CommentTemplate: A Lightweight Code Generator for Java built with Eclipse Modeling Technology

Jendrik Johannes, Mirko Seifert, Christian Wende, Florian Heidenreich, and Uwe Aßmann

DevBoost GmbH
D-10179, Berlin, Germany

Technische Universität Dresden
Chair of Software Technology
D-01062, Dresden, Germany

{firstname.lastname}@devboost.de

Abstract. In this paper we present CommentTemplate, which realizes code generation features, as known from many model-driven development tools, as a Java language extension. This paper first introduces CommentTemplate and discusses its features and limitations. Second, it discusses CommentTemplate as an example of (a) a tool that makes concepts from model-driven development easily accessible for Java programmers and (b) a powerful and stable tool that was realized in a short time thanks to Eclipse modeling technology.

1 Introduction

One of the fundamental technologies of model-driven development approaches is code generation. Hence, many code generation technologies emerged over the years. In the Eclipse modeling project alone, multiple solutions exist [1–6]. Furthermore, there exist numerous code generation tools and framework for Java outside Eclipse, which can also be combined with Eclipse modeling technology (e.g., [7–9] and many more).

An issue model-driven development approaches, and code generation in particular, face when originating from academia, is their adoption by the "common" developer in industry. First, an approach needs adequate tool support above the level of academic prototyping. Second, such tools should be easily accessible for the developer. That is, the developer should be able to immediately start working with the tools without the need to acquire more theoretical knowledge in the beginning (flat learning curve).

Eclipse, and the Eclipse Modeling Framework (EMF), have greatly supported these two points in the past. First, the Eclipse plugin mechanism and the extendability of EMF allow a rapid development of new high-quality modeling tools by reusing existing, well-tested functionality. Second, Eclipse, being the

development platform of choice for many Java developers, gives Java developers a quicker access to modeling technologies, because they are integrated into the tool (Eclipse) they use on a daily basis. However, this second point so far mainly addressed integration with the Eclipse IDE and not with the Java language itself. This makes it still difficult for Java developers to easily understand and use modeling tools which, at a first glance, are not connected to Java programming.

We previously developed the Java Model Printer and Parser (JaMoPP) [10], to bring Eclipse modeling and the Java language closer together. JaMoPP consists of a Java metamodel (defined in EMF's Ecore) and the tooling to parse Java source (and byte) code into instances of that metamodel as well as to print instances back to Java source code. This increases integration of modeling and Java both on the modeling and metamodeling level. On the modeling level, JaMoPP is used to handle the Java language equally to other modeling languages, which allows Eclipse modeling tools to work on Java source code as on other models (e.g., model-to-model transformations can be used to generate Java code). On the metamodeling level, JaMoPP allows the integration of Java and other (modeling) languages. New elements can be inserted into Java by extending the Java metamodel and syntax or parts of Java can be reused in other (modeling) languages by importing the Java metamodel and syntax.

In this paper, we present CommentTemplate, which is a code generation extension for the Java language. CommentTemplate is both an example of a tool that transfers concepts of model-driven development to the programming tool world and an example of a powerful, stable tool that was developed in a short time by taking advantage of Eclipse and Eclipse modeling technology. CommentTemplate takes the important features of existing code generation languages, which are not offered by Java itself already, and implements those as a lightweight extension for Java instead of providing a new language. This demonstrates how fundamental features of model-driven technology can be realised closely to an existing and well-accepted programming language to make these features, which are valuable in their own right, more accessible for programmers. CommentTemplate is developed based on JaMoPP and consists of approximately 800 lines of code. This shows how JaMoPP can indeed be used on the metamodeling level for developing tools that integrate into the Java language and thus benefit from the existing Java tooling in the Eclipse IDE. Although, we are both modeling enthusiasts and Java programmers, we prefer using CommentTemplate over other code generation tools in certain situations. We motivate our reasons for that by discussing the advantages and disadvantages of CommentTemplate.

The paper is structured as follows: Section 2 introduces CommentTemplate an explains its features. Section 3 discusses how CommentTemplate is implemented using Eclipse modeling technology. Section 4 takes a critical look at the usefulness of CommentTemplate and discuss its application areas. Section 5 discusses limitations of CommentTemplate and Sect. 6 points at related work, before a conclusion is given in Sect. 7.

```
1  @CommentTemplate
2  public String helloWorld() {
3      String greeting = "Hello";
4      /*<html>
5          <head><title>greeting world!</title></head>
6          <body>*/
7          for (int i = 1; i <= 5; i++) {
8              String greeted = "World" + i;
9              /*
10                 greeting greeted!<br/>*/
11             if (greeted.equals("World2")) {
12                 /*
13                     greeted, you are the best!<br/>*/
14             }
15         }
16         /*
17         </body>
18     </html>*/
19     return null;
20 }
```

Listing 1. HelloWorld @CommentTemplate method

2 CommentTemplate Syntax and Features

CommentTemplate is a Java language extension for code generation. It makes use of and extends the following Java language elements: *multi-line comments*, *methods*, *local variables* and *annotations*. A code generation template written in CommentTemplate is shown in Listing 1.

A template is defined as a Java method annotated with `@CommentTemplate` that has the return type String.[1] Inside the Method, one can use multi-line comments (`/* */` notation) to define fragments of the template. Around these fragments, arbitrary Java code can be written and used to formulate, for example, loops (Line 7) or conditions (Line 11). Furthermore, one can refer to local variables of type String that are declared before the corresponding template fragment. In the example, the variable `greeting` (declared in Line 3) is referred to two times inside template fragments (Lines 5 and 10).

A `@CommentTemplate` method can be called as any Java method inside arbitrary Java code. However, it will return the expanded template as String. That is, all template fragments are appended and the variables inside the fragments are filled with their values. In the example of Listing 1, the String shown in Listing 2 is produced.

This sums up the basic features of CommentTemplate. However, CommentTemplate offers two additional annotations which help with syntax conflicts between templates and output syntax.

First, one can observe, that no special quotation is used to mark variables in a template fragment in the example (e.g. `greeting` in Line 5). Other code generation tools usually define a fixed symbol for escaping such variables. This is sometimes problematic, because depending on which output is generated, escape symbols can conflict with the output syntax. CommentTemplate does

[1] To write compilable code, a `return null;` statement is needed at the end of the method, which will be replaced by the CommentTemplate compiler (cf. Sect. 3).

```
1  <html>
2      <head><title>Hello world!</title></head>
3      <body>
4          Hello World1!<br/>
5          Hello World2!<br/>
6          World2, you are the best!<br/>
7          Hello World3!<br/>
8          Hello World4!<br/>
9          Hello World5!<br/>
10     </body>
11 </html>
```

Listing 2. Expanded Hello World template of Listing 1

not define such a symbol itself. However, it allows the user to do so by offering the `@VariableAntiQuotation` annotation which takes a String formatting pattern as argument. In the example, we could add an annotation like `@VariableAntiQuotation("#%s#")` to define that variables should be enclosed in # characters. In the example in Line 5, we would then need to write `#greeting#` to a access the *greeting* variable. In such a case, CommentTemplate could be extended to check if a variable is declared and report a missing declaration to the user (cf. Sect. 5.3). If `@VariableAntiQuotation` is not used, this kind of feedback can not be provided.

Second, CommentTemplate still relies on one problematic fixed symbol, which is `*/` to end a template fragment. To generate this symbol in the output (e.g., when generating Java code with comments), an additional feature is needed. Again, usually, a fixed escape symbol, to escape such symbols which are part of the template language itself, is offered. CommentTemplate makes this configurable by offering the `@ReplacementRule` annotation. This annotation takes two arguments, a *pattern* and a *replacement*, which allows the specification of a replacement for a certain String. For the problem described above, one can use `@ReplacementRule(pattern="#/", replacement="*/")`, which replaces all occurences of `#/` with `*/`. `#/` can then be used as an alternative for `*/`.

Both `@VariableAntiQuotation` and `@ReplacementRule` can be applied on the level of single `@CommentTemplate` methods but also on the level of classes. This allows a fine grained control of which escape characters are used where and helps to avoid syntax conflicts with the output syntax.

3 CommentTemplate Compiler

CommentTemplate is implemented as a Java-source-to-Java-source compiler using JaMoPP. Since JaMoPP handles Java source code as models based on an Ecore metamodel of Java, one could also look at the CommentTemplate compiler as a model-to-model transformation with Java as input and output metamodel. The implementation is realized in Java but could also be realized in a model transformation language such as QVT.

The following transformation is performed by the compiler for each method annotated with `@CommentTemplate`: An instantiation of a StringBuilder is added to the beginning of the method body and the method's return statements are

```
1  public String helloWorld() {
2      StringBuilder __content = new StringBuilder();
3      String greeting = "Hello";
4      __content.append("<html>\n");
5      __content.append("\t<head><title>");
6      __content.append(greeting.replace("\n", "\n\t"));
7      __content.append(" world!</title></head>\n");
8      __content.append("\t<body>");
9      for (int i = 1; i <= 5; i++) {
10         String greeted = "World" + i;
11         __content.append("\n");
12         __content.append("\t\t");
13         __content.append(greeting.replace("\n","\n\t\t"));
14         __content.append(" ");
15         __content.append(greeted);
16         __content.append("!<br/>");
17         if (greeted.equals("World2")) {
18             __content.append("\n");
19             __content.append("\t\t");
20             __content.append(greeted.replace("\n","\n\t\t"));
21             __content.append(", you are the best!<br/>");
22         }
23     }
24     __content.append("\n");
25     __content.append("\t</body>\n");
26     __content.append("</html>");
27     return __content.toString();
28 }
```

Listing 3. The Compiled HelloWorld @CommentTemplate method of Listing 1

modified to return the content of that StringBuilder as String. Furthermore, all elements inside the method body are checked for multi-line comments.[2] This accessibility of comments is an important feature of JaMoPP exploited in the CommentTemplate realization, which is not offered by mechanisms such as plain Java reflection. If a comment is found, it is split into lines and for each line, a *StringReference* (Metaclass in the Java metamodel) is instantiated, which is filled with the text from the comment line. Each String that is generated this way, is passed to the StringBuilder by adding an *append()* call to the position in the method, where the comment was located before. For the example, the result of this transformation is shown in Listing 3.

An additional feature of the compiler is the indentation handling. It is a well known problem in code template development, that formatting of the template and formatting of the output are mixed. This basically concerns indentations, tab (`\t`) characters, at the beginning of a line. In the example of Listing 1, one can observe this issue. In Lines 3–19, one tab character is used to indent the method body. This is formatting of the template, but not of the output. The generated `<html>` (Line 4), for instance, should not be indented in the output (cp. Listing 2; Line 1). Additional indentations of blocks inside the method (e.g., for-loop in Lines 8–14) should also be ignored in the output. Therefore, the CommentTemplate compiler keeps track of indentation and corrects the indentation in each String line. This behavior was adopted from Xtend2 [5].

[2] In the Java metamodel, every metaclass is a subclass of *Commentable* which contains the comments that are located before the element in the source file as String.

4 Application and Discussion

One can look at CommentTemplate from two viewpoints. First, one can regard it as yet another code generation tool. Second, one can look at it as a Java language extension that is so slim, that it does not even extend the syntax of the language. With a critical look from these two viewpoints, two questions arise:

1. Why do we need another code generation tool if there are so many already?
2. Is a language extension that does not extend the Java syntax powerful enough to add any significant new functionality that is not offered by Java itself or can be added through a library alone?

4.1 CommentTemplate vs. Code-Generation Tools

We developed CommentTemplate out of a need we had ourselves, which was not met by existing code generation solutions. For us, the most important properties for a code generation tool were: (a) compilation of templates to Java such that we can run them as plain Java applications without requiring a template interpreter at template expansion time, (b) no dependencies in the template code to (possibly changing) runtime frameworks or libraries, (c) tight integration into Java, since this is the language we are working with most, (d) the ability to reasonably format both the output and the template directly (i.e., without using an additional formatting post-processor).

While many existing solutions meet one or more of these criteria, none of them satisfied us completely. Before we decided to develop CommentTemplate, we were going with Xtend2 [5], because it fulfilled criteria (a) and (d) as well as (c) to some degree.

A problem we had with Xtend2 was that it did not satisfy requirement (b) by providing a runtime library which was evolving with each version of Xtend2 over the last year. This made it difficult to develop and use two code generators, which were developed with different Xtend2 versions, side-by-side inside one Eclipse installation, since it is not possible to run two Xtend2 versions in the same Eclipse. This violated our requirement that the more than 700 Eclipse plugins of our open-source toolbox DropsBox [11], which are integrated by a Continuous Integration system and deployed on a single update-side, should all be able to run together in one Eclipse installation.

Regarding criteria (c) one still has to consider that Xtend2 is a separate language and not Java. Although, the barrier for Java developers to get started with Xtend2 is low, since it compiles to Java source code and provides tooling that is oriented at and integrated with the Eclipse Java Development Tools (JDT), developers still need some time to familiarize with the syntax and programming model of Xtend2. If a Java developer only needs a code generation feature, CommentTemplate provides this in the familiar Java syntax and tools. For CommentTemplate, the JDT Java editor can be reused directly with all its established features. No additional editor or other kind of UI tooling (e.g.,

buttons or menus) are needed, since the compiler runs automatically in the background when a Java file is saved. On the downside, CommentTemplate does not offer other additional languages features which are sometimes useful for code generation (cp. limitations of CommentTemplate discussed in Sect. 5).

Instead of developing CommentTemplate, we could have extended or patched an existing open source code generation solution. However, thanks to model-driven technology, in particular EMF and JaMoPP, CommentTemplate was developed in a very short period of time (the initial working version was written on one afternoon). This way, we obtained a solution that exactly met our requirements with little effort.[3]

This shows that Eclipse modeling and metamodeling technology can be used for rapid tool development in a quality that exceeds the level of prototypes.

4.2 CommentTemplate vs. Java libraries

Before we used a code generation tool, we performed code generation with Java directly. In fact, all templates (approx. 230) of our modeling tool EMFText [14] are written in plain Java. The code of these templates bears a likeness to compiled CommentTemplate code (cp. Listing 3). To make it better readable, we wrote a small Java library that helped us with the formatting. However, the main issue remains, which is that there is no way to write a block of output code as a String in Java, since Strings do not support multiple lines. This renders the templates difficult to write and read, because there is a lot of boilerplate between the lines.[4]

Consequently, *multi-line template fragments* is the feature that CommentTemplate provides, which a Java library alone can not provide. Although, we do not extend the Java syntax, we use an existing element of the syntax — multi-line comments — for this feature, by altering the semantics of this element. The advantage of avoiding syntax extensions is that the existing tooling can be used without adjustment. Such an adjustment, in case of the JDT, would be possible but complex, time consuming and error prone; let alone that other Java IDEs would also need adjustment.

Nevertheless, altering the syntax of an existing element can be difficult or even dangerous. For multi-line comments, however, this is not the case. From the compilers point of view, comments have no existing semantics. From a user's point of view, there is still the alternative of single-line comments (`//`) to use inside `@CommentTemplate` methods. Furthermore, outside of `@CommentTemplate` methods, no new semantics are given to multi-line comments.

---

[3] Currently, we use CommentTemplate for example to implement the generator of our Hibernate DSL HEDL [13] (which in turn is used in multiple customer projects) and for our HTML5/JavaScript based company website `http://www.devboost.de`.

[4] The minimal way to write multi-line Strings in Java is to use String concatenation with the plus (+) operator, which still requires opening and closing the String in each line with quotes (").

5 Limitiations and Future Work

As described in the previous section, we designed CommentTemplate as a minimal extension to Java that met our requirements for a code generation tool. Compared to other code generation languages, CommentTemplate misses certain features. Currently, we believe that these features are out of scope for CommentTemplate as we discuss in the following.

5.1 Expressions inside Template Code

First, CommentTemplate does not allow complex expressions inside the template fragments (multi-line comments). Only String variables can be referenced to inject content into the template. Many code generation languages have an escape mechanism, which allows the definition of expressions inside an anti-quotation.

In cases where the `@VariableAntiQuotation` annotation is used in CommentTemplate, we could also support arbitrary Java expressions. For this, we would need to extend the CommentTemplate compiler such that it takes the String from the anti-quoted part and parses it as an expression. Since most of the tooling for this is provided by JaMoPP already, implementing this would probably be possible with reasonable effort.

Currently however, we feel that this feature is not necessary. One can always define a new local variable and derive its value with an arbitrary expression. This forces the developer to keep the template fragments clean from complex calculations. A similar separation of data calculation and template code is also deliberately enforced by StringTemplate [8] with the same argument. However, future experiences with CommentTemplate might lead us to rethink this issue.

5.2 Support for closures/lambda expressions

Most of the mentioned code generation languages provide closure/lambda expression support. Many rely on OCL, or a variant of it, for this. This is convenient in code generation, for example to iterate collections or to map a collection of Objects to a collection of Strings.

We agree that this is a helpful language feature for code generation, but also for other tasks. Thus, we see the responsibility to provide such features at Java itself and they might indeed be provided in Java 8 [15]. Currently, Java classes that use CommentTemplate can be combined with other JVM-based languages which already provide such features such as Scala [16] or Xtend2 [5]. However, this does not allow to use features of these languages and CommentTemplate inside the same class. For this, the CommentTemplate would need to be ported to work directly on the source of these languages.

5.3 Future Development

In the future, we plan to add some additional Eclipse tool support for CommentTemplate. Although, the existing JDT tooling can be reused without extension,

the usability of CommentTemplate can be improved. For instance, when anti-quotation is used, CommentTemplate can report errors about variables that do not exist directly in the source (currently these errors are only seen in the compiled version) and could offer code completion for variables inside templates. Since JaMoPP/EMFText do already offer infrastructure to add this kind functionality, we will constantly improve on it.

Furthermore, we plan to port the templates of EMFText, which are currently written in plain Java (cp. Sect. 4.2), to CommentTemplate. This gives us the possibility to further assess CommentTemplate on a collection of 230 templates.

6 Related Work

CommentTemplate was motivated by the functionalities and ideas behind existing code generation languages and language features such as JET, EGL, Acceleo, Xpand, Xtend2, MOFScript Velocity, StringTemplate or JSP [1–9]. The idea of compiling the templates to Java source code can be found in JET [1], Xtend2 [5] and JSP [9]. The separation of template and output formatting by a smart handling of tab characters was adopted from Xtend2 [5]. The limitation that CommentTemplate does not support expressions inside templates, can be seen as a strength which forces the user to separate model and view as publicized by StringTemplate [8]. Unique to CommentTemplate is its closeness to Java and its lightweightness reflected in its low number of new features added to Java and the fact that compiled templates consist of plain Java code without any dependencies despite Java itself.

7 Conclusion

This paper demonstrated CommentTemplate, a light-weight Java language extension for code generation. CommentTemplate is developed with Eclipse modeling technology on the basis of EMF and JaMoPP. It is an example of both a powerful, stable tool that is developed in short time thanks to Eclipse modeling technology and a tool that brings important concepts of model-driven development closer to the programming world.

Installation Instructions and Screencast

CommentTemplate can be installed from the DropsBox update-site available at `http://www.dropsbox.org/update_trunk` (category CommentTemplate).
A screencast of the CommentTemplate installation and usage is available at `http://www.dropsbox.org/CommentTemplate`

Acknowledgments

This work is supported by:

References

1. Eclipse Foundation: JET Project. www.eclipse.org/emft/projects/jet (April 2012)
2. Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.A.C.: The Epsilon Generation Language. In: Proc. of ECMDA-FA'08. Volume 5095 of LNCS., Springer (2008)
3. Eclipse Foundation: Acceleo Project. www.eclipse.org/acceleo (April 2012)
4. Eclipse Foundation: Xpand Project. www.eclipse.org/modeling/m2t/?project=xpand (April 2012)
5. Eclipse Foundation: Xtend Project. www.eclipse.org/xtend (April 2012)
6. Eclipse Foundation: MOFScript Project (April 2012)
7. Apache Software Foundation: Apache Velocity Project. velocity.apache.org (April 2012)
8. Parr, T.: StringTemplate. www.stringtemplate.org (April 2012)
9. Oracle: JavaServer Pages Technology. www.oracle.com/technetwork/java/javaee/jsp (April 2012)
10. Heidenreich, F., Johannes, J., Seifert, M., Wende, C.: Closing the Gap between Modelling and Java. In: Proc. of SLE'09. LNCS, Springer (March 2010)
11. DevBoost GmbH and Software Technology Group Dresden: The Dresden Open Software Toolbox (DropsBox). www.dropsbox.de (April 2012)
12. DevBoost GmbH: DevBoost Website). www.devboost.de (April 2012)
13. DevBoost GmbH and Software Technology Group Dresden: HEDL - Hibernate Entity Definition Language. www.emftext.org/language/hedl (April 2012)
14. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Derivation and Refinement of Textual Syntax for Models. In: Proc. of ECMDA-FA'09. Volume 5562 of LNCS., Springer (June 2009) 114–129
15. Java Community Process: JSR 337: Java SE 8 Release Contents. www.oracle.com/technetwork/java/javaee/jsp (April 2012)
16. École Polytechnique Fédérale de Lausanne (EPFL): The Scala Programming Language. www.scala-lang.org (April 2012)