Tabular Code Generation: Write Once, Generate Many

Joel Jones
Department of Computer Science
University of Alabama
jones@cs.ua.edu

1 Intent

Provide a way of encoding information only once and deploying it in two or more environments. Code for the environments is generated programmatically from a data file.

2 Motivation

PressPot is a system that adds annotations to Java `.class` files. [3] These annotations are used by an "annotation-aware" Java Virtual Machine (JVM) which uses these annotations to generate high-quality machine code quickly. If these annotated `.class` files are sent to a normal JVM, they are ignored and the program runs normally. One of the annotations, used for assigning machine registers, is called the virtual register or VR annotation. In the PressPot system, there are two components, the annotator (the front-end) and the annotation-aware Java Virtual Machine (the back-end). The Press Pot back-end is based upon kaffe, an open-source JVM. [6] These two components both need information about the semantics of the Java bytecodes, as well as information about the associated VR annotation. Although the annotator (the front-end) and

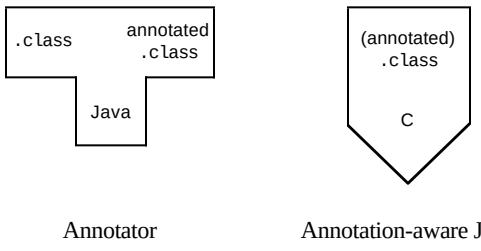


Figure 1: T-Diagram for Annotator and Annotation-Aware JVM

the annotation-aware Java Virtual Machine (the back-end), are not tightly coupled, they

do share a need for detailed information about the detailed definition of Java byte code and its relation to the associated annotation.

However, the front-end is written in Java and the back-end is written in C. The normal solution to the problem of information sharing between two applications is to write a data structure or object (or a sets of such) and use this common code in both applications. In the PressPot system, this would be possible. There is a natural choice in the PressPot system of writing the common code either in Java or in C. If the common code was written in C, then the front-end could use a native calling interface to access the code in a C version of the software. If the common code was written in Java, then the kaffe JVM could use its Java code execution mechanisms to execute the Java version of the common code. However, there are two disadvantages to this approach. First, there is the conceptual distance. The maintainer of the annotation-aware version of kaffe may be thoroughly familiar with C and Java bytecode, but may not be familiar with the Java programming language. To understand the behavior of the program, it is better if the source language is the same. The same argument applies to the maintainer of the annotator. They would rather see the source code in Java, rather than in C. The second disadvantage is performance. Calling C code from a Java program requires the additional overhead that calling Java code does not. Data formats must be converted, garbage collection potentially frozen while the C code is being executing, etc. Calling Java code from a C program involves similar additional overhead.

We solve the problem of keeping information centralized and expressing that information in homogeneous manner by generating code from data. By expressing the information as data, we keep the information in a single place, the data file. By programmatically generating source code for a particular environment, we keep the maintainer visible code homogeneous.

3 Applicability

Use this pattern when the following conditions hold:

- There are two (or more) environments (applications) that share a need for information.
- These environments have dissimilar programming languages or infrastructure.
- The information to be shared has a structure that is static during execution of an application.

Expanding on the last point above, this pattern should not be used as a substitute for input to an application.

The typical (wrong) solution, when there are two environments that share common information, is to have two parallel representations of the information, one per environment. Any changes to the information must be made in both places and is therefore hard to keep consistent. By generating environment specific code from common source, we trade maintaining consistency between two environments for maintaining consistency between the common source file and the generated files.

The applicability of this pattern when two different programming languages are used is easy to see. However, when the same programming language is used, but code is needed for different operating systems or APIs, it's use is less obvious. A common solution for dealing with differing infrastructure is to use Adapter and Factory Method to hide the system differences. [2] Tabular code is appropriate when the infrastructures are profoundly different, such as systems where one of the environments is an embedded system with a strict memory allocation discipline and the other environments have a less constrained memory allocation discipline. Tabular code is appropriate here, as the code may need to be different on almost every line.

4 Structure

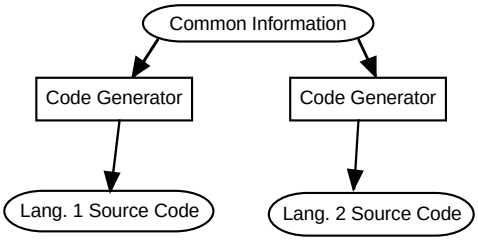


Figure 2: Code Generation from Data

5 Participants

As shown in Figure 2, there are three kinds of participants in this pattern.

Common Information The data file that has the information that is common to multiple environments and which is to be encoded as programming language text specific to those environments. The format of this repository may be as simple as a single line in a text file to a complicated binary, self-referential format stored in an object-database.

In PressPot, this data file takes the format of a multi-line, tab-separated text file.

Code Generator The code that takes as input the Common Information and converts it into programming language source code.

In PressPot, this takes the form of awk[1] scripts that read in tab-separated files. There is one awk script for every type of code structure to be generated.

Source Code The programming language source code that is incorporated into the source code of the application that is specific to each environment.

In PressPot, this takes the form of source code in Java, used by the annotator, and source code in C, used by the annotation-aware version of kaffe.

6 Collaborations

Each Code Generator takes the same Common Information as input, where the Common Information is stored in a single file. When a Code Generator is executed, it produces output for a specific environment/language. This output is typically stored in one or more files. These files are then compiled and linked into separate applications.

A single file for Common Information is the most common case, but multiple files may be used if the information is used for more than code generation and is more naturally stored in separate files.

7 Consequences

There are several consequences of using this pattern. First, the data cannot be tested directly for its correctness in a programmatic fashion. If code was duplicated to achieve the same effect as this pattern, the code could be tested easily. Second, this pattern has a reliance on a scripting language, in addition to the two (or more) languages of the target environment. Such scripting languages can be replaced by a programs written in one of the target languages, thereby eliminating the need for a programmer proficient in some scripting language. Third, there is a potential for a slight performance penalty by using generated code versus hand-crafted code. However, we do get a performance speed-up by generating code, as opposed to reading the data used as logic at run-time and building objects more dynamically. Such a use of run-time interpreted code might be entail use of the Interpreter. [2] Fourth, having logic as data makes using the information for other purposes more convenient. For example, during the implementation of the virtual machine, we used the bytecode data files as input to generate a LaTeX table which was used to track which bytecodes had been implemented in the annotation-aware code generator.

8 Implementation

There are several issues that are important or useful to consider when implementing this pattern. We consider here consistency management, input language design, testing, debugging, and error correction, as well as source readability and data file management.

The use of this pattern trades one consistency maintenance problem for another. The first case is to have parallel implementations of the shared information. There, the two implementations in the two different languages must be kept consistent with each other. The second case is the structure described by this pattern. The first consistency maintenance problem is difficult to automate, as configuration management tools such as *make* can not do a simple check on file modification dates. The second consistency maintenance problem, as described here, is easy to automate, as the file modification date of the source file can be checked against the file modification dates of the generated files.

In implementing this pattern, one of the implementation issues to be considered is the design of the input data file format. The input data file format used in in Press

Pot is just one kind of format that can used. There, we used a file that consisted of lines of tab-separated values. Such a format is easy to write and easy to read, either programmatically or by a human.

A more complicated input language may be used—one which may require a full-blown parser. One example is the interface definition language (IDL) used by the Common Object Request Broker Architecture (CORBA). An IDL file describes an interface (service) that a distributed program may implement, much like what may be contained in a ".h" file in a C program. A "stub" generator takes as input an IDL description and can generate the client side interface in multiple high-level source languages such as C++ and Java. These stubs are then compiled and linked into the different client programs that access the service. With a more complicated language, what can be "said" is broader than in a simple language, but it is more complicated to read. In such a situation, a hand-written parser, such as recursive descent might be used. Another approach is to use a parser-generator, such as yacc. [4]

As mentioned in the Consequences section, testing becomes more difficult using this pattern than otherwise, as errors in the data file can't be tested directly. Rather, the error symptoms are propagated to multiple places in the source code. Additionally, errors in the data file are hard to distinguish from errors in the translation software.

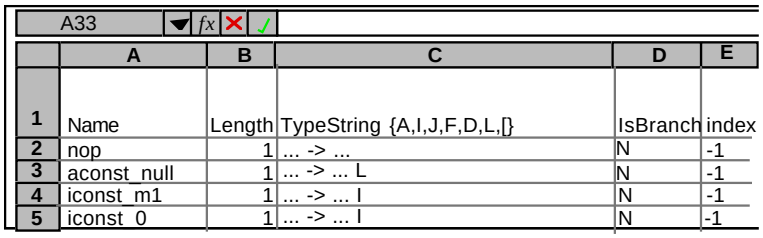
Several ways of ameliorating the testing difficulty can be useful. One is to write consistency checks in the Code Generators. When using a simple tabular data format, one might check that the number of columns on the line matches the number expected, that column values are in range (particularly on numeric columns,) and that the number of lines is within bounds. If a more complicated input format is used, such that a full-blown parser is used for reading the data file input, then syntax errors detected will indicate a problem with the input data file.

Another implementation issue when writing the translators from the data file format to the source code format is to generate source code that is readable. This involves several implementation tasks. First, formatting is important. Generated source code should follow normal coding conventions in terms of indentation, line breaking, spacing, etc. Furthermore, it is also useful to generate two kinds of comments unique to generated code. The first kind of comment is to place at the beginning of the generated code something to the effect of "Don't modify; this code is auto-generated." This will point out to the maintenance programmer that any changes to the generated code will be lost the next time the code generator is run. The second kind of comment is to place information about the generation process itself—most useful is the name of the data file, along with information about time, date, option flags, etc.

The last implementation issue we consider is data file management. The instantiation of this pattern in PressPot utilizes a table-like data format, rather than something more complicated. If such a table-like structure to the data file is evident, then the use of a spreadsheet facilitates manipulation and error-prevention. In this context, the equational facilities of the spreadsheet are not used, only the data manipulation aspect.

9 Sample Code

In figure 3, we see a portion of the spreadsheet used by Press Pot for handling information about Java bytecodes. We show only a portion of the information in this figure. The first row is header information, describing the contents of each column. The second and subsequent rows contain information about the bytecode named in the first column. Column “B” gives the length of the bytecode in bytes. Column “C” gives the way that the bytecode uses the operand stack. Whenever a change is made to the



	A	B	C	D	E
1	Name	Length	TypeString {A,I,J,F,D,L,[}	IsBranch	index
2	nop	1	... -> ...	N	-1
3	aconst_null	1	... -> ... L	N	-1
4	iconst_m1	1	... -> ... I	N	-1
5	iconst_0	1	... -> ... I	N	-1

Figure 3: Spreadsheet from PressPot

spreadsheet, typically when some error is found, a several step process is performed to generate source code. First, the spreadsheet is saved as a tab-separated file, wherein each row in the spreadsheet becomes a line in the file and every column in the spreadsheet is separated by tab characters. Then, several awk scripts are run to generate both Java and C source code. The Java source code is:

```
        public static final int lengths[]= {
            1, 1, 1, 1,   /* 0 1 2 3 */
    ...
```

This source code is generated by the awk script:

```
    #! /bin/nawk -f
    BEGIN {
            IFS = "\t"  # set field separator to tab
            getline # skip header in input
          }
          {
            lengths[NR-2] = $2
          }
    END   {
            printf("    public static final int ",
           "lengths[] = {\n");
            for (i = 0; i < 256; i+=4) {
```

```
            printf("        %d, %d, %d, %d, ",
                  lengths[i], lengths[i+1],
                  lengths[i+2], lengths[i+3])
           printf("/* %d %d %d %d */\n", i, i+1,
                  %i+2, i+3)
          }
          printf("    };\n");
        }
```

The C code generated is similar to the Java code above:

```
int lengths[] = {
    1, 1, 1, 1, /* 0 1 2 3 */
    1, 1, 1, 1, /* 4 5 6 7 */
...
```

The source code for generating C source code is very similar.

```
#! /bin/nawk -f
BEGIN {
        IFS = "\t"  # set field separator to tab
        getline # skip header in input
      }
      {
        lengths[NR-2] = $2
      }
END   {
        printf("int lengths[] = {\n");
        for (i = 0; i < 256; i+=4) {
          printf("    %d, %d, %d, %d, ",
                lengths[i], lengths[i+1],
                lengths[i+2], lengths[i+3])
           printf("/* %d %d %d %d */\n", i, i+1,
                  %i+2, i+3)
        }
        printf("};\n");
      }
```

The differences are the removal of leading spaces and in the declaration of the int array.

The output of running these awk scripts is then incorporated into a Java or C source file. The complete source file is generated by a script that runs various similar scripts for each table, as well as incorporating any static code needed.

10 Known Uses

Using data to guide code generation is a very old approach to certain problems. As mentioned in the Implementation section, the IDL language in CORBA uses a single

file to generate code for multiple languages. Another contemporary example is the use of Java source code as input to the javac compiler, which produces `.class` files, and as input to the javadoc tool, which produces `.html` files. Admittedly, HTML is not a programming language, but it does have a strict syntax that must be followed when generating HTML markup codes.

Another use is in computer-aided software engineering (CASE) tools, such as Rational Rose. [5] CASE tools automate the production of requirement, analysis, and design documentation. Many of these tools allow the production of source code from the design documents. The design documents include class diagrams with specifications of inheritance, methods, aggregation, etc. CASE tools commonly support the production of source code from the design documents, with the potential of producing source code for different languages such as Java and C++.

11 Related Patterns

This pattern is similar to Adaptive Object Models in form, in that logic is represented as data[9]. However, the motivations for applying this form are different. First, in Adaptive Object Models, the logic is represented as code as a way of allowing a (power) user to extend the system, without the aid of a programmer. Second, there is only a single environment in which the logic as data is to be used.

Another related pattern is Interpreter, in that logic expressed in one language is executed in the context of another. [2] However, the form and intent are both different. Interpreter's logic consists of traversals over a tree structure and this traversal occurs at run-time. Also, there is only one language executed, the tree structure.

Program Generator is a special instance of this pattern, where the applicability is driven more by efficiency concerns, rather than consistency between dissimilar environments. [8] In Program Generator, high-level specifications are not directly executable, interpretation is considered to be too inefficient, and producing a compiler to machine-code is too be too much work. Therefore, code is generated.

Table-driven Design is another specialization of this pattern, where the applicability is driven by the tediousness of the changing the specification into source code for a single environment. [7] In Table-driven Design, the specification is translated into a table (array) structure, one per every application. The general interpreter of the table is reused in all applications.

12 Acknowledgments

Thanks to Phyllis Jones for providing the IDL/CORBA example.

References

[1] AHO, A. V., KERNIGHAN, B. W., AND WEINBERGER, P. J. *The AWK Programming Language*. Addison-Wesley, Reading, MA, USA, 1988.

[2] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1994. ISBN 0-201-63361-2.

[3] JONES, J., AND KAMIN, S. Annotating Java class files with virtual registers for performance. *Concurrency: Practice and Experience 12*, 6 (2000), 389–406.

[4] LEVINE, J., MASON, T., AND BROWN, D. *lex & yacc*, 2nd ed. O'Reilly, 1992.

[5] RATIONAL SOFTWARE CORPORATION. Rational rose: Product information. http://www.rational.com/products/rose/prodinfo.jsp, June 2002.

[6] TRANSVIRTUAL TECHNOLOGIES. Kaffe OpenVM^{TM}.

[7] WAKE, W. Pattern patter: Table-driven design. http://users.vnet.net/wwake/patterns/pat9809.shtml, September 1998.

[8] WAKE, W. Pattern patter: Program generator. http://users.vnet.net/wwake/patterns/pat9905.shtml, May 1999.

[9] YODER, J. W., AND JOHNSON, R. The adaptive object model architectural style. *Proceeding of The Working IEEE/IFIP Conference on Software Architecture* (August 2002).