Architecture-driven Incremental Code Generation for Increased Developer Efficiency

Malte Brunnlieb
Capgemini Deutschland GmbH
malte.brunnlieb@capgemini.com

Arnd Poetzsch-Heffter
AG Softech
TU Kaiserslautern
poetzsch@cs.uni-kl.de

Abstract: Modern information systems are often developed according to reference architectures that capture not only component structures, but also guidelines for framework usage, programming conventions, crosscutting mechanisms e.g., logging, etc. The main focus of reference architectures is to allow for reusing design knowledge. However, following good maintainable and large reference architectures often creates an undesirable programming overhead.

This paper describes a fine-grained pattern-based generative approach that reduces such implementation overhead and improves the overall developer efficiency by transferring the advantages of reference architectures to code level. Our generative approach extracts specific information from the program, generates implementation and configuration fragments from it, and structurally merges them into the code base. Integrated into the Eclipse IDE, the generation can be incrementally triggered by the developer within the developer's individual workflow. Furthermore, the experiences made within three real world projects will be presented.

1 Motivation

As part of today's software engineering, especially in model driven development, generators have become omnipresent. Until today very different use cases have been managed by different generator applications, e.g., generation of code from different models or Create/Read/Update/Delete (CRUD) user interface generation like done by SpringFuse/Celerio [RR13] or the NetBeans IDE [Com13]. Code generators are often domain specific and focus on small to medium scale projects. Nevertheless, to assure architectural conformance and thus good maintainability also of generated code, the generated code has to be structured in an architectural consistent and easily readable way especially in large scale projects. Hence, our generator approach focuses rather on small architecture-driven implementation patterns. Using these it becomes possible to guide and support the developer during the development of architecture compliant software. This paper explains a text-template-based generator approach—the APPS-Generator[1]—, which is capable of generating very fine-grained increments of code. It has been implemented as an internal developer tool[2] focusing on a smooth integration into the developer's workflow and thus resulting in a guided and architecture-compliant development.

[1] APPS stands for Application Services—a Capgemini-internal business unit
[2] with all rights belonging to Capgemini Deutschland GmbH

2 Related Work of Pattern-based Code Generation

By definition of Arnoldus et al. [AvdBSB12] template-based code generators can be divided into homogeneous and heterogeneous generators. The presented approach can be classified as heterogeneous template-based generator as the template (meta) language is different from the target (object) language. In the domain of homogeneous template-based generators, there are approaches like [BST+94] to enable reusable patterns right within object language. There are also heterogeneous template-based generators, e.g., for CRUD applications like [RR13][Com13]. Also use case independent template-based generators like the Hibernate JPA 2 Metamodel Generator [Inc10] are available to generate any contents from a database input source. As a generation approach focusing on patterns Budinsky et al.[BFVY96] realized a generator for design patterns defined by the Gang of Four (GoF). In contrast to such a generic approach, Heister et al.[HRS+98] indicate, that code generation is more effective for domain specific environments as patterns can be much more specific as in a general context. Our presented APPS-Generator will also implement an approach with basic assumptions on architectural guidelines, such that it can be considered as domain specific, too. The major difference is the ability of incrementally generating code right into the currently developed solution by using structural merge techniques. Furthermore, due to incrementally generating code, the APPS-Generator focuses on fine-grained code templates with the advantage to be able to generate small increments directly fitting into the developers individual workflow.

3 APPS-Generator

For the development of the APPS-Generator, we identified five challenges which have to be managed in order to implement a generic, integrated, and fine-grained incremental generation approach:

1. Structural merge
2. Extension of the template language for enabling Java type checks
3. Context-aware / parametrized generation
4. Generation of semantically and technically closed increments
5. Usable integration into a development environment

With a text-template-based generation approach, we can generate any text-based content from a given object model. But for an integrated and fine-grained incremental generation approach we also have to cope with very fine-grained contents which have to be merged into existing files. So we do not refer to the original use case of text-templates to generate one file per template, but we rather use text-templates to generate patches, which can be applied to existing code. To implement this use case, we have divided the APPS-Generator into mainly three core components, one integrated open source template engine, and the user interface as described in Figure 1. The first input for the APPS-Generator is an input file containing information for processing the target templates. Therefore the input transformer extracts all needed information from the input file and provides these in a simple object model for template processing. After that the model processor injects further context information into the model, which are defined in the context configuration. Currently

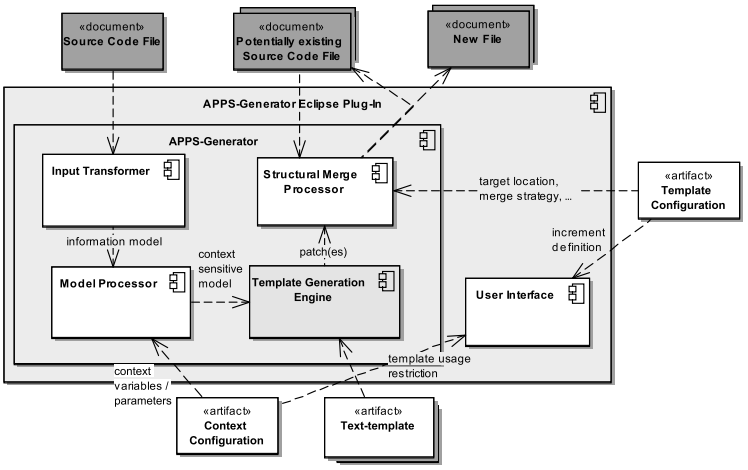

Figure 1: High-level architecture of the APPS-Generator

this can be injected constants or extracted values from the input file location. This enabled us to meet Challenge 3. Furthermore, the model will be extended by a toolkit implementation, which enables simple Java type checks like 'isSubtypeOf' in the text-templates later on such that we can meet Challenge 2. The next step of the generation process is to create a patch, which has to be applied to a potentially existing file afterwards. The patch generation is done by a template generation engine like FreeMarker[pro13] or Velcity[Fou13], which basically generates text from a given object model and a text-template. This generated patch as well as a newly introduced template configuration are the inputs for the structural merge processor addressing Challenge 1. The template configuration mainly specifies the target file location and some more meta-values for every template. If the target file already exists and the target language is supported for a structural merge, the patch will be merged into the existing file. Otherwise a new file will be generated.

On top of the APPS-Generator we have developed a user interface as an eclipse plug-in. With this user interface we are able to meet the last two challenges. On the one hand it consumes the context configuration as this configuration also provides a mapping of inputs to possibly usable templates. This mapping is used to restrict the user to meaningful generation products, which lead to good information hiding as basis for a usable user interface meeting Challenge 5. On the other hand the user interface reads the template configuration as this configuration also defines groups of templates. These groups specify small software increments, which are defined on the basis of the developers needs and which always lead to valid compiler results. This grouping of templates and the ability to integrate small patches into existing files enables us to meet also Challenge 4 of the challenges.

To get a deeper understanding about the different input artifacts and how the Challenges 1-3 have been meet by the APPS-Generator implementation, let's assume our architecture prescribes us to implement a copy constructor for every entity in the core layer. Furthermore, the copy constructor should only perform deep copies for fields, which type are

```
package ${pojo.package};
class ${pojo.name} {
  ${pojo.name}(${pojo.name} o) {
    <#list pojo.fields as field>
      this.${field.name} = o.${field.name};
    </#list>
}}
```

Listing 1: FreeMarker template for a simple copy constructor

subtypes of any specific architectural defined type. Therefore we first need to specify the location of the copy constructor dependent on the target language structure in the template. This can be achieved by specifying the same type within the same package as existent in the input file. As the current implementation of the APPS-Generator uses the FreeMarker engine to generate patches, the example of such a template—shown in Listing 1—is specified using FreeMarker syntax: `${...}` states a read action on the object model and the `<#list ... as ...>...</#list>` construct iterates over the list of Java fields. So we generate a patch, which does not only contain the copy constructor itself, but also the package and class declaration. This is necessary to cause a structural merge conflict of the type declaration later on. For now the copy constructor is simply implemented by a reference copy of each field. In addition, the target file location will be defined in the template configuration as shown in Listing 2. The `destinationPath` is a relative path starting at a configurable root folder. We define it equally to the chosen input file's path and select the `javamerge_override` merge strategy. This strategy tells the APPS-Generator to merge the patch into an existing file if necessary and whenever a unresolvable conflict occurs—e.g., the copy constructor already exists—the patch will override the conflicting contents, whereas conflicts of classes will be resolved recursively. So far we gain a fine-grained template which (re)generates a copy constructor for any Java class as input. In addition to the Java merge algorithm there are also implementations for structural merging XML and the Java property notation.

```
<template id="copy_constructor"
          destinationPath="src/${pojo.package}/${pojo.name}.java"
          templateFile="copy_constructor.ftl"
          mergeStrategy="javamerge_override" />
```

Listing 2: Template configuration for the copy constructor template

For copy constructors it might also be interesting to consider specific types to be deep copied, such that we need simple Java type checks in the template language. As FreeMarker itself does not provide such checks, we have implemented a utility class for 'isSubtypeOf' and 'isAbstract' type checks. The utility implementation holds a Java class loader to load the types given as parameters and returns a boolean value. Given that we are now able to adapt the copy constructor template by a simple case distinction such that all fields of a special type will be deep copied.

So far we have implemented the (re)generation of a possibly architectural guided copy constructor. But let's assume there is another architectural constraint, which forces us to adapt the generated implementation depending on the component the input file is associated

with. Thus, we need context variables, which we assume to be extractable from the path respectively package of the input. To the advantage of our approach more and more reference architectures arise, which often prescribe strict naming conventions. So architectural information such as layer or component names are often encoded into file paths respectively package names and thus match our assumptions. How to retrieve context information from a Java input file is shown exemplarily in Listing 3. The context configuration can contain multiple `trigger` definitions, which specify a mapping between input class types—matching the `typeRegex`—and a set of associated meaningful templates—contained in the `tempalteFolder`. For parametrization purposes `variableAssignments` can be specified, which can be assigned to a value of a regular expression group given by the `typeRegex` or to a string value. The variables will be added to the object model for template generation such that our additional architectural constraint dependent on the input's component can be considered in the templates as well.

```
<trigger id="coreLayerEntity"
         typeRegex=".+\.core\.([^\.]+)\.entity.+"
         templateFolder="entity_templates" >
  <variableAssignment variable="component" regexGroup="1" />
  <variableAssignment variable="var1" value="value" />
</trigger>
```

Listing 3: Trigger example for a context configuration

4 Industrial Experiences

The APPS-Generator has been used in three Capgemini projects in the context of very different use cases. In the first project the APPS-Generator has been used to generate an architecture-conform CRUD application with a JSF user interface. The main challenges were the incremental integration of Spring configuration entries within the architectural predefined files, simultaneously merging xHTML files to integrate new navigation elements, and merging newly generated Java fields into existing Java source files.

The second project uses the APPS-Generator in a more fine-grained generation scenario. The hash method, the equals method, and different copy constructors had to be regenerated dependent on the current input's fields and their types—e.g., for copy constructors as shown in the paper's example. In addition this scenario required the extension of the FreeMarker template engine to cope with Java type checks.

The third project used the APPS-Generator as a generic development tool. The basis was a huge collection of over 400 Java types—in the following called Hibernate entities—generated with Hibernate from the customer's existing database. In order to implement the functional requirements of the new system, the Hibernate entities had to be structured within a type hierarchy of commonly usable interfaces. Dependent on the type name and the field names of each hibernate entity, the subtype relation and all inherited methods could be generated right in place. In this use case the APPS-Generator was not even used driven by a standardized architecture, but it was driven by the developer's needs of automation. Thus the APPS-Generator also provides an interesting framework for defining and using macros and assist the developer in reoccurring tasks.

5 Conclusions

As a generic, integrated and fine-grained incremental generation approach the APPS-Generator could be established in the development process of the developer in very different use cases. It has been indicated, that with the approach of fine-grained incremental code generation the efficiency of the developer himself can be improved. This encompasses the traditional generation of separated code as well as the generation of code within existing code currently under development.

Nevertheless, we have observed that especially the structural XML merge mechanisms require further development and research. Among existing generic structural XML merge implementations, an approach has to be developed which is able to take a XML dialect's semantics into account and thus be able to adequately merge two XML documents of the same XML dialect. Furthermore, some efforts should be allocated for extending the given XML and Java parsers to also take comments into account as these are ignored for efficiency reasons in parsers and therefore unfortunately are lost after a structural merge.

References

[AvdBSB12] Jeroen Arnoldus, Mark G. J. van den Brand, Alexander Serebrenik, and Jacob Brunekreef. *Code Generation with Templates.*, volume 1 of *Atlantis Studies in Computing*. Atlantis Press, 2012.

[BFVY96] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic code generation from design patterns. *IBM Syst. J.*, 35(2):151–171, May 1996.

[BST+94] D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin. The GenVoca model of software-system generators. *Software, IEEE*, 11(5):89–94, 1994.

[Com13] NetBeans Community. Generating a JavaServer Faces 2.x CRUD Application from a Database - NetBeans IDE Tutorial, October 2013. https://netbeans.org/kb/docs/web/jsf20-crud.html.

[Fou13] The Apache Software Foundation. Apache Velocity Site - The Apache Velocity Project, November 2013. http://velocity.apache.org/.

[HRS+98] Frank Heister, Jan Peter Riegel, Martin Schuetze, Stefan Schulz, and Gerhard Zimmermann. Pattern-Based Code Generation for Well-Defined Application Domains. Technical report, In Frank Buschmann, Dirk Riehle (Eds.): Proceedings of the 1997 European Pattern Languages of Programming Conference, Irsee, 1998.

[Inc10] Red Hat Inc. Hibernate JPA 2 Metamodel Generator, March 2010. http://docs.jboss.org/hibernate/jpamodelgen/1.0/reference/en-US/html_single/.

[pro13] FreeMarker project. FreeMarker Java Template Engine - Overview, June 2013. http://freemarker.org/.

[RR13] Nicolas Romanetti and Florent Ramière. SpringFuse - Online Java Code Generator, October 2013. http://www.springfuse.com/.