Improving Implementation of Code Generators: A Regular-Expression Approach

Maria Consuelo Franky  
Departamento de Ingeniería de Sistemas  
Pontificia Universidad Javeriana  
Bogotá, Colombia  
lfranky@javeriana.edu.co

Jaime A. Pavlich-Mariscal  
Departamento de Ingeniería de Sistemas  
Pontificia Universidad Javeriana  
Bogotá, Colombia  
jpavlich@javeriana.edu.co

Departamento de Ingeniería de Sistemas y Computación  
Universidad Católica del Norte  
Antofagasta, Chile  
jpavlich@ucn.cl

Abstract—Code generators are important tools in software development, to automate repetitive coding tasks, facilitate portability, abstract implementation details, and reduce development costs. However, as the complexity of code generators grow, they tend to be harder to maintain, especially when there is a large amount of templates involved. This paper proposes an approach for code generation based on regular expression substitution. Instead of using templates for code generation, this approach transforms existing source code, using regular expression substitution, to implement the required functionality. We are currently applying this technique to strengthen the code generation framework of Heinsohn Business Technology, as part of the "Lion" project co-financed by Colciencias. Our experience shows that, although this approach has a steeper learning curve, it facilitates capitalization of the experience of software development organizations, selecting successfully modules that are taken as a reference source for code generation in future projects.

Keywords—Code Generation; Computer Aided Software Engineering (CASE); Frameworks; Web Applications Development

I. INTRODUCTION

A code generator is “a software tool that accepts as input the requirements or design for a computer program and produces source code that implements the requirements or design” [33]. Code generators are very useful tools to reduce the effort to develop a software system. For instance, in the context of the Model-Driven Architecture (MDA) [37], code generation has a very important role to automatically implement the models created by designers. Similarly, Domain-Specific Modeling (DSM) [34] uses code generation to implement the solution to the domain model. Overall, a code generation tool, combined with appropriate modeling artifacts, can automate repetitive tasks, facilitate portability, abstract implementation details, and reduce development costs, among other benefits [35] [36].

A very common approach to implement code generators is the use of templates. A template describes a way to generate a piece of code from a set of input data, often in the form of models [38]. Template languages can be used to specify the structure of templates and include mechanisms to reference elements from the input data, to perform code selection, and iterative expansion [38].

Templates offer a degree of flexibility in code generation, since one can substitute templates to generate code for different platforms or architectures. However, as the complexity of a code generator grows, more templates are required to be maintained. Moreover, template debugging can be difficult and error prone, since one must first generate code from those templates, execute and debug that code, and then propagate the corrections back to the template. Overall, the more templates a code generator has, the harder is its evolution and maintenance.

This paper proposes and analyzes a technique to generate code based on regular expression substitutions. Figure 1 describes the overall approach. Boxes represent assets (source code, configuration files etc.), rounded boxes represent executables, and arrows represent flow of information. The assumption is that there is a reference source code component, i.e., source code that has been successfully used and tested in previous projects. Developers want to include a similar functionality in a current project. A component parameterization consists of a set of regular expression substitutions that parameterize the source code module for its reutilization in other projects. For instance, one can specify that class names must include an arbitrary prefix, before generating the module for the current project. A project-specific component configuration specifies the concrete values for the parameters denoted by the component parameterization. For instance, the project-specific module configuration can include the specific prefix for the class names for the current project.

The code generator takes as input the reference source code from the previous project, uses the regular expressions specified in the component parameterization to find all of the relevant places in the code, and substitutes those places with
the information of the project-specific module configuration. The result is a new source code module that can be directly incorporated into the current project. The code generator can also modify the source code of the current project to better integrate the desired module.

The authors successfully applied this approach to integrate a security module into web applications. Instead of using templates, the source code of the security module is directly reused into a new project. In this same line of work, the authors are executing a project called "Soporte al desarrollo de aplicaciones empresariales mediante frameworks de generación" ("Enterprise application development support through generation frameworks"). This project, co-financed by Colciencias, is a joint effort between the SIDRe research group of the Pontificia Universidad Javeriana and Heinsohn Business Technology (HBT), a software development company. The goal is to apply the proposed approach to strengthen and extend the code generation framework of HBT.

The remainder of this paper describes the proposed approach and discusses its value. Section II explains the basic concepts required to understand the approach. Section III explains the main characteristics of template-based and regular-expression-based code generation. Section IV describes a case study that illustrates the usage of regular-expressions for code generation. Section V analyzes the advantages and disadvantages of both approaches, based on the results of the case study. Section VI discusses related work. Section VII concludes.

II. BACKGROUND

Automatic software construction is a technology that has been present since several decades ago and it is considered an important tool for abstracting programming language details. Some of the first Computer Aided Software Engineering (CASE) tools date back to the decade of 1980. CASE tools assisted engineers to specify the software design and to generate part of the code of the application. These tools, focused mainly in developing software for mainframes, lost popularity with the advent of Internet, the diversity of user interfaces, and the complexity of multi-layered systems.

The idea of automatic software generation has regained strength during the last years, particularly for enterprise applications. The development of these applications, which include support for distributed processing across the Internet and multi-layered architectures, require the use of frameworks with the following characteristics:

- Ability to capitalize good practices, by using generators that reuse those practices in new projects.
- Fast application development and reduced learning time for developers, so that software products can reach the market as quick as possible.
- Support for standard software patterns and widespread software architectures, such as .NET and Java EE.
- Automatic or semi-automatic generation of non-functional concerns of the application (e.g. security, auditing, etc.), to let developers focus in the construction of the business-related functional concerns of the application.

In this context, a framework is a set of reusable services and components, organized in an extensible structure, to simplify application development.

There are two types of frameworks: infrastructure frameworks, and code generation frameworks. Infrastructure frameworks are software libraries to implement common application concerns, usually associated to non-functional requirements, such as security, persistence, message queuing, etc. These patterns describe best practices to utilize a particular infrastructure framework.

To reduce the learning curve of patterns and infrastructure frameworks, software development organizations construct code generation frameworks. Code generation frameworks are used to generate the skeleton of a project and also to progressively generate additional functionality of new modules and use-case implementation (incorporating infrastructure frameworks). In a software application, the automatic generation of its skeleton contributes to reduce the total time and cost of a project and provides firm basis for the development of the remaining components.

In addition to automating part of the development process, code generation frameworks can assist source code standardization by incorporating the best development practices within the organization. Since developers do not need to learn all of the details of the underlying infrastructure frameworks, learning and training time can be reduced. All of the above can also leverage the use of agile methods to develop software.
Currently, there are well-known standards for infrastructure frameworks [12], but there are no standards for code generation frameworks (except if based on MDE-MDA [34] [36]). However there are several proprietary and open source approaches for code generation [5], [6], [7], [8], [9], and [10]. Software development companies can adapt the above solutions for their specific way to develop applications.

III. CODE GENERATION FRAMEWORKS

A code generation framework comprises multiple individual code generators that create source code with specific functionality. Code generators can be implemented in multiple ways: using template languages, using regular expression substitutions (the proposed approach), and as part of a model-driven approach. The remainder of this section describes these techniques.

A. Template-based Code Generation

Template-based code generation approach relies on the use of templates, which denote the way to transform the input data of the generator into textual files. Some widespread template languages are Velocity [16], Jelly [17], FTL [18], Acceleo [39], JET [40], Xpand [41], and MOFScript [42].

In general, a template language has the following characteristics:

- It includes text that will be included verbatim in the generated code.
- During the processing of a template file there is a context with variables that store the input data to the code generator. Those variables can be referenced in the template file; the template generator substitutes those references with the value of the corresponding variables in the generated code.
- A template file can also have conditional and loop statements that will write specific text in the generated code (either once or iteratively), based on logic conditions.
- A template file can include macros, which are substitution functions that facilitate reuse of template portions.

For instance, the following template uses a language called Velocity [16].

```velocity
#foreach (${name} in ${list})
    import com.$(project).${name};
#end
```

This code contains variable references and a loop to iteratively write a text portion. If the following values are used as input to this template:

```java
project = acme
list = Reports,Security
```

The resulting file will be the following:

```java
import com.acme.Reports;
import com.acme.Security;
```

The generated code corresponds to the beginning of a Java file that imports two classes.

Macros can be used to facilitate reuse of template portions. For instance consider the following macro:

```velocity
#macro (my_macro)
section for ${Ejb} ejb -->
    <module>
        <ejb>${Ejb}_ejb.jar</ejb>
    </module>
<!--
#my_macro()
#end
```

The macro `my_macro` can be used to facilitate the generation of an XML descriptor for a Java EE [13] application:

```xml
<application>
    <display-name>acme</display-name>
    <description>acme Application</description>
    <!--
    #my_macro()
    -->
</application>
```

When the template is executed by the code generator, my_macro is invoked, yielding the following code (assuming that the value of `${Ejb}` is “Security”):

```xml
<application>
    <display-name>acme</display-name>
    <description>acme Application</description>
    <!--
    section for Security ejb -->
    <module>
        <ejb>Security_ejb.jar</ejb>
    </module>
    <!--
    #my_macro()
    -->
</application>
```

The text in bold represents the text generated by the macro.

A software development company can take advantage of the experience gained from previous projects to make a code generation framework. A previous project that uses good design and programming practices can be used as a base. Source code files of that project can be modified to convert them to templates. Those templates parameterize project information, such as component names, use case names, project and company name, etc.
The above process can be repeated for several modules, until the organization has a large set of templates for diverse functions: skeleton generation, CRUD operations generation, use case implementation generation (of various types), component generation, etc. A code generation framework comprises a set of interrelated code generators, such as the above ones.

More specifically, the life cycle of a template-based framework is the following:

1. From a previous successful project, source code files implementing the desired functionality are selected.
2. Those source files are used as a reference to create the templates and macros, parameterizing all of those strings in the source files that need to be generalized: project name, package name, class names, use case names, hard-coded access control roles, etc. If it is required to add any functionality to the generated code that was not present in the reference source code, it must be added directly to the templates.
3. Code generators are created based on those templates, to generate source code with a certain structure. The first code generator should create the skeleton of the new project. The remaining generators should be designed to incrementally add functionality into the generated skeleton.
4. The generated code can be modified by developers to add new functionality. Unless the generator uses a template language that allows incremental generation [39] [40] [41], the generator cannot be used again to generate the same code, since the changes made by developers can be lost.

When developers find better ways to implement portions of a system, they must implement new versions of the corresponding templates. Developers must repeat the entire template creation process (Steps 1-3). The new version of the generator will be available for new projects, but it may be incompatible with previous projects, unless it is adequately designed for incremental generation.

B. Code generation based on regular expression substitutions

A regular expression is a pattern that denotes a set of strings of characters. A regular expression has the following main components [19] [20]:

- Specific characters that a string must contain.
- Character classes that denote characters that a string may contain at a specific position, e.g., numbers, letters, etc.
- Quantifiers that indicate the presence or absence of certain characters inside a string. For instance the `+` symbol after a character class indicates that said character class must be present one or more times in the string. The `^` symbol preceding a character indicates that said character class must not be present in the string.

For example, the following regular expression represents all of the strings that denote a class declaration in Java:

```
public[\s]+class[\s]+[^\s]+\s
```

\`\` denotes space character. \`[\s]+\` denotes that one or more spaces can separate the word ‘public’ from ‘class’. \`[^\s]+\` denotes a sequence of one or more characters that are not spaces, which in this expression is used to represent class names.

Regular expression tools can detect strings that comply with a given regular expression and can also transform those strings. Examples of such tools are the java.util.regex library of Java [21], and the replaceregexp command in ANT [22]. An example of the latter is the following code that transforms a set of property declarations, inserting the string “New” at the beginning of each property name. For instance, the string prop=value would be transformed into Newprop=value:

```
<replaceregexp
  match="([^\s]+)=([^\s]+)"
  replace="New\1=\2"
  byline="true"
  <fileset dir=".">
    <include name="*.properties"/>
  </fileset>
</replaceregexp>
```

The match keyword denotes the regular expression to search for in the input string. The parentheses ‘(’ and ‘)’ surround two groups, which can be later referenced as \1 and \2, respectively. The replace keyword indicates the substitution to perform when a string matches the regular expression. The replace expression prepends New to the first group, then appends an ‘=’ sign, and then appends the second group. The fileset keyword indicates the set of files in which the substitution will be applied.

To illustrate the potential of regular expression substitution, the following example prepends a security annotation to a set of java classes:

```
<replaceregexp
  match="(public class)"
  replace="@Restrict(\#(identity.loggedIn)) \1"
  byline="true"
  <fileset dir=".">
    <include name="*.java"/>
  </fileset>
</replaceregexp>
```

For instance, the following class declaration:

```
public class MyClass
```

will be transformed into:
The above tools can be used to construct code generation frameworks. Figure 2 is a more detailed view of the proposed approach shown in Figure 1. Boxes represent assets (source code, configuration files, etc.), rounded boxes represent executables, arrows represent flow of information, and rectangles with the corner bent indicate additional information about the boxes to which they are connected. The reference source code from a previous project is utilized as input to the generation process. The component parameterization is a set of regular expression substitutions, implemented with regular expression tools, such as the above. Those substitutions parameterize the portions of the source code that must be changed to adequately incorporate the reference source code into any project: the way to rename classes, packages, configuration files, etc. The component parameterization can also specify portions of the source code of the current project that could be modified, for instance, places in the code where annotations should be added, attributes to be added to classes, etc.

The project-specific component configuration is a properties file, a list of keyword-value pairs that indicate the specific values for the elements parameterized by the component parameterization. For instance, assume that a component parameterization includes the following regular expression substitution:

```xml
<replacerexp
  match="\(public\[s+class\[s+TheProject[^s]+" 
  replace="\1 \{project.name\}\2"
...
</replacerexp>
```

The above code will substitute the class name prefix TheProject with the name of the current project, parameterized as project.name.

An example project-specific component configuration is the following properties file:

```properties
project.name=MyNewProject
```

Consequently, the current project will rename all of the prefixes TheProject to MyNewProject.

The life cycle of a regular-expression generation framework is the following:

1. A first generator of a framework is created. Its purpose is to create the skeleton of the project.
2. Each new generator should be based on source code from a successful component of a previous project. For instance, a report-generation component of a previous project may comprise: web pages, navigation flow description files, business objects source code, persistent entities source code, etc.
3. Names of modular units (e.g. classes, web pages, etc.) of the reference source code are replaced by more generic names. These names should clearly denote the meaning of each unit to facilitate reuse.
4. The reference source code with generic names is compiled and tested to ensure that it still provides the required functionality and has no errors.
5. A component parameterization is created, including all of the substitutions to be performed in the reference source code. Any new project should include a project-specific component configuration, with the specific values for the parameters of the component.
6. If it is required that the generated sources contain certain characteristics not included in the reference source code, the component parameterization can include additional regular expression substitutions to be applied directly in the source code of the current project to include the desired functionality. For instance, adding security tags to classes, methods, and web pages.
7. The generated code can be modified by developers to add new functionality. Unless the generator incorporates incremental generation features found in template languages [39] [40] [41], the generator cannot be used again to generate the same code, since the changes made by developers can be lost. When developers find better ways to implement portions of a system, they can modify the reference source code. The reference source code can be directly compiled and tested without requiring a previous generation process. The component parameterization should not require to be changed, unless the reference source code drastically changes its architecture.
C. Code Generation in a Model-Driven Environment

In the context of Model-Driven Engineering (MDE) [25][26], software can be developed starting from models, i.e., abstractions of the system to be developed. A code generation framework generates code that implements those models.

The traditional approach for code generation in MDE is template-based. Models are the input to code generators and can be stored using different formats, such as XMI [43] and EMF [44] among others. Templates reference model elements for code generation [39] [40] [41] [42].

Models can also be implemented using a generator based on regular-expression substitution. Unlike the approach of the previous section, the MDE-based generation framework is still under development by the authors and it is presented here to show the roadmap of the authors' ongoing work.

Figure 3 describes the approach. Notation is the same as Figure 2. Dashed arrow represent an inclusion relation. Reference source code components from previous projects are parameterized similarly to the plain regular-expression approach. Instead of a project-specific component configuration, there is a design model that describes the architecture of the system. The design model may include an abstract representation of reference source code components if necessary. The code generator takes the design model as input, generates the skeleton of the current project, and automatically incorporates all of the required components (the ones referenced in the design model).

To better understand the approach, Figure 4 shows an example design model (a class diagram), simplified for illustrative purposes. Assume that developers have parameterized two reference source codes: a component for persistent entities with CRUD operations and a component for reports. Each class in the diagram has a stereotype denoting the reference code component that will be used to implement that class. Hotel and Room have the «Entity» stereotype, which means that the code that implements these classes will be generated using the persistent entity component with CRUD operations. SalesReport and IncomeReport have the «Report» stereotype; the code that implements them will be generated using the component that implements reports. Note that the parameters to substitute in the reference source code are taken directly from the design model. For instance, all of the modules that implement SalesReport can have “SalesReport” as prefix for their names.

The life cycle of such a code generation framework is very similar to the framework described in the previous section. The first code generator must create the skeleton of the project. The subsequent generators should add functionality to that skeleton. Each generator is evolved by modifying its corresponding reference source code. The main difference is that the MDE-based generation framework must include a generator that takes as input the design model, invokes the generator that creates the skeleton, finds all of the stereotypes that denote reference components (e.g. «Entity», «Report») and invokes the corresponding generators for each component.

If developers require to generate code for a different platform, the MDE-based generation framework can also include generators for other languages or platforms. The process to incorporate these new generators is the same as above, since the model structure remains the same, regardless of the implementation platform/language.

IV. CASE STUDY

The authors are currently working in a Project called “Soporte al desarrollo de aplicaciones empresariales mediante frameworks de generación” (“Enterprise application development support through generation frameworks”). This project, co-financed by Colciencias [47], is a joint effort between the SIDRe research group of the Pontificia Universidad Javeriana and Heinsohn Business Technology (HBT). HBT is a software development company with more than 500 engineers that develop relatively large software projects (100+ engineers) for the public and private sectors of Colombia [31]. The objective of this project is to apply the proposed approach to strengthen and extend the Lion Framework, a code generation framework of HBT for Java EE [13].

To improve the Lion generation framework, it is of practical use to have a well-known framework as a base. The authors decided to utilize the Seam framework, an actively-maintained, open source framework based on Java EE [14]. Seam is a widely used project, and many of the features that have been developed for Seam, eventually become incorporated into the JEE standard [14].
Seam includes a generation framework, called SeamGen, which is based on the FreeMarker Template Language (FTL) [18]. SeamGen can create the skeleton of a Java EE application that contains the following elements:

- Control elements based on JSF [13], Seam [14], and EJB [12].
- Properly configured descriptors.
- Logging code based on Log4j [23].
- Inclusion of several infrastructure frameworks for supporting requirements of security, asynchronous tasks, auditing, business processes, etc.

Additionally, SeamGen also creates the code for CRUD operations over business entities [14] [15], which includes web pages and business components. Since many applications require an important amount of CRUD operations, this generator can significantly reduce development costs.

However, the code generated by SeamGen is not sufficient to construct enterprise applications. There are some elements that need improvement in SeamGen. For instance, code generated by SeamGen is not organized in modules associated with use cases. Additionally, SeamGen does not generate a security module to manage users and permissions. Although there are Seam libraries to incorporate security into web pages and business components, they must be manually incorporated into code.

One of the additions made to the Lion Framework is the automatic incorporation of a security module into the generated code. Instead of using templates for security code generation, the approach is to use regular expression substitutions. This only requires to use an existing security module that has been successfully utilized in previous projects. In particular, the Lion Framework utilizes the CincoSecurity module [28] [29] [30], an open source module that has been successfully used in several projects at HBT. CincoSecurity was originally developed by one of the authors’ company, called CincoSoft, to provide security to their applications.

Figure 5 describes the security code generator of the Lion Framework. There is the source code of the current project that may have been generated using SeamGen. There is a demonstration project that includes the CincoSecurity module and its component parameterization that describes the substitutions to perform both into CincoSecurity and the source code of the current project to incorporate the security annotations. The project-specific configuration indicates basic project information, such as project name, base package, paths, etc.

Figure 5 Security code generation based on regular expression substitutions

To facilitate maintenance of the generated application, the code generator creates a file structure that groups files in folders according to modules and use cases. In terms of security, the code generator automatically performs the following operations:

- Add security restrictions to require an authenticated user or a user of specific roles to invoke each method in a business component.
- Add security restrictions to application descriptors, requiring an authenticated user or a user of specific roles, to constrain access to web pages.

In addition, the code generator performs the following operations:

- Changes all of the string literal values in a web page to locale properties, to facilitate internationalization.
- Configure the application descriptors according to the specific application server and database that is going to be utilized.

Overall the generator described in this section combines generators of SeamGen [14] and several regular-expression based generators and reference source code modules that capitalize the best practices and experiences of Heinsohn Business Technology [31].

In the first phase of the project with the Lion Framework, the software developers of the project (four engineers) tried the template-based and regular–expression-based approaches with different code generation examples. One of such examples is the generation of a CRUD use case for a business object. Another example is a simple business use case with a page that invokes actions of an EJB session bean. The same generator
was implemented with templates and with regular expressions. The developers compared both implementations with respect to several criteria, described in Section V. Later, the same procedure was performed by 14 students of a graduate course of the Pontificia Universidad Javeriana. The results of evaluations of these practices are presented in the following section.

V. COMPARISON BETWEEN CODE GENERATION APPROACHES

Based on the results of the Lion Framework case study, this section describes a qualitative comparison between code generation approaches described in Section III, particularly the template-based approach and the regular-expression-based approach. Since it is still an ongoing work, the MDE-based approach is not included in this comparison. However, the analysis of the regular-expression-based approach should also apply for the MDE-based approach.

The criteria utilized for the comparison are:

- Learning curve. How difficult is it to learn the technologies required to build the generation framework
- Implementation effort. How much work is required to implement the generation framework
- Debugging. How difficult is to debug a generation framework
- Maintainability. How difficult is it to evolve the generation framework over time.

The following sections describe each comparison criteria in detail.

A. Learning Curve

Learning curve for template-based generators is gentler, since template languages require learning a small set of commands, in addition to the language in which the code will be generated.

In contrast, regular expressions have a much steeper learning curve. Developers must familiarize with the theory of regular expressions, and also learn the essentials of regular expression languages. Regular expression languages tend to be harder to read than template languages, thus they are harder to learn [19] [20]. In addition, developers must also learn part of the grammar of the source language in order to write adequate regular expressions.

Overall, it requires less effort to learn template languages than regular-expression languages.

B. Implementation Effort

Creating a template-based code generation framework requires the construction of a set of templates. This set of templates is derived from source code of previous successful projects. Although the task of building templates is not overly complex, it is error prone, since the errors cannot be detected until the code is generated. Moreover, template creation is not automatable and the larger the amount of reference source code, the larger the amount of templates that must be developed. Therefore, the effort of building a template-based generator can grow significantly as more source code is required to be generated.

For a regular-expression-based code generation framework, there is no need to create templates from existing source code. Source code with relatively minor modifications can be used to create a generator. Those minor modifications are usually simple text substitutions (not based on regular-expressions) and can be easily performed with tool assistance [45] [46]. However, as mentioned in the previous section, writing regular expressions is a complex task. It requires a significant effort to write the component parameterization, so the source code can be reused in future projects.

Overall, regarding implementation effort it cannot be said that one effort is better than the other, since both have advantages and disadvantages.

C. Debugging

Template-based approaches require to make a template for each file in the reference source code. This process is error-prone, since developers may introduce unwanted changes in the templates that do not correspond to the original source code. Moreover, debugging of those templates requires that developers first generate code using the templates, compile that code, execute it, determine whether the generated code behaves consistently or not with the reference source code, and propagate any correction back to the templates.

If an error occurs in the generation, i.e., if the generator creates unwanted or erroneous code, it is relatively simple to correct the error, since template languages are simpler than regular expression languages. The assumption is that developers follow a discipline to organize templates in a way similar to the original code, to facilitate finding the template that originated the error and the place that needs to be corrected.

In contrast, the regular-expression approach does not require to manually create new files for each reference source file. With minor modifications, the reference source code can be used to create a generator. Therefore, the risk of having an incorrect representation of the reference source code is reduced.

If an error occurs in the generation it is also relatively simple to find the origin of the error, since the generated code corresponds directly with the reference source code. However it requires more expertise to correct an erroneous regular expression, because of the difficulty of the language. Bugs in regular expressions may also yield unwanted changes in the source code, which may be difficult to detect. However, static type checking languages, such as Java, might increase the chances of detecting those errors at compile-time.

Overall, both approaches have different advantages and disadvantages. Therefore it cannot be said that one approach is better than the other in terms of debugging.
D. Maintainability

The evolution of a code generation framework is associated to the creation of a new generator or the modification of an existing one.

For the template-based approach, whenever a new generator is required, developers must repeat the process of creating templates for each reference source file. If a generator needs to be modified, developers must modify the corresponding templates, generate code, compile and execute it, to ensure it realizes the required changes.

For the regular-expression approach whenever a new generator is required, developers must perform minor modifications to the reference source files, and create the required regular expressions substitutions. If an existing generator needs to be modified, developers only need to modify the reference source code, compile and test it. Unless the reference source code requires significant structural changes, regular expression substitutions do not need to be altered, since they refer to relatively immutable portions of the reference source code. For instance, a regular expression that substitutes class names may not need to be changed, since the grammatical structure of class names does not change over time.

Overall, creation of new generators over time requires a similar effort for both approaches. However, changing an existing generator is significantly easier for the regular-expression approach, since it usually requires only to modify the reference source code. Moreover, the essential structure of a code generator is more stable in the regular expression approach, since template-based approaches frequently require changes in templates whenever developers want to evolve a generator. Regular expression approaches only need changes in the reference source code. Therefore, maintainability is better for the regular-expression approach.

Table I summarizes the comparison between the approaches. The symbol ‘+’ indicates that the corresponding approach is better than the other according to the corresponding criteria. The symbol ‘-’ indicates that the approach is worse than the other. The symbol ‘=’ indicates that both approaches satisfy the criteria similarly.

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Template-based</td>
</tr>
<tr>
<td>Learning curve</td>
<td>+</td>
</tr>
<tr>
<td>Implementation effort</td>
<td>=</td>
</tr>
<tr>
<td>Debugging</td>
<td>=</td>
</tr>
<tr>
<td>Maintainability</td>
<td>-</td>
</tr>
</tbody>
</table>

Both approaches are similar in terms of implementation effort and debugging. Learning curve is better for template-based approaches. However, maintainability over time is significantly better for regular expression approaches.

VI. RELATED WORK

Regular expression substitution is a form of program transformation [50], in which source code is manipulated to create new, derived programs. Stratego [54] is a tool is a language to specify program transformation, based on rewriting rules. Rules written in Stratego have a degree of independence from the target programming language. Similarly, RASCAL [50] is a domain-specific language that integrates source code analysis and transformation, providing rewriting rules for source code. Spoon [53] is a set of Java libraries for program analysis and transformation that focuses in transformations for the Java language. Kalleberg and Visser [52] propose a technique to facilitate incorporation of program transformation reusing existing compilers.

In general all of the above approaches can be used to make code generators in a similar way to regular expressions. One important advantage of the regular expression approach over the above approaches is that it does not require programmers to learn the entire grammar of the target programming language, but only those sections required for the substitution. In some cases when there is only name substitution, programmers can be completely oblivious of the grammar of the target language.

VII. CONCLUSIONS AND FUTURE WORK

This paper presented an approach for code generation based on regular expression substitutions and compared it with the traditional template-based approach. The approach of regular expression substitutions, while it has a steeper learning curve than template-based approaches, it offers a better maintainability.

From the experience of the case study, the direct use of reference source code for code generation with regular expression substitutions facilitates capitalization and standardization of good practices and experience of software development organizations.

Ongoing work is to add new regular-expression-based generators to the Lion Framework and validate them using real-world projects. Some examples of such generators are: a generator for Java Messaging Service (JMS) components [24], a generator that uses specific libraries to create reports, among others. Another work is to migrate to Java EE 6 [32] (the current generator is based on Java EE 5).

Future work is to incorporate the MDE-based approach, described in Section III, into the Lion Framework. The plan is to incorporate graphical and textual models in which engineers can specify the application design and automatically coordinate all of the generators to create a fully-fledged application.

REFERENCES
