ISSN:0975-9646

Vivek Tripathi et al, / (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 6 (3) , 2015, 3046-3050

Comparing and Contrasting different Approaches of Code Generator(Enum,Map-Like,If-else,Graph)

Vivek Tripathi[1] Sandeep kumar Gonnade[2]

Mtech Scholar[1] Asst.Professor[2]

Department of Computer Science & Engineering,

Mats University,Aarang,C.G,India.

***Abstract*— Code generators are very useful tools to reduce the effort to develop a software system. They had a very important role to automatically implement the models created by designers. However, as the complexity of code generators grow, they tend to be harder to maintain, especially when there is a large amount of overhead involved. This paper throws a light on the approaches of code generation for regular expression and finite automata and made a comparison between them, so that we can analyse a technique to generate code based on this approaches.**

***Keywords*— Regular Expression, Code Generators, Enum, Map-Like, If-else, Graph, NFA,DFA.**

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" [3].

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

Code generation is the technique of writing and using programs that build application and system code. To understand code generation, you need to understand what goes in and what comes out.

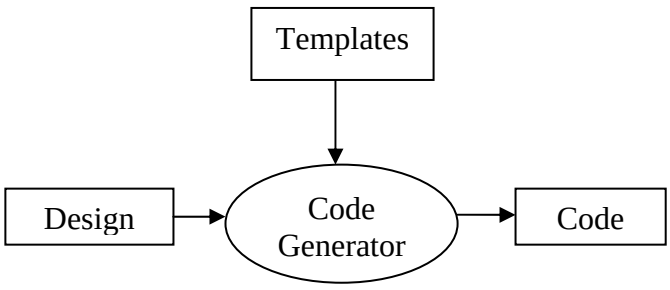

Fig 1. The process of code generation

The code generator reads in the design, then uses a set of templates to build output code that implements the design. The separation between code generation logic in the generator and output formatting in the templates is akin to the separation between business logic and user interfaces in web applications. 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 with the following characteristics [4]:

- **Quality**: We want the output code to be at least as good as what we would have written by hand.
- **Consistency**: The code should use consistent class, method, and argument names. This is also an area where generators excel because, after all, this is a program writing your code.
- **Productivity**: It should faster to generate the code than to write it by hand.
- **Abstraction**: We should be able to specify the design in an abstract form, free of implementation details. That way we can re-target the generator at a later date if we want to move to another technology platform.

Creation of a Code Generator for Regular Expressions takes the following steps to be followed [5]:

- From input regular expression to its respective grammar.
- From regular expression to nondeterministic automaton.
- From NFA to its equivalent deterministic automaton.
- Final generation of java source code using code generators for that DFA.

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 [6]. 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 [6]. It offers 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.

By code generation we mean the compiler's process of converting some of intermediate representation of source code (in this case graphs) into an independent Java Class that whenever is executed represent the same graph automaton. As input for a code generator can be parse trees or abstract syntax trees. We use abstract syntax trees that later on are converted into an intermediate language (sequence of instruction) such as graphs.[7]

Ullman in [1] described a "compiler" for regular expressions are useful to turn the expressions we write into executable code. During this project instead of compiler we will use two other terms: Automatic Programming, and/or Code Generator. Automatic programming identifies a type of computer programming mechanism that generates a computer program (source code) to allow programmers to write high level code. Code Generators or Application generators are software tools that help programmers to generate a complete program or part of it in a very quick way according to the given input specification [8]. Using code generators the programmer can easily edit or modify and execute the output source (program).

The major advantages of using code generators are:

- Saving a lot of development time
- Useful as a learning tool for writing code
- Programs are easy to modify and maintain

A. Regular Expression

A regular expression is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern. They can be used to search, edit, or manipulate text and data.

The java.util.regex package primarily consists of the following three classes.

- ***Pattern Class:*** A Pattern object is a compiled representation of a regular expression. The Pattern class provides no public constructors. To create a pattern, you must first invoke one of its public static compile methods, which will then return a Pattern object. These methods accept a regular expression as the first argument.
- ***Matcher Class:*** A Matcher object is the engine that interprets the pattern and performs match operations against an input string. Like the Pattern class, Matcher defines no public constructors. You obtain a Matcher object by invoking the matcher method on a Pattern object.
- ***PatternSyntaxException:*** A PatternSyntaxException object is an unchecked exception that indicates a syntax error in a regular expression pattern.

To develop regular expressions, ordinary and special characters are used:

TABLE 1
USAGE OF SPECIAL CHARECTERS IN RE

/$	^	.	*	\.
+	?	[	]	[^...]
[...]	\w	re*	(?: re)	\n
^regex	X|Z	re+	re?	(re)
re{ n}	(?> re)	\b	\Q	\E

B. Code Generators

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.

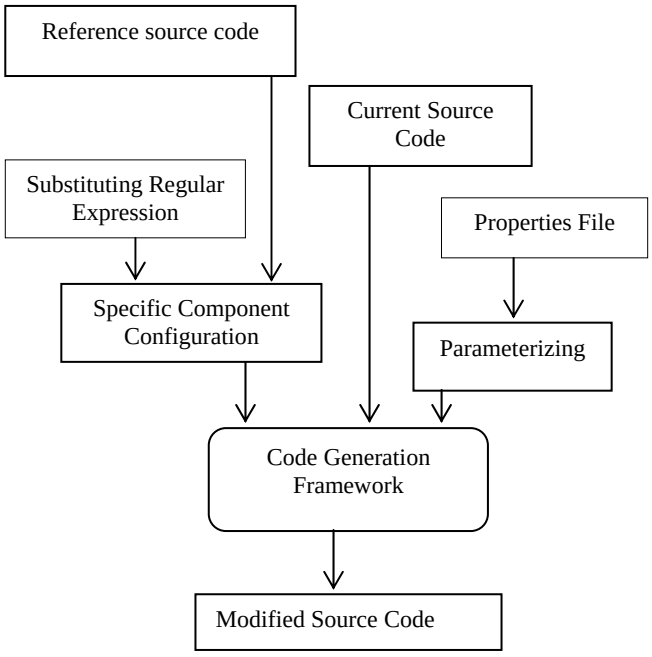

Fig. 2 Overview of the code generator

II. APPROACHES OF CODE GENERATION

Following are the ways to interpret an automaton using Java Code.

A. The Enum Approach

Enums are essentially list of classes, and each member of the enum may have a different implementation. Each enum element may have a different implementation. For example, et us assume that we want to implement the automaton, that represent this regular expression R =^(a+)(b*)(c*)$. We can write the states as elements of the Enum State as follows:

TABLE 2
THE ENUM APPROACH

```
interface State {
   public State next();
}

class Input {
  private String input;
  private int current;
  public Input(String input) {this.input = input;}
  char read() { return input.charAt(current++); }
}

enum States implements State {
  State0 {
    @Override
      public State next (Input input) {
        switch (input.read ()) {
           case 'a': return State1;
           default: return DeadState ;} }
},
  State1 {
    @Override
      public State next (Input input) {
        switch(input.read()) {
           case 'a': return State1;
           case 'b': return State2;
case 'c': return State3;
case '': return null;
default: return DeadState ;} }
},
...
 DeadState {
  @Override
    public State next (Input input) {

return DeadState ;}
};
```

The advantage of using enum approach is that it is clean and Simple approach. All the logic of the automaton is in the same place and it is easy to trace the automaton.Each enum element describes its functionality, by this we mean that each transition is defined.

B. The Map- like Approach

The transitions of a DFA automaton using the map-like approach looks like: hashMap<HashMap<State,Input>, State> that means that for each state on each input we go to another state, where HashMap<State,Input> is the unique key representing the state from where on an input we go to another State represented as the value of the map.

The map is ordered according to the natural ordering of its keys, or by a Comparator typically provided at sorted map creation time. This order is reflected when iterating over the sorted map's collection views.

For example, if we want an object implement the `Map` interface and iterate over every pair contained within it, the following line of code will give the ordering of elements depending on the specific map implementation.

TABLE 3
THE MAP-LIKE APPROACH

```
for(Map.Entry<String,String> entry : map.entrySet())
{
System.out.println(entry.getKey()+"/"+
entry.getValue());
}
```

C. The If-Else Approach

The if-else approach is an easy way to implementing a DFA automaton, such a way works by creating an if statement for each input and the state of the automaton.For example, let us assume that we want to implement the automaton for the shown above; the if-else approach will work as follows.

TABLE 4
THE IF-ELSE APPROACH

```
public class className {
 protected final int state2 = 2; protected final int state1 = 1;
protected final int state0 = 0; protected final int deadState =
-1;
protected int currentState = 0;
public void update(String edge) {
 if(currentState == state2 && edge.equals("b"))
{ currentState =
 state2;
   }
    else if(currentState == state1&& edge.equals("a"))
{
   currentState = state1;
   }
   else if(currentState == state1 && edge.equals("a")) {
   currentState = state1;
   }
    else if(currentState == state2 && edge.equals("b")) {
   currentState = state2;
   }
   else { currentState = deadState; }}
   public boolean matches()
   {
   if ((currentState == state2 || currentState == state1))
   return true;
   else
   return false; }
   }
```

D. The Graph Approach

To represent the automaton the first thing we have to do is build that automaton, so we should create for each state of the DFA a node in the Java class, and for each arc between states and edge should be created. All the nodes and edges are created in the buildGraph () method, where each node and edge is labeled.

Following procedures are to be followed in this approach:

1. A global SetBasedDirectedGraph is created, together with all kind of statedefinitions: state, dead state, edge, current state.

2. To create a node, you must use the method createNode of the SetBasedDirectedGraph,and as a parameter the id of that node should be given.
3. The edges are created using the createEdge method of the SetBasedDirectedGraph class, where the source and the target must be specified, the label is added by using setPropertymethod.

After the end of the input string has been reached, the match's method is called, that simply checks if the description of the current node is Accepted or Initial & Accepted.

III. COMPARING APPROACHES OF CODE GENERATION

A. *Conversion*

Each conversion is defined in Enum approach, therefore it is easy to trace the automaton as the logic is in same place.

In comparison to map-like structure, the conversion is done with the help of a unique key representing the state from where on an input we go to another State represented as the value of the map.

The moves in the if-else approach is done with the help of update method in which transforms the current state of the automaton into another state depending on which input character of the input string is next read. When there is no more characters to be read from the input string the matches method is called, which simply checks if the current state of the automaton is one of the accepted states.

Whereas in Graph approach the transitions are represented by connecting two nodes with an edge.

B. *Execution*

Each state is having enum element, and each of these element is having different execution. In map-like keys represents the inputs and values represent the states.

Whereas in if-else an easy way of implementing automaton is done, by creating an if statement for each input and the state of the automaton. The graph approach takes representation of graph in adjacency matrix and adjacency list.

C. Debugging

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.

In contrast, the Graph approach does not require to manually creating new files for each reference source file. Therefore, the risk of having an incorrect representation of the reference source code is reduced.

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. Sustainability

Overall, creation of new generators over time requires a similar effort for both approaches. However, changing an existing generator is significantly easier for the map-like 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 enum approaches frequently require changes in templates whenever developers want to evolve a generator. If- else approaches only need changes in the reference source code. Therefore, maintainability is better for the enum approach.

Table 5 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 5
COMPARISONS BETWEEN CODE GENERATION APPROACHES

Criterion	Approach			
	Enum	**Map-Like**	**If-Else**	**Graph**
Conversion	+	_	+	_
Execution	+	+	=	=
Debugging	+	_	=	=
Sustainability	+	_	_	+

CONCLUSIONS

This paper provides an idea about the approaches of code generation and made a contrast between them. This comparison shows that Enum approach is having a better functionality among all. As its implementation takes larger memory space because for each state enum element is created.

A better solution is required to simplify and minimize the source code, but so far it has not been proven that they are correct for each case, so that is a work to be done in future. Providing the enum and the map like approach while implementing finite automata using the source code generator will help programmers increase the productivity.

ACKNOWLEDGMENT

This paper has been kept on track and been seen through to completion with the support and encouragement of numerous people including my well-wishers and my friends. At this moment of accomplishment, first of all I pay homage to my guide. This work would not have been possible without his guidance, support and encouragement. Under his guidance I successfully overcame many difficulties and learned a lot. I can't forget his hard times.

REFERENCES

[1] A.V.Aho and J. D. Ullman "Patterns, Automata and Regular Expressions" in Foundations of Computer Science, NewYork:W.H. Freeman & Company, 2010.

[2] P. Linz "*Introduction to Theory of Computation; Finite Automata; Regular Languages and Regular Grammars*" in An Introduction to Formal Languages and Automata, 3rd ed. Massachusetts: Jones & Bartlett Learning, 2010.

[3] IEEE Standard Glossary of Software Engineering Terminology/IEEE
Std 610.12-1990. Inst of Elect & Electronic, 2009.

[4] Suejb Memeti, "Automatic Java Code Generator for Regular Expression and Finite Automata", Degree Project, Course code: 5DV00E, 2012.
K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," IBM Systems Journal, vol. 45, no. 3, pp.621–645, 2006.5DV00E, 2012.

[5] ANTLR (2005), Abstract Syntax Tree, [Online], Available: http://www.antlr2.org/doc/trees.html.

[6] Arora, Sh. Bansal and A. Arora " Application Generators" in Comprehensive Computer and Languages, New York: Firewall Media, 2005, ch. 2 sec. 4.1, pp. 41

[7] R.Sinha at International Journal of Computer Trends & Technology "Transmutation of Regular Expression to Source Code using Code Generators",Vol 3(6), 2012, pp 787-791.

[8] Christin Lungu (2012), Automaton Implementation in Java, [Online], Available:
http://java.dzone.com/articles/automaton-implementation-java