reader target) {
  target.AddStrategy(ConfigureServiceCall());
  target.AddStrategy(ConfigureUsage());
}
private ReaderStrategy ConfigureServiceCall() {
  ReaderStrategy result = new ReaderStrategy("SVCL", ServiceCall.class);
  result.AddFieldExtractor(4, 18, "CustomerName");
  result.AddFieldExtractor(19, 23, "CustomerID");
  return result;
}
private ReaderStrategy ConfigureUsage() {
  ReaderStrategy result = new ReaderStrategy("USGE", Usage.class);
  result.AddFieldExtractor(4, 8, "CustomerID");
  result.AddFieldExtractor(9, 22, "CustomerName");
  return result;
}
```

Fig. 3. Generated code (adapted from [8])

a domain-specific language (DSL). Figure 4 shows the condensed information garbled (or even hidden) in the Java code in Figure 3.

```
mapping SVCL ServiceCall
   4-18: CustomerName
  19-23: CustomerID
mapping USGE Usage
   4- 8: CustomerID
   9-22: CustomerName
```

Fig. 4. Domain specific description for code from Figure 3

The template for the code generation can of course be described using Velocity (cf. Figure 5) or our approach (cf. Figure 6).

```
public void Configure(Reader target) {
  #foreach( $map in ${config.Mappings})
  target.AddStrategy(Configure${map.TargetClassNameOnly}());
  #end
}
#foreach( $map in ${config.Mappings})
  private ReaderStrategy Configure${map.TargetClassNameOnly}() {
  ReaderStrategy result =
    new ReaderStrategy("$map.Code", typeof ($map.TargetClassName));
    #foreach( $f in $map.Fields)
    result.AddFieldExtractor($f.Start, $f.End, "$f.FieldName");
    #end
    return result;
} #end
```

Fig. 5. Template using Velocity (adapted from [8])

To our experience, the templates for our engine tend to be a bit easier to understand than the ones for Velocity because each replaceable tag is usually followed by an example of what could be generated from it. It turned out to be the best strategy, to either use meaningful names (as we did in the example) or to take names that obviously are meant for replacement, e.g. by beginning with double underscores.

Also typos like a forgotten semicolon at the end of a statement are usually discovered quicker, as modern IDEs compile source in the background and mistakes are highlighted immediately. This is possible because the templates for our engine are compilable. Using template engines like Velocity a generation process is required first to detect this kind of errors. As the template file is valid Java source code, various helper functions of modern IDEs can be used to create or modify the template. This includes the generation of get/set-methods, the re-

```
public void Configure(Reader target) {
  /*foreach %Mappings% */
  target.AddStrategy( /*C Configure %ClassName% () */ ConfigureSCall() );
  /*end*/
}
/*foreach %Mappings% */
private ReaderStrategy /*C Configure %ClassName% () */ ConfigureSCall() {
  ReaderStrategy result = new ReaderStrategy
    ( /*C " %name% " */ "SVCL" , /*C %ClassName% */ ServiceCall .class);

  /*foreach %Entries% */
  result.AddFieldExtractor( /*C %LowerBound% */ 4 ,
    /*C %UpperBound% */ 18 , /*C " %Name% " */ "CustomerName" );
  /*end*/
  return result;
} /*end*/
```

Fig. 6. Template using proposed template engine

naming of variables, code assistants like listing all available methods by typing a dot after a class name, and the generation of method bodies for all methods of a superclass. But these features are just by-products of the main advantage, the improved possibility of refactoring the template file together with the generated files.

In our experiment, we applied a number of refactorings, including the following ones using the Eclipse built-in refactoring engine to hand-written source code that interacts with the generated source code from the example. All refactorings lead to an automatic and correct change of our template without any additional effort:

- Renaming
  e.g. `Reader` to `Parser` or `ReaderStrategy` to `ParserStrategy`.
- Change method signature
  e.g. adding an additional parameter to `ReaderStrategy.addFieldExtractor` with a default value or deleting one parameter of the same method.
- Extract Interface
  e.g. extracting an interface from `ReaderStrategy` called `Strategy` and use that instead of `ReaderStrategy` in the generated code.
- Move
  e.g. move the class `ReaderStrategy` to another package. The necessary imports are also updated in the template file.

This list is certainly not complete, but could give an overview on how to apply refactorings that change both the generated and the hand-written source.

In addition to the given experiment, we e.g. developed a code generation for statecharts. A result of further experiments and discussions during the summerschool was that only hand-written code and the template are refactored automatically but the model stays unchanged. Depending on the concrete template

this might be a drawback because the model can contain elements like class, method or variable names which refer to elements in the implementation but are not changed by a refactoring. It is still an open question if there is a lightweight way to overcome this limitation or if only a heavyweight solution exists.

4 Easy Method for Developing a Code Generator

For a complete understanding, we describe a method to develop such a code generator that makes use of our template engine in the following three steps:

1. A prototype for the generated code is programmed manually by developing an example as it can be found in Figure 3. To our experience this first step simplifies the following steps tremendously because programming an example first is usually a lot easier than starting off directly with the template.
2. The variation points (template holes) of the class are identified and special comments are added directly before the words to be replaced. The form of these comments are described in Section 2 and the resulting template looks e.g. like the source shown in Figure 6.
3. In order to generate the resulting Java code shown in Figure 3 from the DSL description given in Figure 4, we need model classes whose instances will represent the information of the DSL description (abstract syntax) internally.

Extracting the information from a textual description is a typical task for a parser, which can be generated by parser-generator or compiler-compiler [5]. A parser generator takes a grammar as input and generates a running parser. A number of tools also generate a default set of classes representing the abstract syntax tree (AST) (e.g. [3, 9, 14]). Available tools differ in the underlying parser technology and the form of these AST classes quite heavily. Some syntax representations for example use untyped trees, others build rather deeply nested trees.

MontiCore is a project at the Institute for Software Systems Engineering at the Technical University of Braunschweig that develops techniques to simplify the definition of domain-specific languages. As MontiCore focuses on analysis algorithms, formal verification techniques and generation mechanisms, it is restricted to textual input. One of its key techniques is to enrich a grammar, such that it contains enough information to generate both parser and AST classes. Technically speaking, the description for creating an AST is identical to the parser rules. This both restricts the choices of a developer in a sense that the AST structure strongly corresponds to the grammar (similar to [9]), but it improves the effectiveness of the developer, as it allows very compact definitions of languages. As a side effect it simplifies the development of new languages for less experienced users.

MontiCore is not a parser-generator on its own, but is built on Antlr, a rather widely used tool [12], as a backend to generate the parser component. The AST class construction and the grammar description form is similar to the one used by SableCC [9]. The underlying parsing technology is a recursive-descent predicate

LL-parser which simplifies the compositional embedding of languages into each other in comparison to using a LR-parser.

```
_______ MontiCore-Grammar _______

File:
  (Mappings:Mapping)+ ;

Mapping:
  !"mapping" Name:IDENT ClassName:IDENT
  (Entries:MapEntry)* ;

MapEntry:
  LowerBound:NUMBER "-" UpperBound:NUMBER
    ":" Name:IDENT ;


_____________ EBNF _____________

File ::=
  (Mapping)+

Mapping ::=
  'mapping' IDENT IDENT (MapEntry)*

MapEntry ::=
  NUMBER '-' NUMBER ':' IDENT
```

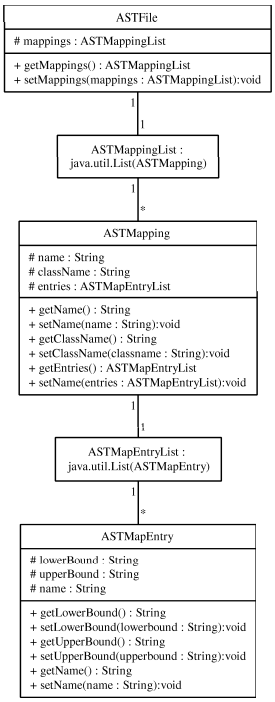

Fig. 7. Overview of MontiCore descriptions

The following example in Figure 7 contains the MontiCore grammar, the respective EBNF grammar that describes the same language and a UML class diagram of the derived AST classes. The generated parser instantiates objects of these classes to build the internal representation of the model (abstract syntax). MontiCore is e.g. used in the example in Section 3 to parse the input data from the DSL description and to generate the AST classes which are used by the template engine. The parser and AST classes of the template engine itself are also constructed using MontiCore.

Based on our experience so far, it is worth extending the current capabilities of defining domain-specific languages and code generators for them, because it indeed speeds up the development and in particular the agile development of software systems.

5 Conclusions and Outlook

In this experiment we have shown how templates for code generation can look like, so that they allow for an automatic refactoring within an agile development

process. We also have demonstrated the usability of this approach by a longer example, where we used MontiCore, a newly developed prototype, to simplify supporting work like the generation of a parser for input data and the creation of classes storing this data.

However, for our approach it was necessary to develop a new template replacement engine and we could not reuse e.g. Velocity. For demonstration purposes, it was sufficient to use the described features. However, for practical use it would be good to have more capabilities of Velocity accessible through the template engine.

Furthermore, the concrete syntax of the template engine is still not very elegant. We used it mainly for experiments first and also integrated it into MontiCore to generate the AST classes itself. But the experiments also showed that for practical usage additional tags simplify the development. This has e.g. led to the development of additional tags like `/*O ... */`. Also the approach where only one model object has the focus at a certain point was weakened by adding the ability to access model objects upwards in the AST-hierarchy by `%number%attribute%`. These experiments encourage us to combine the idea to embed tags as comments with the comfort of a grown-up template engine like Velocity in near future.

References

1. Apache Velocity Website. `http://jakarta.apache.org/velocity/`.
2. Poseidon for UML Website. `http://www.gentleware.com`.
3. M. van den Brand, P.-E. Moreau, and J. Vinju. Generator of efficient strongly typed abstract syntax trees in Java. *IEE Proceedings - Software*, 152(2):70–78, 2005.
4. K. Czarnecki and M. Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *Proceedings of GPCE '05*, pages 422–437, 2005.
5. C. Donnelly and R. Stallman. *Bison: The Yacc-Compatible Parser Generator.* iUniverse Inc., 2000.
6. Eclipse Website. `http://eclipse.org`.
7. M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, 1999.
8. M. Fowler. *Generating Code for DSLs*, 2005. `http://www.martinfowler.com/articles/codeGenDsl.html`.
9. E. Gagnon and L. Hendren. SableCC – an object-oriented compiler framework. In *Proceedings of TOOLS 1998*, August 1998.
10. J. Herrington. *Code Generation in Action.* Manning Publications Co., 2003.
11. W. F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.
12. T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software – Practice and Experience*, 25(7):789–810, 1995.
13. B. Rumpe. *Agile Modellierung mit UML : Codegenerierung, Testfälle, Refactoring.* Xpert.press. Springer-Verlag, 2005.
14. J. Visser. Visitor combination and traversal control. In *Proc. of OOPSLA '01*, pages 270–282, New York, NY, USA, 2001. ACM Press.