Specification-based code generation using literate programming tools

Carla Marceau
Odyssey Research Associates
33 Thornwood Dr.
Ithaca, NY 14850

January 18, 2000

1 Introduction

For over forty years, programmers have been writing programs that write programs. The usual method of generating code begins with some internal representation of the input (for example, a syntax tree) and employs `fprintf` statements or their equivalent. The "specification" of what is to be generated is in the programmer's mind. There are several disadvantages to this method.

There is no specification of the code to be generated. A declarative specification enables communication between designers and programmers; it provides a vehicle to the programmer for articulating his design; and it can be inspected for correctness.

The code must be generated in the order of the target language. The grammar of the target language is typically unforgiving about the order of the program text. However, the requirements of the target language may correspond poorly with the natural order of the input. For example, awkward programming may be required to maneuver the results of traversing an input syntax tree into the required output grammar.

The generated code may require further editing by hand. Sometimes we generate "skeleton" programs that require further customization by the user. Although code generation is automatic, it can be tricky to regenerate the code without losing changes made by the user.

We encountered these problems while implementing a back end for the SunSoft IDL compiler[1] to generate code to create a secure CORBA ORB for research purposes. As is common in such projects, we divided the work. My coworkers designed the client stubs and server skeletons, while I concentrated on writing a back end for the IDL compiler. The back end had to produce seven files for each IDL input file; the contents of each file had to meet the needs of the generic code produced by my colleagues. It soon became

---

[1] The SunSoft IDL compiler front end is generously made available to the CORBA community by SunSoft, Inc., a subsidiary of Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043.

apparent that "illustrative examples" were an inadequate way for them to communicate their needs to me. There were just too many ways to generalize from a single example to the correct output. In addition, I could see that large sections of the code were going to be redesigned and rewritten several times. In desperation, I began casting about for other ways to specify the code to be output by the back end.

The solution was to factor the code-generating problem into two independent problems: extracting the relevant information about the input to code generation, and describing what the resulting file should look like in general.

This paper describes `specweb`, a readable, language-independent way to describe the desired output of code generation, and a tool, `spectangle`, that automatically generates code from a `specweb` specification and a simple description of the input.

The remainder of this paper is organized as follows. Section 1 introduces literate programming and `noweb`, a popular language-independent literate-programming tool. Section 3 describes an extension to `noweb` that makes it possible to conveniently specify code generation. The implementation of spectangle, a tool for code generation, is presented in Section 4. Section 0 relates our experiences with `spectangle` on several projects and compares it with related work.

2 Literate Programmin g

Literate programming was developed by Donald Knuth [2, 3] as a technique for presenting programs to be read by human beings, as opposed to computers. In literate programming, all the tools of typesetting are available for documenting the problem to be solved, the design, invariants of the program, and so on. Just as importantly, the exposition of the program is freed from the tyranny of the order imposed by the language grammar. In literate programming this freedom enables the programmer to present his code in the proper expository order and lets the tangling process worry about the order the compiler expects. Extracting the program from the web input is called "tangling"; converting the document to TeX for typesetting is called "weaving."

Knuth's literate programming system was called WEB; it was restricted to Pascal programs and to typesetting in TeX, Knuth's typesetting language. More general literate programming systems developed to enable programmers to use other programming languages and typesetting systems. One of the most popular of these is `noweb`, created by Norman Ramsey [4, 5]. In addition to being independent of programming language and typesetting language, `noweb` is fast, well documented, and easily extensible. `specweb` is based on Ramsey's `noweb`.

A `noweb` program consists of *chunks*; documentation and code chunks may appear in any order. Documentation chunks begin with a line that starts with an at sign ("@") followed by a space or new line. Code chunks begin with a *code chunk definition*:

```
<<chunk name>>=
```

on a line by itself. A chunk is terminated by the beginning of another chunk, or by end of file.

Code chunks contain program source code and references to other chunks. A *code chunk reference* is the chunk name in angle brackets, with no equal sign, e.g.

```
<<chunk name>>
```

Several code chunks may have the same name; `notangle` concatenates their definitions to produce a single chunk. Code chunk definitions are like macro definitions; `notangle` extracts a program by expanding one chunk (by default, the chunk named `<<*>>`). The definition of that chunk contains references other code chunks, which are expanded in turn, and so on.

3 Specweb: a language for specifying code generation

We have stated that we factor the problem of code generation into tow parts: extracting the relevant information about the input and specifying what the output should look like.

The relevant information about the input can be viewed as consisting of a set of *entities* of various kinds and of *attributes* of the entities. For example, the entities of most programming languages consist of variables, types, functions, and so on. The attributes of an entity depend on what kind of entity it is.

As a concrete example, consider the type declarations of Figure 1, which shows a simple IDL enum type declaration. It mentions four entities, an enum type and three labels (in IDL, an enumeration literal is called a label).

```
enum flavors (vanilla, chocolate, strawberry);
```

Figure 1. Fragment of an IDL interface declaration

The entities of the declaration are shown in Table 1.

Table 2 shows some of the attributes for the declaration of `flavors`. The attributes capture the information that `flavors` has three labels, that `vanilla` is a label of `flavors`, and that `vanilla` is the first label (labels are zero-origin indexed). Attributes for the other two labels are not shown.

Type	Entity
enum	flavors
label	vanilla
label	chocolate
label	strawberry

Table 1. Entities of the IDL fragment

Entity	Attribute	Value
flavors	numLabels	3
flavors	label	vanilla
flavors	label	chocolate
flavors	label	strawberry
vanilla	type	flavors
vanilla	ordinal	0

Table 2. Some attributes for IDL fragment

Now, suppose that we wish to create the files corresponding to the Java mapping for `flavors`. The Object Management Group (OMG) has published a specification for the language mapping from IDL to Java. In order to implement the mapping, we will create a specweb specification for the mapping of IDL enum types. The specification is a noweb file with a special sub-grammar used for chunk names, so that we can define code chunks for the given entities. Within chunk names, braces and dots have a special meaning. Within chunk names, `{enum}` means “instance of `enum`.” The construct `{{enum}.label}` means “attribute label of an instance of `enum`.”

When a name like `{enum}` appears in the name of a code chunk definition (e.g., `<<declaration of {enum}>>=`), then the definition stands for one code chunk definition for each `enum` entity. For example, consider the simple code chunk

```
<<{label} declarations>>=
<<{label} value declaration>>
<<{label} object declaration>>
```

This stands for a number of code chunks, one for each label. The occurrence of {label} in the chunk definition binds the name throughout the chunk. Thus, given the entities of Table 1, this stands for three code chunks, one of which is:

```
<<vanilla declarations>>=
<<vanilla value declaration>>
<<vanilla object declaration>>
```

In general, a specweb code chunk *reference* stands for one or more instantiated code chunk references. Consider the code chunk reference

```
<<{{enum}.label} declarations>>
```

If `enum` is bound to `flavors`, this stands for:
`<<{flavors.label} declarations>>`,

which in turn stands for

```
<<vanilla declarations>>
<<chocolate declarations>>
<<strawberry declarations>>
```

Eventually information about the entities and their attributes has to get out of chunk names and into the generated code. For this purpose, chunk references of the form

`<<{entityName}>>` and `<<{{entity}.attribute}>>` are treated specially. Consider the code chunk

```
<<{label} intval>>=
_<<{label}>>
```

For the label `vanilla`, this expands to

```
<<vanilla intval>>=
_vanilla
```

Specweb types. So far, we have not used the word “type” to describe `enum` and `label`, in order to avoid any confusion with the IDL types of our example. Now, however, we need to be more explicit. A *specweb type* defines the attributes that an entity is expected to have. `Enum` and `label` are examples of specweb types.

Specweb types have name equality. An entity may have more than one type, so that users can implement their own subtyping or other type systems. For example, `flavors` may be an `enum` and also be an `idltype`, following the type hierarchy of CORBA IDL.

The attributes associated with `enum` and `label` are defined in Figure 2.

```
type enum
attribute fullPackageName String single
attribute name String single
attribute numLabels String single
attribute label label list
type label
attribute name String single
attribute ordinal String single
attribute type enum single
```

Figure 2. Specweb types to describe an IDL enumeration type

The table in Figure 2 states that `enum` is a specweb type with three attributes: `name`, `numLabels`, and `label`. The attributes `name` and `numLabels` are Strings, while the `label` attribute has a value which itself an entity of specweb type `label`. The fourth item in the definition of each attribute is its *number*, which is one of four keywords:

single. The attribute must have just one value.

optional. The attribute may have zero or one values.

list. The attribute may have one or more values.

optlist. The attribute may have zero or more values.

Most of the attributes in our example must be defined just once, but an enum can have one or more labels.

Optional attributes. Optional or optlist attributes are useful for conditionally generating code. For example, a Java array declaration might look like

```
<<{{variable}.type}>> <<brackets if {{variable}.isArray}>>
<<{{variable}.name}>> <<{{variable}.initialization}>>;
```

The type `variable` has an optional attribute `isArray`. If the `isArray` does not have a value for variable `foo`, then `<<brackets if {{variable}.isArray}>>` is omitted for `foo`. Otherwise it is expanded.

This is the only mechanism for conditional code generation in specweb. Specifically, it is not possible to generate code only if two values are equal or if something is *not* true. Optional attributes are similar to if/then statements. To get the effect of if/then/else statements, it is usually preferable to have attributes that take two distinct values—for example, `isArray` could take the values `isArray` and `isNotArray`.

Entity and attribute names. Specweb has a flat namespace, with no notion of scopes. Thus, if two entities are both named "x," specweb assumes that they are the same entity. Therefore, it is necessary to name each entity uniquely in the entity and attribute tables. (We have ignored this requirement in this paper for the sake of simplicity.) Most entities will therefore need an attribute `name` that recaptures its simple name.

In our simple example, we might use IDL's scoped names to uniquely identify entities. Thus, the entity table would look more like the following.

Type	**Entity**
enum	context::flavors
label	context::flavors::vanilla
label	context::flavors::chocolate
label	context::flavors::strawberry

Special characters. Because of their special meaning to spectangle, braces and periods are not allowed in chunk names in a spectangle specification.

List separators. When one chunk reference expands to a list of chunk references, the default list separator is a new line. For example, the chunk reference `<<{{enum}.label} declarations>>` expands to

```
<<context::flavors::vanilla declarations>>
<<context::flavors::chocolate declarations>>
<<context::flavors::strawberry declarations>>
```

A different separator can be specified by enclosing it in brackets and quotation marks immediately following the left brace that produces the list. For example, suppose that we wish to reproduce the original IDL enum type declaration. We can write it as

```
enum <<{{enum}.name}>> (<<{[", "]{{enum}.label}.name}>>);
```

For our example, this expands to

```
enum flavors (vanilla, chocolate, strawberry);
```

4 How spectangle works

We can now state how spectangle produces a noweb document from a specweb specification. `Notangle` is the noweb program that produces code from a noweb specification. Spectangle acts as a front-end to `notangle`. The `noweb` file that it accepts as input contains two kinds of chunks. Some, which we call template chunks, are not intended as input to `notangle`. We have described how spectangle instantiates template chunks. In addition to the template chunks, there must be at least one non-template chunk from which noweb can begin tangling.

Spectangle begins by examining the non-template chunks, that is, chunks whose names do not contain special spectangle syntax. If one of these chunks contains a spectangle chunk reference (e.g., "`<<{label} declarations>>`"), spectangle expands it for all labels. When all chunk references in the chunk have been completely expanded, the chunk is output to `notangle`.

The instantiated chunk references require instantiation of template chunk. For each instantiated chunk reference (e.g., "`<<context::flavors::vanilla declarations>>`"), spectangle generates the corresponding chunk from the appropriate template. The newly generated chunks are non-template chunks, and spectangle processes them and outputs them to `notangle`. Like the original non-template chunks, they may contain further references to chunks that require instantiation.

Work that requires code generation generally proceeds in two stages—programming and production. Spectangle has two phases, as shown in Figure 3 and Figure 4. In the first phase, the specification file and the type table are input and a code generator object is produced. (Spectangle is written in Java. The code generator is a serialized Java object.) In the second phase, corresponding to the production stage of development, the code generator uses the entity and attribute tables to generate code. During this stage, there is no longer any need to reread and reprocess the specification file, since the information in it is encapsulated in the code generator.

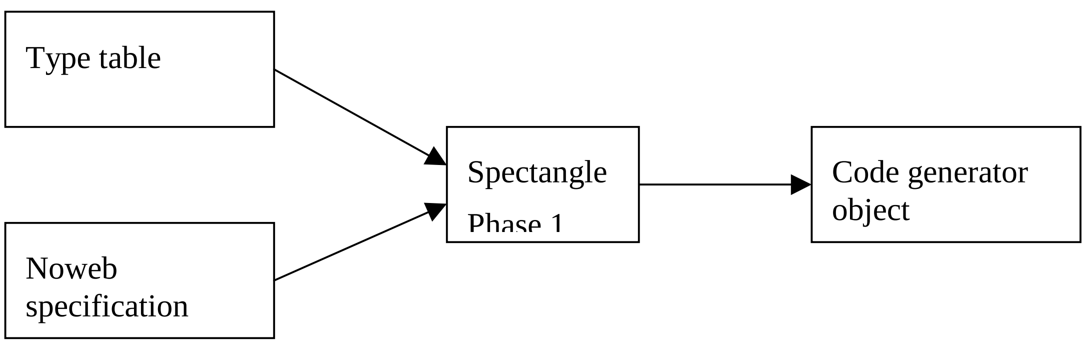

Figure 3. Spectangle Phase 1: creating the code generator object

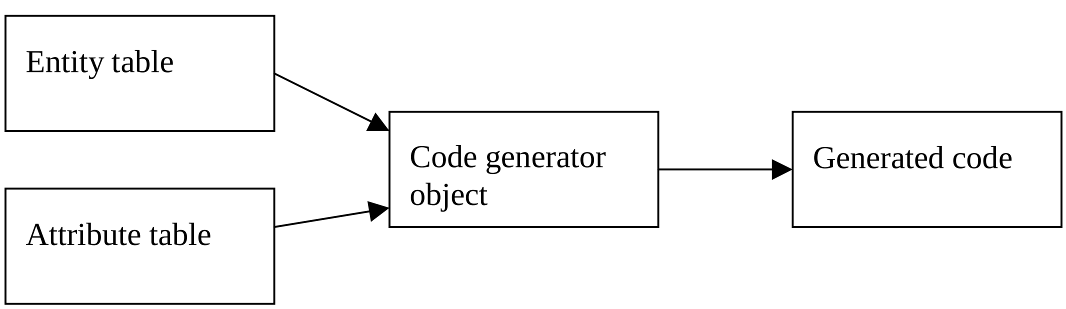

Figure 4. Spectangle Phase 2: generating code

The first phase makes use of the filter feature of noweb. All noweb programs are implemented as pipes. The first member of the pipe is markup, which translates the noweb input into an internal form that is convenient for processing. Filters (such as the first phase of spectangle) can manipulate the program in this form before sending their output to the backend of notangle.

Spectangle has been written and implemented on Windows NT. Most people who write a literate program write the prose documentation part in LaTeX, and noweb supplies a program—noweave—for turning the program into a typeset document for the human reader. Writing with a text editor is important for most literate programming, because it is necessary to read and update the code frequently during programming and debugging. This is much less true in specifying code generation. It is possible to maintain the original specification in Word, for example. When the specification is updated, the Word document can be written out in text format. The text file can be used by spectangle as a specification. The appendix of this paper is a specification for the Java mapping for IDL enum types, written as a Word document.

5 Discussion

People sometimes ask why a code generator is necessary. The answer is that it is not necessary, but it can significantly improve productivity. The appendix to this paper contains a specweb specification for the Java mapping of IDL enum types. In general, writing such a specification is hard. Typically the writer provides a general description of the mapping, which is rather abstract but precise, and supplements it with an example, which is very concrete but difficult to generalize from. A specweb specification is precise and intuitive, combining the advantages of both approaches. Further, it can be used to automatically generate the code. Finally, writing the specifications is easy and enjoyable, because it is simply writing the program, merely leaving some blanks to be filled in for specific instances of the program.

A specweb specification can be read by human beings, discussed in walkthroughs, and critiqued by reviewers. It can be tested easily, using hand-generated entity and attribute tables. This can be done without recompiling the system that produces the code, which frequently speeds development. On the projects on which we used specweb, programmers unanimously felt that they had saved time by comparison with traditional code generation. Furthermore, they felt that they had enjoyed the experience and looked forward to using specweb again.

Factoring the code generation problem into two problems (extracting information about the input and specifying the output) has another advantage. A programmer may write a backend to a compiler in order to generate code. After the project has been successfully completed, he realizes that he can use the same input to generate an automatic debugger. Now he must modify the compiler backend to write more print statements to generate the debugger. Using the specweb approach, the compiler produces entity and attribute tables for its input. That job, once done, is done. To generate an automatic debugger, the programmer just needs to write the specweb specification for the debugger.

In our experience, programmers quickly mastered the concepts of speweb. It took somewhat longer to find the most convenient specweb idioms for expressing code. There is a trade-off between the number of attributes and the complexity of the specification. Ideally, the attributes should simply express the essential semantics of the entities, and the specification should do everything else. There are a few cases, however, where we have pushed additional complexity into the attributes. One of these is identifier generation. Given a variable named `foo_bar`, we may want to generate additional identifiers (`fooBar`, `FooBar`, `FOO_BAR`, `Foo_Bar`). Currently, the only way to do this is by creating additional attributes. A second example is expressing conditionals—programmers would like to have a more expressive way to define conditional code. The COGENT code generation system (see Related Work, below) enables the user to define functions alter the generated code. We prefer to keep the number of mechanisms at a minimum and worry that such functions may open the door to just the kind of operational code generation that we wish to avoid.

Related work

Budinsky et. al. [1] have created a code generation system, COGENT, with a declarative specification to generate code for design patterns. Their specification language includes named, parameterized code fragments called macros; special directives assign values to parameters and direct the placement of code. Conditional and repetitive directives are included, but there are no classes of entities to drive instantiation. Their system interactively supplies the COGENT interpreter with the kind of information contained in our entity and attribute tables. Thus COGENT depends on a *mapper* that mediates between user input (in this case interactively via web pages) and the COGENT interpreter and scripts. By contrast, spectangle is a stand-alone batch tool.

Because the COGENT interpreter is called directly by the mapper, experimenting with new parameters (our entities and attributes) is somewhat more complicated. The mapper has to be (re)programmed to generate the required attributes. By contrast, experimental entity and attribute tables can be constructed by hand and tested before the code is written to generate them.

The IBM system includes some facilities not available in specweb. *User transformations* enable the user to define operations on macros. For example, xxx:toupper expands to xxx with each character capitalized. The user can supply his own transformation functions, which the COGENT interpreter calls. In spectangle, such transformations require additional attributes, as discussed above.

Two problems raised by Budinsky et.al., which we have also encountered, arise when the user must edit generated code "skeletons." In their terms, these are

Invasiveness. The user must understand where he is required to modify the code and where modifications may not be made. Even with a skeleton file full of instructions, it is all too easy to make mistakes.

Irreversibility. Regenerating the code involves a hand merge of the previously modified code with the new and possibly different generated skeleton.

A possible solution to these problems would be for the system to supply the user with a list of code chunk names requiring definition. References to these code chunks appear in other files that the user does not edit, thus obviating the invasiveness problem. The user must supply the definitions for code chunk names of the list, e.g.

```
<<body of function foo>>
```

The user's code chunks would be input to spectangle in later "builds" as part of the literate program.

When the list of code chunk names is regenerated, the user compares the old list and the new. He can remove code chunk definitions that are no longer needed (though they are harmless if not deleted) and add definitions for newly generated code chunk names. What makes this easier than re-editing generated files is that the system makes sure that the code chunks are placed in the correct location in the files. This considerably reduces the problem of irreversibility.

The disadvantage of this approach is that the user must be aware of and actually engage in literate programming in order to modify generated code. This may prove a major obstacle to some programmers, although others may welcome the change to write and document their code in a literate manner.

The tables used by spectangle (type table, entity table, and attribute table) are in a very simple format so that they can be easily written out by, for example, a compiler back end. An alternative approach would be to use the XML format, together with standard programs for reading and writing text files in that format. The advantage of that approach would be that type tables and the entity/attribute table[2] could be generated using standard software, even from data collected for an entirely different purpose.

6 Acknowledgements

The work on spectangle has benefited greatly from discussion with my colleagues at Odyssey Research Associates (ORA), especially Matt Stillerman and Cheryl Barbasch. Norman Ramsey created noweb and made it easily extensible; without noweb, this work would never have been done. He also assisted my early attempts to write literate programs and provided valuable criticism of an early version of specweb.

7 References

1. Budinsky, F.J., *et al.*, "Automatic code generation from design patterns," in *IBM Systems Journal* **35**:2 (1996).

2. Knuth, D.E., *Literate Programming*. CSLI Lecture Notes. Vol. 27, 1992, Stanford, CA: Center for the Study of Language and Information. 368.

3. Knuth, D.E., "Literate programming," in *The Computer Journal* **27**:2 (1984), pp. 97-111.

---

[2] The entity and attribute tables would be combined into a single XML file.

4. Ramsey, N., "Literate programming simplified," in *IEEE Software* **11**:5 (1994), pp. 97-105.

5. Ramsey, N., "Weaving a language-independent web," in *Communications of the ACM* 32:9 (1989), pp. 1051-1055.

8 Appendix: IDL to Java mapping for enum types

@

An IDL enum type is mapped to a Java class that implements the Java interface `IDLEntity`. The class for the `enum` type statically declares the enumeration literals for the type. Each instance of the class has a private `__value` field, which can be accessed by a public `value()` method. The `from_int()` method returns the (unique) enum literal object for a given integer. Type `enum` has attributes `name`, `label`, and `numLabels`. The `label` attributes are the enumeration literals for the type and `numLabels` is how many of them there are.

```
<<{enum} class declaration>>=
public class <<{{enum}.name}>> implements
org.omg.CORBA.portable.IDLEntity {

    // declaration of the enumeration literals
    <<{{enum}.label} declarations>>

    public int value() {
        return __value;
    }

    // get enum literal with specified value
    public static <<{{enum}.name}>> from_int(int i){
        switch(i) {
        <<case statement for {{enum}.label}>>
        default:
            throw new org.omg.CORBA.BAD_PARAM(
               "Enum out of range: [0.." +
               (<<{{enum}.numLabels}>> - 1) + "]: " + i);
        }
    }

    // constructor
    protected <<{{enum}.name}>>(int value) {
        this.__value = value;
    }

    // private fields
    private int __value;
}
```

@

The case statement in the `from_int()` method has to return the unique object corresponding to the given int. To see how we program that, first let's look at the mapping for labels (literals). The label type has two attributes. One is its value, the integer corresponding to the label (0, 1, 2, etc.). The other is the name of the enum type.

The class has two static members per enumeration literal. One of these provides a name for the numeric value associated with the literals (0, 1, 2, ...) and the other is a public static final instance of the enumeration literal. In effect, there is only one instance of any enumeration literal, so the default Java `equals()` and `hash()` methods will work correctly on enumeration literals.

The numeric (integer) value has a special name. The object takes the name of the literal.

```
<<{label} intval>>=
_<<{{label}.name}>>
```

@

```
<<{label} declarations>>=
<<{label} value declaration>>
<<{label} object declaration>>
```

@

Values of assigned sequentially starting with zero, but not by this code. We assume that the values are assigned by the IDL compiler. Each label has an attribute that gives its value.

```
<<{label} value declaration>>=
public static final int <<{label} intval>> =
     <<{{label}.ordinal}>>;
```

@

We create one static instance of the enum `label`. It is of the enumeration type, and has the correct value. This ensures that the default `java.lang.Object` implementation of `equals()` and `hash()` will work correctly.

```
<<{label} object declaration>>=
public static final <<{{{label}.type}.name}>>
<<{{label}.name}>> =
    new <<{{enum}.name}>>(<<{label} intval>>);

<<case statement for {label}>>=
case <<{label} intval>>:
    return <<{{label}.name}>>;
```

@

The holder class.

In the IDL to Java mapping, each IDL type gets a holder class, which provides support for CORBA `out` and `inout` parameter passing, by providing a place to "hold" the returned value. The holder class's name is the enum's mapped Java classname with "Holder" appended to it.

```
<<{enum} holder name>>=
<<{{enum}.name}>>Holder
```

@

```
<<{enum} holder class declaration>>=
final public class <<{enum} holder name>>
implements org.omg.CORBA.portable.Streamable {
    public <<{{enum}.name}>> value;
    public <<{enum} holder name>>() {
    }
    public <<{enum} holder name>>(<<{{enum}.name}>> value) {
        this.value = value;
    }
    public void _read(
        org.omg.CORBA.portable.InputStream input) {
        value = <<{enum} helper name>>.read(input);
    }
    public void _write(
        org.omg.CORBA.portable.OutputStream output) {
        <<{enum} helper name>>.write(output, value);
    }
    public org.omg.CORBA.TypeCode _type() {
        return <<{enum} helper name>>.type();
    }
}
```

@

The helper class provides static methods needed to manipulate an IDL type. It is generated for every IDL type, and here we only care about its name.

```
<<{enum} helper name>>=
<<{{enum}.name}>>Helper
```

@

Both the class file and the holder file are in a package directory. For IDL type foo, the Java mapping includes three files: foo.java, fooHelper.java, and fooHolder.java. (Helper classes are defined similarly for all types, so the Helper code would be generated by the mapping for IDL types, not the mapping for enums.)

Unfortunately, `{enum}` refers to the unique name of the enum type, which looks something like "module::module2::foo," so the file name will end up as "module::module2::foo.java" instead of "foo.java". Therefore, we will create a single output file that concatenates the output files, like this:

```
<<*>>=
<<files for {enum}>>

<<files for {enum}>>=
<<{enum}.java>>
<<{enum}Holder.java>>
```

@

The file Foo.java contains the package declaration, imports, and a class declaration. In this case, no imports are needed.

```
<<{enum}.java>>=
<<package declaration for {enum}>>

<<{enum} class declaration>>

<<{enum}Holder.java>>=
<<package declaration for {enum}>>

<<{enum} holder class declaration>>
```

@

The name of the package must be provided as an attribute.

```
<<package declaration for {enum}>>=
package <<{{enum}.fullPackageName}>>;
```