## SOMobjects Developer Toolkit

# Emitter Framework Guide and Reference

**A Framework of the System Object Model**

**Version 2.1**
**October 1994**

**Note:** Before using this information and the product it supports, be sure to read the trademark information under “Trademarks” on page ix.

## Version 2.1 (October 1994)

**The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law:** INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THE PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

It is possible that this publication may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

Requests for technical information about IBM products should be made to your IBM Authorized Dealer or your IBM Marketing Representative.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577.

COPYRIGHT LICENSE: This publication contains printed sample application programs in source language, which illustrate AIX, OS/2, or Windows programming techniques. You may copy and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the AIX, OS/2, or Windows application programming interface.

Each copy of any portion of these sample programs or any derivative work, which is distributed to others, must include a copyright notice as follows: “©(your company name) (year) All Rights Reserved.”

However, the following copyright notice protects this documentation under the Copyright laws of the United States and other countries which prohibit such actions as, but not limited to, copying, distributing, modifying, and making derivative works.



The term “IBM” is a registered trademark and “SOMobjects” and “System Object Model” are trademarks of International Business Machines Corporation.

Notice to US Government Users – Documentation related to restricted rights – Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

# Emitter Framework Guide and Reference

## Contents

# About This Book

This book describes the **Emitter Framework** of the **SOMobjects Developer Toolkit**. It explains how programmers can easily write a new *emitter* to be used with the SOM Compiler to produce special-purpose output from an IDL interface specification.

In addition to this book, refer to the *SOMobjects Developer Toolkit Users Guide* for general information about the System Object Model (SOM) and the SOM Compiler, as well as to the *SOMobjects Developer Toolkit Programmers Reference Manual* for specific information about the classes, methods, functions, and macros supplied with the SOMobjects Toolkit.

## How This Book Is Organized

This book contains two parts: an introductory users' guide and a reference manual. The users' guide gives an overview of the emitter-writing process and the classes and methods provided by the Emitter Framework. The second part is a reference manual for the classes, methods, and functions provided by the Emitter Framework. The reference pages describe the classes in alphabetical order, with the methods of each class given in alphabetical order following their corresponding class. Lastly, the functions are given in alphabetical order. The reference page for a **class** contains the following topics:

| | |
|---|---|
| **Description**: | A description of the class. |
| **File Stem**: | The file stem for the class's IDL interface specification (.idl) file and its usage binding (.h/.xh) files. |
| **Base**: | The class's direct base (parent) classes. |
| **Ancestor Classes**: | The class's ancestor (indirect base) classes. |
| **Metaclass**: | The class's metaclass. |
| **New Methods**: | The names of the methods that the class introduces (grouped roughly according to purpose). Each new method is documented on a separate reference page. |
| **Overriding Methods**: | The names of the methods that the class overrides from ancestor classes |

The reference pages describe the methods of each class in alphabetical order following the corresponding class. The reference page a **method** contains the following topics:

| | |
|---|---|
| **Purpose**: | The purpose of the method in brief. |
| **Syntax**: | The method's C/C++ procedure prototype (which includes the method procedure's return type and the names and types of its parameters). The in/out/inout keywords associated with each of the method's parameters in the method's IDL declaration are also shown. These keywords are shown for information only; they are not actually present in the method procedure prototype. |
| **Description**: | A description of the method's use. |
| **Parameters**: | A description of each of the method procedure's parameters**.** |
| **Return Value**: | A description of the method's return value. |
| **Example**: | An example of using or overriding the method, if available. Although methods of SOM classes are language neutral (that is, they can be invoked from any programming language that can use SOM), the examples given here are written in C. |
| **Original Class**: | The name of the class that introduces the method (which is documented separately in this book)**.** |
| **Related Information**: | Related methods that can be found in this book. |

The reference page for a **function** contains the following topics:

| | |
|---|---|
| **Purpose**: | The purpose of the function in brief. |
| **Syntax**: | The function's prototype (which includes the return type and the names and types of the parameters). |
| **Description**: | A description of the function's use. |
| **Parameters**: | A description of each of the function's parameters**.** |
| **Return Value**: | A description of the function's return value. |
| **Example**: | An example of using the function, if available. |
| **Related Information**: | Related methods and functions that can be found in this book. |

## Who Should Read This Book

This book is for the professional programmer using C, C++, or another language to implement a special-purpose *emitter* to be used as a back end to the SOM Compiler.

This book assumes that you are an experienced programmer and that you have a general familiarity with the basic notions of object-oriented programming. Practical experience using an object-oriented programming language is helpful, but not essential. This book also assumes familiarity with the System Object Model (SOM) , the SOM Interface Definition Language (SOM IDL), and the SOM Compiler. These topics are discussed in the *SOMobjects Developer Toolkit Users Guide*.

# Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

AIX
IBM
Operating System/2
OS/2
OS/2 Workplace Shell
RISC System 6000
SOMobjects
System Object Model

For convenience, the acronym “SOM” is used in this publication to reference the technology of the System Object Model, and the term “SOM Compiler” is used to reference the compiler of the System Object Model.

Each of the following terms used in this publication is a trademark of another company:

| | |
|---|---|
| Intel | Intel Corporation |
| IPX | Novell Corporation |
| Lotus 1-2-3 | Lotus Development Corporation |
| Microsoft EXCEL | Microsoft Corporation |
| Microsoft Windows | Microsoft Corporation |
| NetWare | Novell Corporation |
| Objective-C | The Stepstone Corporation |
| Smalltalk | Digitalk Inc. |

The term “ANSI C” used throughout this publication refers to American National Standard X3.159–1989.

The term “CORBA” used throughout this publication refers to the Common Object Request Broker Architecture standards promulgated by the Object Management Group, Inc.

# Emitter Framework Guide and Reference

## 1.1 Introduction

The purpose of the SOM Compiler is to translate an IDL interface definition into one or more other useful forms. For example, an interface definition can be translated into a programming language binding file, an implementation template file, a documentation file, a description that can drive a class browser, or a pretty-printed interface specification. Given the number of programming languages with which SOM can be used and the many development-support tools that can leverage object interface definitions, the SOM Compiler needs to produce a large number of output forms. Therefore, an important structural feature of the SOM Compiler is that it minimizes the effort involved in developing and maintaining new compiler "back-ends."

Figure 1 shows how the SOM Compiler is structured to utilize a number of "emitters". Each emitter produces a different output file. As shown in the figure, the only part of the SOM Compiler that varies with different output targets is the emitter. A new emitter must be developed and maintained for each output target, but the IDL parser remains unchanged.

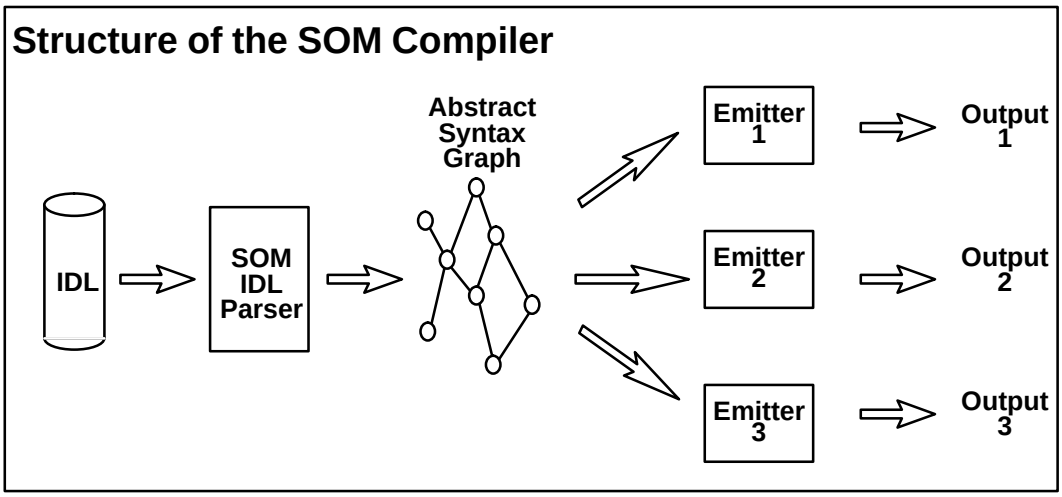


Figure 1. The structure of the SOM Compiler

To make it easier to develop new emitters for use with the SOM Compiler, the SOMobjects Toolkit provides a collection of classes called the Emitter Framework. The Emitter Framework consists of several support classes and a general emitter class that can be subclassed to produce a specific emitter. The SOMobjects Toolkit also provides the **newemit** facility for automatically generating new emitters. This automatically generated emitter is then easily customized as needed for a particular output format.

The goals of the emitter classes are to provide an object-oriented framework for emitter development that:

- Insulates new emitter code from changes to SOM's interface definition language (SOM IDL).

- Separates design concerns, to improve the ease of development and maintenance of emitters. The designers of a new emitter should not have to understand the full emitter process. Rather, they simply override methods from one of the Emitter Framework classes. The logic of the Emitter Framework causes the methods to be invoked at the correct time.
- Supports a template facility that allows developers to specify the form of an output file in a highly readable and maintainable manner (as a template).
- Breaks up the control logic for output-file construction into small, easily maintained (and frequently reusable) units.

## 1.2 Structure of the Emitter Framework

The Emitter Framework is structured as depicted in Figure 2.

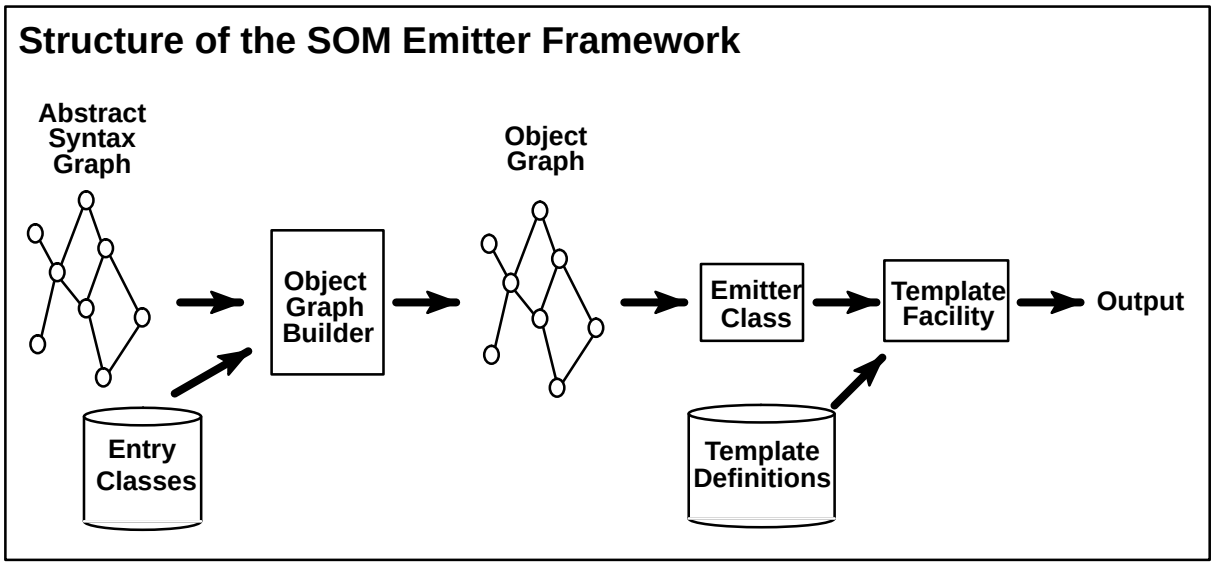


Figure 2. The structure of the SOM Emitter Framework

### The object graph builder

The input to the Emitter Framework is an abstract syntax graph of data structures. As shown in (the earlier) Figure 1, the abstract syntax graph is produced by the SOM Compiler. The SOM Compiler's IDL parser reads the input .idl file and converts the interface descriptions that are included in it (either directly or indirectly) into the abstract syntax graph. Each node of the abstract syntax graph represents a syntactic unit of the interface definition (a method declaration, a parameter, an attribute declaration, and so forth).

Once the abstract syntax graph has been constructed by the SOM Compiler, the object graph builder, the front-end of the Emitter Framework, traverses the abstract syntax graph, building an isomorphic graph of "entry" objects. An *entry* object represents a syntactic unit of the interface definition. For example, a **SOMTClassEntryC** object represents an entire interface definition, **SOMTMethodEntryC** objects represent method declarations, **SOMTParameterEntryC** objects represent method parameter declarations, and so on.

Both the IDL parser and the object graph builder are closed parts of the Emitter Framework; they cannot be extended or modified by programmers using the Emitter Framework. Two important forms of flexibility are provided, however.

1. SOM IDL syntax provides for an open-ended set of "modifiers" that can be associated with most syntactic elements in an interface definition. Modifiers specified in a .idl file are accessible to emitters written using the Emitter Framework.
2. Before the object graph builder is run, an emitter can cause some or all of the Emitter Framework classes to be "shadowed" (effectively replaced by user-defined subclasses). When the programmer shadows a particular *entry class* (**SOMTClassEntryC**, etc.), the object graph builder uses instances of the programmer's subclass of that entry class, rather than instances of the original entry class. Thus, the programmer can modify the object graph even though the object graph builder creates all the entry class instances in code that is not open to the programmer.

## The entry classes

The entry classes are used to construct the object graph produced by the object graph builder described above. Each node of the object graph is an instance of one of the entry classes. Each instance of an *entry class* represents one syntactic unit of an IDL interface definition — that is, one piece or one "entry" from the complete IDL interface definition. An entry object serves two important functions:

1. Holding information about the corresponding syntactic element of an IDL specification.
2. Defining symbols that can be used as placeholders in an emitter's *output template*.

The subsequent major section "Emitter Framework Classes" describes the entry classes and the use of symbols in more detail.

## The emitter class

The emitter class, **SOMTEmitC**, is the class that drives the process of producing an output file. Constructing a new emitter requires creating a new subclass of **SOMTEmitC** and overriding one or more of its methods (principally, the **somtGenerateSections** method) so that it produces the desired output. The new emitter is run by creating an instance of the new subclass of **SOMTEmitC** and invoking the **somtGenerateSections** method on it.

## The template class and template definitions

The template class, **SOMTTemplateOutputC**, is used by an emitter to produce output. It recognizes template descriptions of the output, so that most of the information about how the output file should look can be placed in a template definition and does not need to be embedded in the emitter code.

The template definitions describe the content and format of various *sections* of the output file, and the emitter controls which of these sections are output and in what order. The emitter calls on an instance of the **SOMTTemplateOutputC** class to have a particular section produced from that section's template definition.

The following section discusses the components of the Emitter Framework and describes the recommended procedure for producing a new emitter.

## 1.3 Emitter Framework Classes

The Emitter Framework consists almost entirely of classes.

- The *emitter class* (**SOMTEmitC**) manages the overall activity of an emitter, obtaining information from the entry objects and directing the template object to produce specific sections.
- The *template class* **(SOMTTemplateOutputC)** manages the output of specific sections to the *target* (output) file. It provides a template facility to make the specification of the output file simple.
- Most of the classes are the *entry classes* **(SOMTEntryC** and its subclasses), each of which represents some syntactic unit of an IDL definition.

The only Emitter Framework classes that users will explicitly instantiate (create instances of) are their own subclasses of the emitter class (**SOMTEmitC**). The remaining classes are instantiated automatically by the Emitter Framework.

The remainder of this section discusses each of these classes. For more information, the Reference portion of this book provides detailed information on all of the attributes and methods supported by each class in the Emitter Framework.

### The emitter class (SOMTEmitC)

**SOMTEmitC** is the primary class of the Emitter Framework. It provides overall control for the emitting process. An emitter writer will always need to subclass this class or one of its subclasses, override some of its methods (primarily the **somtGenerateSections** method), and perhaps add a few new methods. The next section describes this process in more detail.

An instance of **SOMTEmitC** (an emitter) has as attributes a target file, a target class or target module, a template object, and a name, as follows:

- The *target file* is the file to which output will be directed.
- The *target class* (the class about which information will be emitted) is represented by an object of class **SOMTClassEntryC**. This object is the root of the object graph built by the object graph builder when the emitter is invoked on a class definition (see Figure 2).
- The *target module* (the module about which information will be emitted) is represented by an object of class **SOMTModuleEntryC**. This object is the root of the object graph built by the object graph builder when the emitter is invoked on a module definition (see Figure 2).
- The *template object* of the emitter (an instance of **SOMTTemplateOutputC**) maintains the symbol table and controls the format and content of the sections that the emitter produces. (The emitter itself controls which sections are actually emitted and their order.) The template object is initialized from the output template file.
- The *emitter name* is the name by which the emitter is invoked via the "–s" option of the "sc" command. The emitter name is used to determine which passthrus in the input .idl file are directed to that emitter.

The **SOMTEmitC** class provides methods for:

- Opening the output template file (**somtOpenSymbolsFile**),
- Getting the value of *global modifiers* (those specified via the "–m" option of the "sc" command),

- Setting standard symbols associated with the target class and its metaclass (**somtFileSymbols**), as well as setting standard section-name symbols (**somtSetPredefinedSymbols**),
- Generating the output file from the output template (**somtGenerateSections**). The **somtGenerateSections** method is the primary method that a new emitter will override from **SOMTEmitC**. This method controls which sections will be emitted and in what order.

The **SOMTEmitC** class also provides methods for emitting different standard sections of an output file. The standard sections are as follows, with the default section name given in parentheses:

**Prolog** Describes text to be emitted before any other sections (**prologS**).

**Base Includes** Determines how base (parent) class #include statements are emitted (**baseIncludesS**).

**Meta Include** Determines how a metaclass #include statement is emitted (**metaIncludeS**).

**Class** Determines what information about the class as a whole is emitted (**classS**).

**Base** Determines what information about the base (parent) classes of a class is emitted (**baseS**).

**Meta** Determines what information about the class's metaclass is emitted (**metaS**).

**Constant** Determines what information about user-defined constants is emitted (**constantS**).

**Typedef** Determines what information about user-defined types is emitted (**typedefS**).

**Struct** Determines what information about user-defined structs is emitted (**structS**).

**Union** Determines what information about user-defined unions is emitted (**unionS**).

**Enum** Determines what information about user-defined enumerations is emitted (**enumS**).

**Attribute** Determines what information about the class's attributes is emitted (**attributeS**).

**Methods** Determines what information about the methods of a class is emitted (**methodsS**). More specialized method sections can be specified using **inheritedMethodsS** or **overrideMethodsS**).

**Release** Determines how information about the release order statement of a class definition is emitted (**releaseS**).

**Passthru** Determines what information about passthru statements is emitted (**passthruS**).

**Data** Determines what information about internal instance variables of a class is emitted (**dataS**).

**Interface** Determines what information about the interfaces in a module is emitted (**interfaceS**).

**Module** Determines what information about a module is emitted (**moduleS**).

**Epilog** Describes text to be emitted after all other sections are emitted (**epilogS**).

Some sections apply to a variable number of items that must be dealt with iteratively. This can be true of the *base* section (since a class can have more than one base class), as well as the sections for *base includes, data*, *passthru*, *attribute, constant, typedef, struct, union, enum, interface, module,* and *method*. These repeating sections can be preceded by a *prolog* (information to be emitted *prior* to iterating through the items), and followed by an *epilog* (information to be emitted *after* iterating through the items). Names for the standard prolog and epilog sections are as follows:

basePrologS, baseEpilogS, baseIncludesPrologS, baseIncludesEpilogS, constantPrologS, constantEpilogS, typedefPrologS, typedefEpilogS, structPrologS, structEpilogS, unionPrologS, unionEpilogS, enumPrologS, enumEpilogS, passthruPrologS, passthruEpilogS, dataPrologS, dataEpilogS, attributePrologS, attributeEpilogS, methodsPrologS, methodsEpilogS, interfacePrologS, interfaceEpilogS, modulePrologS, and moduleEpilogS**.**

The **SOMTEmitC** class provides methods for emitting each of the sections described above. For example, the **somtEmitProlog** method emits the prologS section, the **somtEmitClass** method emits the classS section, and so on. For repeating sections, the **SOMTEmitC** class provides *scanning methods*. These scanning methods first emit the appropriate prolog section, then iterate through the appropriate items in the interface definition, emitting the appropriate section for each item, then emit the appropriate epilog section.

The following scanning methods are provided by **SOMTEmitC:**

**somtScanBases, somtScanBasesF, somtScanConstants, somtScanTypedefs, somtScanStructs, somtScanUnions, somtScanEnums, somtScanAttributes, somtScanMethods, somtScanData, somtScanDataF, somtScanPassthru, somtScanInterfaces,** and **somtScanModules.**

(The **somtScanBasesF, somtScanDataF,** and **somtScanMethods** methods accept a *filter* argument, for selective scanning.) The topic entitled “The section-name symbols” at the end of this chapter lists all the section-emitting methods defined by **SOMTEmitC** and the sections that they output. The Reference portion of this book describes each section-emitting method in more detail.

User-defined subclasses of **SOMTEmitC** can override the section-emitting methods to change the way that a particular section is emitted. They can also define new section-emitting methods (see “Customizing section-emitting methods” in section 5 of this chapter).

Finally, the **SOMTEmitC** class provides several filter methods. These methods return TRUE or FALSE depending on some characteristic of a specified entry object. For example, the **somtNew** method determines whether the specified method is introduced by the emitter’s target class. These filter methods can be used as arguments to the **somtScanMethods** method to control which methods are processed in a repeating section.

Filter methods provided by **SOMTEmitC** include **somtNew, somtImplemented, somtOverridden, somtInherited, somtAll, somtNewProc, somtNewNoProc,** and **somtVA.** The Reference portion of this book describes each of these methods in more detail.

## The template output class (SOMTTemplateOutputC)

The **SOMTTemplateOutputC** class handles as much as possible of the formatting part of emitter writing, largely by using symbol-based output templates. A *symbol* is a name used to represent a corresponding value. For example, the symbol (or symbol name) "className" is recognized by the Emitter Framework as representing the name of the target class.

An emitter writer uses symbol names as placeholders in a text *template* that patterns the desired output. The template object (of class **SOMTTemplateOutputC**) takes a text template containing symbol names and produces output by substituting data for the symbols that occur in the text template. The values that replace the symbol names come from a symbol table maintained by the template object.

The template file is divided into *sections* that specify the desired output for each syntactic unit of the input IDL specification. To generate a particular section of an output file, an emitter first sets (defines) the values of appropriate symbols in its template-object's symbol table, and then specifies to the template object the name of a section to be output. This design results a good separation between decision logic and format specification. Also, because the format specification is isolated, its readability and maintainability are greatly enhanced.

Following is an example fragment of a text template containing two template sections, "classS" and "metaS". The text template is stored in a *template file* associated with the emitter. (The subsequent paragraphs provide further explanation for preparing the text template.)

```
:classS
class: <className><, classMods, ...>;
?<-- classComment>
:metaS
metaclass: <metaName>
```

New sections are denoted by lines that begin with a colon. The above fragment contains two sections, "classS" and "metaS". (By convention, section names end in capital "S".) An emitter uses the section name to specify to the template object which part of the output file to emit. Lines that begin with a question mark are emitted only if at least one symbol appearing on the line is defined with a nonblank value. Other lines are emitted unconditionally.

Symbols (symbol names) are specified in a template file in angle brackets. Thus, the template above contains the symbols "className", "classMods", "classComment", and "metaName". (A backslash can be used to escape an angle bracket when it is not intended to indicate a symbol.) When a template section is emitted, symbols are replaced with their values. If the symbol has no value, then the symbol is replaced by the string "symbol <...> is not defined", but no error is raised.

In addition to simple symbol substitution, two forms of complex symbol substitution are supported: list substitution and comment substitution. Each of these involves special syntax, as follows.

Comment substitution is specified with two dashes preceding the symbol name (for example, `<-- symbolName>`). When comment substitution is used to emit a symbol, the symbol's value is emitted in comment form. The emitter controls the format for comments by setting the values of its template object's **somtCommentStyle** and **somtCommentNewline** attributes:

- The **somtCommentStyle** attribute determines whether comments are emitted with "– –" at the start of each line (**somtDashesE**), with "//" at the start of each line (**somtCPPE**), in simple C style with each line wrapped in "/*" and "*/" (**somtCSimpleE**), or in block C style with a leading "/*", then a "*" on each line and a final "*/" (**somtCBlockE**).
- The **somtCommentNewline** attribute is a boolean that determines whether the comment starts on a new line.

List substitution replaces a symbol with its value expressed in list form, using specified delimiters. The symbol's value must consist of a sequence of items, separated by newline characters. The list substitution specification consists of two pieces of information in addition to the symbol name: the prefix to put in front of non-empty lists, and the delimiter to put between list items.

All characters before the symbol name are taken as the prefix, and all characters after the symbol name and before the required "..." (which indicates that list substitution is to be used) are taken as the separator characters. Thus <: symbolName, ...> specifies a prefix of ": " and a separator of ", ". The prefix and separator characters must consist of blanks, commas, colons, and semicolons. The value of the template object's **somtLineLength** attribute controls how many list items are emitted on each line.

Within an output template, tabbing can be specified by <@*dd*>, where *dd* is a valid positive integer representing a column number. After a <@*dd*> is encountered in the output template, the next character emitted will appear in the specified column.

Emitting the "classS" and "metaS" sections from the above template, using the following IDL specification as input:

```
#include <somobj.idl>
#include <mhello.idl>

interface Hello : SOMObject /* This is the interface for Hello. */
{
    implementation {
    metaclass = M_Hello;
    functionprefix = "hello_";
    filestem = hello;
    . . .
    };
};
```

would produce the following output:

```
class: Hello, functionprefix = hello_, filestem = hello;

// This is the interface for Hello.
metaclass: M_Hello
```

(The formatting of comments varies, depending on the attributes of the emitter's template.)

The **SOMTTemplateOutputC** class provides methods for:

- Setting and getting the value of symbols in a template object's symbol table (**somtGetSymbol, somtSetSymbol, somtSetSymbolCopyName, somtSetSymbolCopyValue, somtSetSymbolCopyBoth, somtCheckSymbol,** and **somtExpandSymbol**),
- Emitting a particular section of the output template (**somtOutputSection**),
- Emitting a comment (**somtOutputComment**),
- Reading the output template file (**somtReadSectionDefinitions**), and others.

The topic "Defining new symbols" in section 5, "Writing an Emitter — Advanced Topics," describes how to use the symbol-setting methods to define new symbols.

## The entry classes (SOMTEntryC, SOMTClassEntryC, ...)

The purpose of these classes is to hide the syntax of the .idl file. They return information about an IDL interface definition in a way that is neutral to the source syntax of the IDL definition and to the nature of the emitter in which the information will be used.

The entry classes are arranged into the class hierarchy shown in Figure 3.

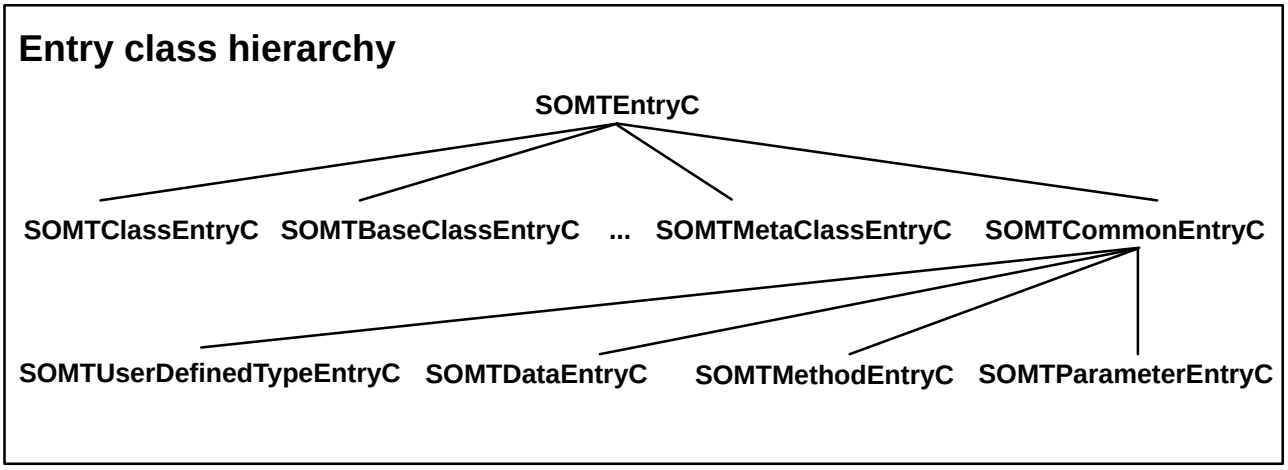


Figure 3. The entry class hierarchy.

With the exception of **SOMTEntryC** and **SOMTCommonEntryC**, all of the entry classes correspond to a specific unit of information in an IDL interface definition. This correspondence is summarized in the following topics.

For more information on a particular entry class or method, see the Reference portion of this book.

### SOMTEntryC

The **SOMTEntryC** class provides **attributes** for accessing the name of an entry (**somtEntryName, somtIDLScopedName,** and **somtCScopedName**), its entry type (that is, whether it represents a class, method, attribute, typedef, etc.) (**somtElementType** and **somtElementTypeName**), its comment (**somtEntryComment**), the line number in the .idl file where the entry is defined (**somtSourceLineNumber**), its type code (**somtTypeCode**), and whether the entry represents a reference to an entry rather than its definition (**somtIsReference**).

The **SOMTEntryC** class also provides methods for accessing the SOM IDL modifiers specified in the "implementation" section of an "interface" statement. Included are the methods:

**somtGetModifierValue,**
**somtGetFirstModifier,**
**somtGetNextModifier,**
**somtFormatModifier,** and
**somtGetModifierList**

When invoked on an instance of **SOMTClassEntryC**, these methods pertain to the class's modifiers; when invoked on an instance of **SOMTMethodEntryC**, they pertain to the method's modifiers, and so on.

The **SOMTEntryC** class also provides the **somtSetSymbolsOnEntry** method, which can be used to create symbols and define their corresponding values for use in the output template. For example, **SOMTClassEntryC**'s implementation of **somtSetSymbolsOnEntry** establishes the symbol "className" containing the name of the current class, **SOMTMethodEntryC**'s implementation of **somtSetSymbolsOnEntry** defines the "methodName" symbol, and so on. [Symbols are described in more detail under the later topic "The template output class (SOMTTemplateOutputC)".]

## SOMTCommonEntryC

Entry objects that an emitter uses are actually instances of one of the subclasses of **SOMTCommonEntryC**, rather than of **SOMTCommonEntryC** itself. These subclasses are the classes **SOMTMethodEntryC**, **SOMTDataEntryC**, **SOMTUserDefinedTypeEntryC,** and **SOMTParameterEntryC**.

The **SOMTCommonEntryC** class provides **attributes** and **methods** for obtaining information about the **type** of a method, parameter, user-defined type, attribute declarator, struct member declarator, or instance variable. For example, it provides the attribute **somtTypeObj** whose value is a pointer to a **SOMTEntryC** object representing the type, the attribute **somtType** that gives a string representation of the type, the attribute **somtArrayDimsString** that indicates array dimensions, and the attribute **somtPtrs** that gives the number of stars associated with a pointer type.

The **SOMTCommonEntryC** class also provides *methods* for accessing type information: **somtGetFirstArrayDimension, somtGetNextArrayDimension, somtIsArray,** and **somtIsPointer**.

## SOMTClassEntryC

A **SOMTClassEntryC** object anchors the entire interface definition for a class. That is, all the parts of a class's interface definition are reachable from the class entry (**SOMTClassEntryC** object) that represents it. When an emitter is run on a class's interface definition (rather than on a module), the emitter has a distinct class entry called the *target class entry* which represents that class.

The **SOMTClassEntryC** class provides **attributes** corresponding to the following characteristics of an IDL interface specification:

- Its source file name (**somtSourceFileName**),
- Its metaclass (**somtMetaClassEntry**),
- The class this class is a metaclass for, if any (**somtMetaclassFor**),
- Whether the entry represents a forward declaration of the class, rather than its definition (**somtForwardRef**),
- The module that contains the class, if any (**somtClassModule**),
- The number of methods the class introduces (**somtNewMethodCount**) or overrides (**somtOverrideMethodCount**),
- The number of static methods the class introduces (**somtStaticMethodCount**),
- The number of procedure methods the class introduces (**somtProcMethodCount**),
- The number of variable argument methods the class introduces (**somtVAMethodCount**),
- The number of parent (base) classes (**somtBaseCount**).

The class also provides **methods** for accessing each of a class's:

- parent (base) classes (**somtGetFirstBaseClass** and **somtGetNextBaseClass**),
- release order names (**somtGetFirstReleaseName, somtGetNextReleaseName,** and **somtGetReleaseNameList**),
- data items (**somtGetFirstData** and **somtGetNextData**),
- passthrus (**somtGetFirstPassthru** and **somtGetNextPassthru**),
- methods (**somtGetFirstMethod, somtGetNextMethod, somtGetFirstInheritedMethod,** and **somtGetNextInheritedMethod**),

- constants (**somtGetFirstConstant** and **somtGetNextConstant**),
- attributes (**somtGetFirstAttribute** and **somtGetNextAttribute**),
- typedefs (**somtGetFirstTypedef** and **somtGetNextTypedef**),
- structs (**somtGetFirstStruct** and **somtGetNextStruct**),
- unions (**somtGetFirstUnion** and **somtGetNextUnion**),
- enumerations (**somtGetFirstEnum** and **somtGetNextEnum**), and
- sequences (**somtGetFirstSequence** and **somtGetNextSequence**).

**SOMTClassEntryC** also provides methods for accessing all type/constant definitions in the order in which they were defined, including structs, unions, enumerations. These methods are **somtGetFirstPubdef** and **somtGetNextPubdef**. Finally, the **SOMTClassEntryC** class provides filter methods for determining whether a method is new (**somtFilterNew**) or overridden (**somtFilterOverridden**).

## SOMTBaseClassEntryC

Every class entry holds a pointer to a base class entry (**SOMTBaseClassEntryC** object) for each of the class's direct base (parent) classes. The base class entry is *not* the class entry for a base class. Rather, it is an object that has an attribute (**somtBaseClassDef**) whose value is the class entry for the base class.

## SOMTMetaClassEntryC

Every class entry holds a pointer to its metaclass entry (**SOMTMetaClassEntryC** object) if the class #includes the .idl file for its metaclass. A metaclass entry is like a base class entry in that it is *not* the class entry for the metaclass. Rather, it is an object that has an attribute (**somtMetaClassDef**) whose value is the class entry for the metaclass. The metaclass entry also has an attribute (**somtMetaFile**) that specifies the file in which the metaclass's interface is defined.

## SOMTModuleEntryC

A **SOMTModuleEntryC** object represents a module within an IDL specification. It provides methods for accessing each of the module's:

- interfaces (**somtGetFirstInterface** and **somtGetNextInterface**),
- nested modules (**somtGetFirstModule** and **somtGetNextModule**),
- constants (**somtGetFirstModuleConstant** and **somtGetNextModuleConstant**),
- typedefs (**somtGetFirstModuleTypedef** and **somtGetNextModuleTypedef**),
- structs (**somtGetFirstModuleStruct** and **somtGetNextModuleStruct**),
- unions (**somtGetFirstModuleUnion** and **somtGetNextModuleUnion**),
- enumerations (**somtGetFirstModuleEnum** and **somtGetNextModueEnum**), and
- sequences (**somtGetFirstModuleSequence** and **somtGetNextModuleSequence**).

**SOMTModuleEntryC** also provides methods for accessing the all of the above definitions in the order in which they were defined. These methods are **somtGetFirstModuleDef** and **somtGetNextModuleDef**.

## SOMTPassthruEntryC

Every class entry holds a pointer to a passthru entry (**SOMTPassthruEntryC** object) for each passthru specification in the implementation section of the class's SOM IDL interface specification. Each passthru entry has attributes representing the target

(**somtPassthruTarget**), the target language (**somtPassthruLanguage**), and the passthru's contents (**somtPassthruBody**), as well as a method (**somtIsBeforePassthru**) for determining whether the passthru is a "before" or "after" passthru.

## SOMTTypedefEntryC

Every class entry holds a pointer to a typedef entry (**SOMTTypedefEntryC** object) for each typedef introduced within the class's interface specification and for each member of a user-defined struct. Each typedef entry provides an attribute representing the base type (**somtTypedefType**) of the typedef and methods for accessing each of the declarator names of the typedef (**somtGetFirstDeclarator** and **somtGetNextDeclarator**). Because a single typedef may have several declarators (that introduce several user-defined types), the **somtTypedefType** attribute of a typedef gives only the *base* type of the user-defined types; to get the full type, users should access each declarator in turn and get its **somtType** attribute.

## SOMTDataEntryC

Every class entry holds a pointer to a data entry (**SOMTDataEntryC** object) for each of the data members (internal instance variables) specified in the implementation section of the class's interface definition, and for each attribute declarator or struct member declarator. The **SOMTDataEntryC** class provides an attribute, **somtIsSelfRef**, that indicates whether a struct member declarator is self-referential (pointing to the same type of structure for which it is a declarator).

## SOMTAttributeEntryC

Every class entry holds a pointer to an attribute entry (**SOMTAttributeEntryC** object) for each of the attribute definition statements within the class's interface specification. Each attribute entry has attributes representing the base type (**somtAttribType**) and whether the attribute is readonly (**somtIsReadonly**). It also provides methods for accessing the attribute declarators (**somtGetFirstAttributeDeclarator** and **somtGetNextAttributeDeclarator**) and their get/set methods (**somtGetFirstGetMethod, somtGetNextGetMethod, somtGetFirstSetMethod,** and **somtGetNextSetMethod**).

Because a single attribute definition statement may have several declarators (that introduce several attributes), the **somtAttribType** attribute gives only the *base* type of the attributes being defined; to get the full type, users should access each declarator in turn and get its **somtType** attribute.

## SOMTMethodEntryC

A class entry holds a pointer to a method entry (**SOMTMethodEntryC** object) for each of the methods the class supports (both new and inherited methods). Each method entry has attributes representing:

- The C/C++ form of the method's return type (**somtCReturnType**),
- Whether the method has a **va_list** parameter (**somtIsVarargs**),
- For overriding methods, the class whose implementation is being overridden (**somtOriginalClass**) and the method being overridden (**somtOriginalMethod**),
- Whether the method is "oneway" (**somtIsOneway**),
- The number of arguments to the method (**somtArgCount**), and
- The context string literals of the method (**somtContextArray**).

The **SOMTMethodEntryC** class also provides methods for getting the method's parameters (**somtGetFirstParameter, somtGetNextParameter, somtGetNthParameter, somtGetIDLParamList, somtGetShortCParamList, somtGetFullCParamList, somtGetShortParamNameList, somtGetFullParamNameList**).

## SOMTParameterEntryC

Method entries contain a reference to a parameter entry (**SOMTParameterEntryC** object) for each of the explicit parameters to the method. (The receiver of the method does not have a corresponding parameter entry; neither do the **Environment** and **Context** parameters, if any.) Each parameter entry has an attribute (**somtParameterDirection**) that indicates whether it is an in, out, or inout parameter, and attributes that give the parameter's declaration within a prototype (**somtIDLParameterDeclaration** and **somtCParameterDeclaration**).

## SOMTConstEntryC

Every class entry holds a pointer to a constant entry (**SOMTConstEntryC** object) for each constant defined within the class's interface specification. Each constant entry has attributes that represent the type (**somtConstType** and **somtConstTypeObj**) and the value (**somtConstIsNegative, somtConstStringVal, somtConstNumVal, somtConstVal,** and **somtConstNumNegVal**) of the constant.

## SOMTEnumEntryC

Every class entry holds a pointer to an enum entry (**SOMTEnumEntryC** object) for each enumeration defined within the class's interface specification. Each enum entry provides methods for getting the enumerator names for the enumeration (**somtGetFirstEnumName** and **somtGetNextEnumName**).

## SOMTSequenceEntryC

Every class entry holds a pointer to a sequence entry (**SOMTSequenceEntryC** object) for each sequence defined within the class's interface specification. Each sequence entry has attributes representing the sequence's length (**somtSeqLength**) and type (**somtSeqType**).

## SOMTStringEntryC

Every class entry holds a pointer to a string entry (**SOMTStringEntryC** object) for each string defined within the class's interface specification. Each string entry has an attribute representing the string's length (**somtStringLength**)

## SOMTUnionEntryC

Every class entry holds a pointer to a union entry (**SOMTUnionEntryC** object) for each union defined within the class's interface specification. Each union entry provides an attribute representing the union's switch type (**somtSwitchType**) and methods for accessing each of its cases (**somtGetFirstCaseEntry** and **somtGetNextCaseEntry**).

## SOMTEnumNameEntryC

Every enumeration entry (of type **SOMTEnumEntryC**) holds a pointer to a **SOMTEnumNameEntryC** object for each enumerator name defined within it. Each **SOMTEnumNameEntryC** entry has attributes representing the enumerator name's value (**somtEnumVal**) and a pointer to the enumeration that defines the enumerator name (**somtEnumPtr**).

## SOMTStructEntryC

Every class entry holds a pointer to a struct entry (**SOMTStructEntryC** object) for each struct defined within the class's interface specification and for each exception the class defines. Each struct entry provides attributes that represent the class in which the struct was defined (**somtStructClass**) and whether the struct actually represents an exception (**somtIsException**), and methods for accessing each of the struct members (**somtGetFirstMember** and **somtGetNextMember**).

## SOMTUserDefinedTypeEntryC

Every class entry holds a pointer to a user-defined type entry (**SOMTUserDefinedTypeEntryC** object) for each type defined within the class's interface specification via a typedef statement. Each user-defined type entry provides attributes representing the typedef statement that defined the type (**somtOriginalTypedef**) and the base type of the user-defined type (**somtBaseTypeObj**). The **somtBaseTypeObj** attribute gives the primitive IDL type (float, short, char, etc.) that underlies a user-defined type, skipping over any intermediate user-defined types.

# 1.4 Writing an Emitter — the Basics

## The 'newemit' facility

The **newemit** emitter generator is a program, written using the Emitter Framework, that generates a complete, working emitter. This emitter is easily customized as needed.

The following steps outline the recommended approach to writing an emitter.

## Running the 'newemit' program

The command to execute the **newemit** program takes the following syntax:

**newemit** [ –C | –C++ ] <*className*> <*filestem*>

Required arguments are the name for a new subclass of **SOMTEmitC** to be created and a file stem for the subclass. The optional –C or –C++ (lowercase is also valid) specifies the language in which the emitter will be written; the default is C. The program produces the following files:

<*filestem*>.idl — An IDL definition of the subclass of **SOMTEmitC** with the specified name. This IDL definition specifies that the **somtGenerateSections** method will be overridden by the new subclass.

<*filestem*>.c or <*filestem*>.C or <*filestem*>.cpp — The C or C++ implementation file for the new subclass of **SOMTEmitC**. Initially, this implementation file has the same code for **somtGenerateSections** as defined by **SOMTEmitC**. (The C++ extension is .C on AIX or .cpp on OS/2.)

emit<*filestem*>.c or emit<*filestem*>.C or emit<*filestem*>.cpp — A C or C++ emitter driver program. Notice that the driver-program name is always "emit" followed by the specified file stem. (The C++ extension is .C on AIX or .cpp on OS/2.)

Makefile — A Makefile for creating a DLL for the new emitter.

<*filestem*>.efw — A sample output template file.

The <*filestem*>.c (or .C or .cpp) and <*filestem*>.efw files will be customized to produce a particular output format. For example, to create a documentation emitter class called "DocEmitter" whose related files (written in C by default) have a file stem of *doc*, we would execute the following command:

```
newemit DocEmitter doc
```

The result would be files doc.idl, doc.c, emitdoc.c, Makefile, and doc.efw. The remaining steps involve customizing the doc.c and doc.efw files.

Notes: (1) For AIX, the <*filestem*> argument to the **newemit** program should consist of *only* lowercase characters. (2) For Windows, **newemit** is a batch file. When you run **newemit** in the Windows environment, the <*filestem*> argument (that is, the second argument to **newemit**) must be no longer than three characters. This is due to the three-character extension length imposed for file names by DOS.

## Designing the output file

Look at a typical IDL interface definition, and hand-construct the desired output file for that interface. For example, suppose we want our "DocEmitter" to construct a documentation file that simply lists the class name and the return types for the methods it introduces or overrides. Thus, given the following IDL specification,

```
#include <somobj.idl>
interface Animal: SOMObject {
void setSound(in string sound);
void makeSound();
};
```

we would want the output file to look like this:

```
The following methods:
  setSound, of type void
  makeSound of type void
are implemented by class Animal.
```

## Constructing an output template

The next step is to construct an output template, based on the sample output file. Separate the sample output file into different *sections*, based on the aspect of the IDL specification to which they most closely correspond (methods, attributes, and so forth). For example, the first three lines of the output file above correspond to method declarations, and the last line corresponds to the class interface definition as a whole.

Although the first three lines all correspond to method declarations, they should be further divided into the portion that constitutes the *prolog* (to be emitted only once, regardless of how many methods are to be described), the *repeating* portion (which should be emitted once per method), and the portion that constitutes the *epilog* (also emitted only once). The first line in the sample output above constitutes the prolog for the method-describing section, and the second two lines are representative of the repeating method-describing section. (There is no epilog section used in this example, although the last line of the output shown could be made part of the "methods epilog" section, rather than the "class" section.)

Next, assign the section names. By convention, section names end in uppercase "S" (such as, classS, methodsS, baseS, and so on). Each section name is given on a separate line, preceded by a colon and followed by the text lines that make up the section.

The most appropriate section names for the "DocEmitter" output template are "methodsPrologS," "methodsS," and "classS."

Then, generalize the text lines within each section into a generic template. That is, replace strings that are specific to a particular interface definition with symbols that represent the syntactic unit of the interface definition from which the string was taken. For example, string "Animal" in the current example can be replaced by the standard symbol <className>.

In sum, the output file shown above could be generalized to the following output template:

```
:methodsPrologS
The following methods:
:methodsS
  <methodName>, of type <methodType>
:classS
are implemented by class <className>.
```

Output templates are typically stored in files having a .efw extension. The **newemit** program creates a generic template file, <*filestem*>.efw. It contains all the standard sections, and each

section contains sample template text that exercises all the standard symbols available within that section. This generic template can be edited to contain only those sections needed by the new emitter, with appropriate text. For the current example, file doc.efw would be edited to contain the generalized output template shown above.

Note: Each of the entry classes (discussed earlier) defines a set of standard symbols based on the kind of entry they represent. These symbols are discussed in the later topic “Standard Symbols.” Its subtopic “The section-name symbols” lists all standard section names, along with the method that emits the section having that section name. New symbols (and new section names) can also be defined, if needed, as described in the topic “Defining new symbols.”

## Customizing emitter control flow

The **newemit** program creates a subclass of **SOMTEmitC** (in this example, “DocEmitter”) that overrides the **somtGenerateSections** method. The **newemit** program also provides a default implementation of **somtGenerateSections** for “DocEmitter” in the doc.c file. This implementation should be customized.

The **somtGenerateSections** method determines which sections of the output template are emitted and in what order. (The output template only specifies which sections are available to the emitter and their contents; it does not control which sections are actually emitted or their order.)

For the current example, we want our emitter to first emit the “methodsPrologS” section of the output template, then the “methodsS” section, once for each method introduced by the class, followed by the “classS” section. However, the default implementation of **somtGenerateSections,** provided by **newemit**, emits the “classS” section first; thus, we must switch the order in which the sections are emitted.

The default implementation of **somtGenerateSections** also emits other sections; however, because those sections are not defined in our example output template, those portions of code should be removed.

The crucial portions of DocEmitter’s implementation of **somtGenerateSections** (in doc.c, and after the class and methods section order has been switched) are shown below:

```
SOM_Scope boolean SOMLINK somtGenerateSections(DocEmitC somSelf)
{
  /* Define symbols available in all sections of the output
   * template.
   */
  _somtFileSymbols(somSelf);

  if (cls != (SOMTClassEntryC *) NULL) {
  /* Emit the "methods" section for each method of the class.
   * If a "methodsProlog" section is defined, it will precede the
   * first method; and if a "methodsEpilog" section is defined, it
   * will follow the last method.
   */
     _somtScanMethods(somSelf, "somtImplemented",
            "somtEmitMethodsProlog", "somtEmitMethod",
            "somtEmitMethodsEpilog", 0);
     _somtEmitClass(somSelf);    /* emit the "class" section */
  }

  return (TRUE);
}
```

Notice the use of the **somtScanMethods** method to iterate through the class’s methods and emit the “methodsS” section for each method. **SOMTEmitC** also defines “scanning” methods for data items, base classes, passthrus, attributes, constants, typedefs, struct, enums, union, interfaces, and nested modules.

## Compiling and running the new emitter

Compile the driver program provided by **newemit** and the implementation of your emitter together to create a dynamically linked library (a DLL) for a new emitter. The **newemit** program provides a Makefile to perform this step (simply enter “make”).

The new emitter can now be invoked by the SOM Compiler via the **–s** option (which overrides the SMEMIT variable, for the current **sc** command, with the specified emitter). For example, “DocEmitter”, packaged in emitdoc.dll, can be invoked by running the SOM Compiler with the “–sdoc” option. (Note that the value of the **–s** option is the file stem specified earlier to **newemit**.) Invoking the following **sc** command will produce an animal.doc file just like the one shown at the beginning of this section:

```
sc –sdoc animal.idl
```

### *Compiling and running under Windows*

The SOM Compiler under Windows is actually a DOS program. As such, it does not have the ability to dynamically load DLLs for new emitters; rather, you must re-link the SOM Compiler so that it includes the user-defined emitter. To compile and re-link a new emitter requires the Symantec C/C++ Compiler, Version 6.0 or greater. The SOM Compiler is built using the 32-bit DOS option of the Symantec Compiler, so please ensure that this is installed. For more details on the 32-bit DOS mode, please consult Symantec’s documentation. These instructions assume that your PATH, INCLUDE, and LIB environment variables are set up to use the Symantec C/C++ Compiler **sc**.

Running the **newemit** batch file generates a Makefile that you should use to compile the new emitter and to link in the necessary SOM Compiler libraries, so that a new version of the SOM Pre-Compiler (**somipc.exe**) can be built.

To create a new template emitter (for example, DocEmitter, with extension .doc), you should type:

```
C:> newemit DocEmitter doc
    doc.idl:
    doc.c:
    emitdoc.c:
    Makefile:
    doc.efw:
    useremit.c:
```

Note: If you want to generate a C++ emitter, then you should type:

```
C:> newemit –C++ DocEmitter doc
    doc.idl:
    doc.cpp:
    emitdoc.cpp:
    Makefile:
    doc.efw:
    useremit.cpp:
```

To link the doc emitter into the SOM Compiler type:

```
C:> make
    sc –I. –c –mx –D__SOMEMIT__ emitdoc.c
    sccx –I. –mx –D__SOMEMIT__ emitdoc.c

    sc –I. –c –mx –D__SOMEMIT__ doc.c
    sccx –I. –mx –D__SOMEMIT__ doc.c

    sc –I. –c –mx –D__SOMEMIT__ useremit.c
    sccx –I. –mx –D__SOMEMIT__ useremit.c
```

```
lib C:\u\otp\lib\scuser.lib -+emitdoc-+doc;
...
Warning: 'emitdoc.obj' not found, can't delete or extract it
Warning: 'doc.obj' not found, can't delete or extract it
...
link386 cx+useremit,somipc,,sc1+sc2+scuser,/NOI;
Output is a DOS EXE file.
```

Note: The Warnings for the .obj files only occur the *first* time the new emitter is compiled.

To test the doc emitter, type (in the *same* directory where you built it):

```
C:> sc -sdoc *.idl
```

Once you have tested the new emitter, you can install it as a permanent emitter in the %SOMBASE%\bin directory by typing:

```
C:> make install
    copy C:\SOM\bin\somipc.exe C:\SOM\bin\somipc.old
           1 file(s) copied
    del C:\SOM\bin\somipc.exe
    copy somipc.exe C:\SOM\bin
           1 file(s) copied
    copy doc.efw C:\SOM\include
           1 file(s) copied
```

If you subsequently decide you want to remove the new emitter, type (in the *same* directory where you built it):

```
C:> make uninstal
           1 file(s) copied
```

This uninstall process will only revert the emitter to the previous version of the SOM Compiler. If you want to use the version that was shipped, then you need to type the following command (assuming that your %SOMBASE% is set to C:\SOM):

```
C:> copy C:\SOM\bin\somipc.org C:\SOM\bin\somipc.exe
```

# 1.5 Writing an Emitter — Advanced Topics

## Defining new symbols

The Emitter Framework defines a number of symbols that can be used in output templates. (These are described in the section “Standard Symbols.”) Programmers can also define additional symbols as needed. This is usually done within an overriding implementation of the **somtGenerateSections** method or some other method of a user’s subclass of **SOMTEmitC**.

Symbol names defined by the Emitter Framework have been chosen to maintain the readability of the template file. There is very little cost associated with the length of a symbol name, nor is there any practical limit on the length of a symbol name. To maintain the readability of template files, it is suggested that emitter writers follow the pattern of the standard symbol names. Note: Symbol names may not consist of double-byte characters.

The value of a defined symbol can be obtained from a template object using the **somtGetSymbol** method. For example, in a C program,

```
_somtGetSymbol(t, ”className”);
```

returns the value of the “className” symbol, assuming that *t* points to a template object for an emitter. (The **somtTemplate** attribute of an emitter refers to its template object.) The **somtCheckSymbol** method can be used to determine whether or not a given symbol is defined.

New symbols are defined using one of the following methods, invoked on a template object:
**somtSetSymbol,**
**somtSetSymbolCopyValue,**
**somtSetSymbolCopyName,** and
**somtSetSymbolCopyBoth**.

Arguments for each method are the name of the symbol and its value (both as strings). The differences among the four symbol-setting methods are whether they make a copy of the name/value to store in the symbol table, or whether the passed strings are stored. If no copy is made, then the string must not be subsequently freed by the calling program. The **somtSetSymbolCopyValue** method is useful for redefining a symbol that already has a value in the symbol table. The **somtSetSymbolCopyName** method is useful when passing a string value that has been allocated and will not be freed. The **somtSetSymbol** method is useful when both situations co-occur. Typically, however, the **somtSetSymbolCopyBoth** method is used. For example, to set the value of the “NewSym” symbol to value “Hello!”, use the following C method call:

```
_somtSetSymbolCopyBoth(t, ”newSym”, ”Hello!”);
```

where *t* is the template object for an emitter. (The **somtTemplate** attribute of an emitter refers to its template object.)

Another method that can be used to define symbols is **somtExpandSymbol**. This method can be used to set a symbol to a value specified within the output template. Given a symbol representing the name of a section in the output template, the **somtExpandSymbol** method expands that section into a buffer by substituting symbol values for symbol names in the template. The result can then be assigned as the value of a symbol, using one of the symbol-setting methods above. In this way, the values of emitter symbols can be defined declaratively in the template file, rather than procedurally within the emitter’s code. For example, if the template (.efw) file for an emitter contains the following section definition:

```
:methodPrefixS
<functionprefix>_
```

then the following C code within the implementation of an emitter's method will set symbol "methodPrefix" to be the expansion of the "methodPrefixS" section in the template file (that is, the value of symbol "functionprefix," if defined by the emitter, followed by an underscore).

```
SOMTTemplateOutputC t = __get_somtTemplate(somSelf);
char buf[MAX_BUF_SIZE];
...
_somtSetSymbolCopyBoth(t, "methodPrefix",
                    _somtExpandSymbol(t, "methodPrefixS", buf));
```

In addition to defining new symbols within a programmer's subclass of **SOMTEmitC**, new symbols can also be defined within user-defined subclasses of an entry class (**SOMTClassEntryC, SOMTMethodEntryC**, and so forth). In this way, the new symbols will be defined at the same time the *standard* symbols are defined (when the **somtSetSymbolsOnEntry** method is invoked on an entry object). This technique is helpful when the symbols will be useful to multiple emitters. (Each of the emitters can use the new entry class rather than defining the symbols.)

To define new symbols within a user-defined subclass of an entry class, override the **somtSetSymbolsOnEntry** method. For example, to define some new symbols to be used in the "classS" section, and to have these symbols automatically defined for every class entry (rather than requiring every emitter to define them), create a subclass of **SOMTClassEntryC** that overrides the **somtSetSymbolsOnEntry** method. Within the overriding method, invoke the parent method, then use one of the symbol-setting methods to define the new symbol(s).

For instance, a user-defined subclass "SMPClassEntryC" of **SOMTClassEntryC** might override **somtSetSymbolsOnEntry** as follows:

```
SOM_Scope long SOMLINK somtSetSymbolsOnEntry(
                            SMPClassEntryC somSelf,
                            SOMTEmitC emitter, string prefix)
{
    SOMTTemplateOutputC t = __get_somtTemplate(emitter);
    long status;
    status = parent_SOMTClassEntryC_somtSetSymbolsOnEntry
                                (somSelf, emitter, prefix);
    _somtSetSymbolCopyBoth(t, _somtNewSymbol(prefix,"ExtPrefix"),
        _somtGetModifierValue (somSelf, "externalprefix");
    return(status);
}
```

Notice that the parent method is invoked first to define the standard symbols, then a new symbol is defined whose value is the "externalprefix" modifier for the class.

The *prefix* parameter of the **somtSetSymbolsOnEntry** method is prefixed to each of the standard symbol names that the method defines. The prefix is set by the emitter framework (when it invokes **somtSetSymbolsOnEntry** on an entry) to match the role of that entry in the class interface definition. For example, the standard symbols defined by a class entry that corresponds to the target class of the emitter will be prefixed with "class", the standard symbols set by the metaclass entry of the target class will be prefixed with "meta", and so on. The **somtNewSymbol** function is provided for users to create new symbols from the prefix passed to **somtSetSymbolsOnEntry** in overriding implementations of **somtSetSymbolsOnEntry** (as in the above example)**.**

It is important that overriding implementations of **somtSetSymbolsOnEntry** in subclasses of **SOMTClassEntryC** in particular use **somtNewSymbol** and the *prefix* parameter to define new symbols (rather than defining new symbols with a fixed name, such as "classExtPrefix") because that method will be invoked not only to define symbols for the target class (for which "prefix" will be "class"), but also to define symbols for its base class(es) and metaclass (for which "prefix" will be "base" or "meta").

When subclassing one of the entry classes, it is necessary to use *shadowing* to have the object graph builder use the new subclass when constructing the object graph, rather than the original entry class. Otherwise, subclassing the entry class will have no effect. See the later topic entitled “Shadowing.”

## Customizing section-emitting methods

The **somtGenerateSections** method of **SOMTEmitC** invokes section-emitting methods, such as **somtEmitClass** and **somtEmitMethod**. To specialize the behavior of one of these methods, we could override them. This would allow us, for instance, to set symbols differently before emitting the “methodsS” section, depending on the characteristics of the method.

An emitter (a subclass of **SOMTEmitC**) can also define *new* section-emitting methods. For example, an emitter could introduce a new section-emitting method, “somtEmitMethod2”. The new section-emitting method can be passed by **somtGenerateSections** as an argument to **somtScanMethods** (instead of passing **somtEmitMethod**), so that for each of the target class’s methods, “somtEmitMethod2” is invoked (instead of **somtEmitMethod**). User-defined sections can also be emitted by changing the value of one of the predefined section-name symbols, as described under the next topic, “Changing section names.”

Section-emitting methods take as an argument the entry object about which information is to be emitted. For example, an argument to **somtEmitMethod** is a method entry object (an instance of **SOMTMethodEntryC**). Each such entry object supports methods for obtaining information about the portion of the IDL interface specification it represents. (For example, a method entry object has an attribute, **somtArgCount**, that gives the number of parameters the method has, and every entry also supports the **somtGetModifierList** and **somtGetModifierValue** methods for obtaining specific information about SOM IDL modifiers.) This information can be used to guide the behavior of the section-emitting methods.

User-defined implementations of section-emitting methods typically define new symbols as needed, as described above, and then invoke the **somtOutputSection** method to produce output from the appropriate template section.

## Changing section names

Each predefined section-emitting method in the Emitter Framework determines which section of the output template to emit based on the value of a predefined section-name symbol. For example, the **somtEmitProlog** method emits the section whose name is specified by the “prologSN” symbol. (“SN stands for “section name.”) The default value of the “prologSN” symbol is “prologS.” Thus, **somtEmitProlog** emits the “prologS” section of the output template by default. The table at the end of the next section lists the default values of all predefined section-name symbols and indicates which section-emitting methods use them.

To change the section that a section-emitting method emits, simply change the value of the appropriate section-name symbol. For example, to have **somtEmitProlog** emit the section “myPrologS” instead of the section “prologS”, invoke the **somtSetSymbol** method as follows from within the **somtGenerateSections** method of your emitter, prior to invoking **somtEmitProlog**:

```
_somtSetSymbolCopyValue(t, ”prologSN”, ”myPrologS”);
```

This technique allows an emitter to use the same section-emitting method to emit multiple sections of the output file. For example, we could have both a “prologS” section and a “myPrologS” section in the output template. The first time **somtGenerateSections** invokes **somtEmitProlog**, it will (by default) emit the “prologS” section. Prior to invoking **somtEmitProlog** a second time, the emitter changes the value of the “prologSN” symbol, as above, to “myPrologS”. Thus, the second time the emitter invokes **somtEmitProlog**, it will emit the “myPrologS” section.

Note: User-defined section names may not contain double-byte characters.

## Shadowing

Some emitters may require subclassing one or more of the entry classes (**SOMTClassEntryC, SOMTMethodEntryC**, etc.) to add new methods or override existing methods. For example, changing the behavior of the **somtGetNextParameter** method would require subclassing the **SOMTMethodEntryC** class. As another example, if an emitter needs symbols that are not predefined by the Emitter Framework, and these symbols would be useful to multiple emitters, then, rather than defining these symbols in **somtGenerateSections** for every emitter, it may be advantageous to subclass one or more of the entry classes and to override the **somtSetSymbolsOnEntry** method in that subclass.

When an emitter subclasses one or more of the entry classes, the driver program that instantiates the emitter must be modified to use *shadowing*. Shadowing instructs the object graph builder to create instances of the new subclass as it builds the object graph to represent the input (rather than creating instances of the original entry class).

Shadowing allows an emitter to substitute a subclass of an entry class for the parent class without having to recompile the library routines that use the original class. The library routines will automatically pick up all of the changes made in the new subclass when shadowing is used.

Shadowing is accomplished using the **SOM_SubstituteClass** macro. For each user-defined subclass of an entry class, modify the **emit** function in the driver program (stored in emit<*filestem*>.c) to include the following instruction, just after the call to **somtopenEmitFile**:

```
SOM_SubstituteClass(<existing entry class name>,
                                <new subclass name>);
```

For example, to shadow entry class **SOMTClassEntryC** with user-defined subclass "SMPClassEntryC"**,** add the following instruction to the **emit** function, just after the call to **somtopenEmitFile**:

```
SOM_SubstituteClass(SOMTClassEntryC, SMPClassEntryC);
```

When shadowing an entry class, the header file for the class being shadowed must be included in the driver program. For example, shadowing **SOMTClassEntryC** would require adding the directive

```
#include <scclass.h>
```

in the driver program (contained in emit<*filestem*>.c).

## Handling modules

When an emitter is run on a .idl file that contains a module, rather than a single class, the *cls* argument to the **emit** function in the emitter's driver program will be a structure such that (`cls->type == SOMTModuleE`), rather than (`cls->type == SOMTCLassE`). The default implementation of the driver program, provided by **newemit**, creates an emitter having a *target module*, rather than a *target class*, then invokes **somtGenerateSections** on that emitter as usual. The default implementation of **somtGenerateSections** method, in turn, invokes different section-emitting methods depending on whether the emitter has a target module or a target class.

When an emitter is invoked on a module, the emitter should emit only the information pertaining to the module as a whole and any typedef and constant definitions within it. Information pertaining to each of the *interface* specifications contained in the module will be emitted subsequently, on a separate invocation of the emitter.

In other words, when the SOM Compiler processes a .idl file containing a module that includes multiple interface statements, it first runs the requested emitter(s), passing a structure repre-

senting the module. It then runs the same emitter(s) again, passing a structure representing the first interface in the module. It then runs the same emitter(s) again, passing a structure representing the next interface in the module, and so on.

All output goes to the same output file, even though the output is produced by multiple invocations of the emitter. (This is controlled by a global variable, set by the SOM Compiler, that indicates to the **somtopenEmitFile** function whether the output file should be opened for writing or for appending. Thus, the first time the emitter is invoked on a particular input file, a new output file is created, but subsequent invocations of the emitter on the same input file simply append to the same output file.)

Because an emitter that is handling a module has no target class, users should avoid invoking any method of **SOMTEmitC** that requires a target class if the emitter is handling a module. This includes **somtEmitMetaInclude, somtEmitMeta, somtScanBases, somtScanMethods**, and so forth.

## Error Handling

The Emitter Framework provides a set of functions to facilitate error handling within user-written emitters. The following functions can be used to issue informational or error messages of differing levels of severity: **somtmsg, somtwarn, somterror, somtfatal,** and **somtinternal**. These functions optionally take the file name and line number where the error occurred and a format string and arguments to be passed to the **printf** C library function. The functions increment the relevant error count and print a message that contains the file name and line number (if specified), an indication of the severity of the message, and the message itself. In addition, the **somtfatal** and **somtinternal** functions remove the output file being constructed and terminate the process. Below is an example of producing an error message using the **somterror** function (*entry* is an instance of one of the Entry classes, and *cls* is the emitter's target class):

```
somterror(__get_somtSourceFileName(cls),
          __get_somtSourceLineNumber(entry),
          "I don't understand the entry named %s.\n",
          __get_somtEntryName(entry));
```

When the **somtfatal** or **somtinternal** function is invoked, the output file being constructed (the one opened using the **somtopenEmitFile** function) is removed and the process is terminated. These actions are also taken if the SOM Compiler detects an internal error within the emitter or if a user-generated interrupt occurs. It may be necessary to prevent these signals from being detected in certain sections of an emitter's code. The Emitter Framework provides two functions, **somtunsetEmitSignals** and **somtresetEmitSignals**, to protect such critical portions of emitter code. These functions take no arguments and return no value. An example is shown below of using these functions to protect a portion of code from signal processing:

```
somtunsetEmitSignals();
/* do some protected processing */
somtresetEmitSignals();
```

See the Reference portion of this book for more information on the error-handling functions provided by the Emitter Framework.

# 1.6 Standard Symbols

The following lists of standard symbols are organized in two ways:

- The first category groups the symbols by what sections of the output template may reference them. These are the lists to reference when writing an output template.
- The second category groups the symbols by which entry class defines them. These are the lists to reference when changing the value of a predefined symbol through shadowing (or when defining an additional, related symbol), as this list indicates which entry class to subclass.

Each symbol is described in more detail in the second part of this section.

## 1. Symbols by section validity

Valid in **all output template sections**, when an emitter has a target class, and in the **interfaceS** section when an emitter has a target module**:**

```
className
classIDLScopedName
classCScopedName
classComment
classInclude
classLineNumber
classMods
classMajorVersion
classMinorVersion
classSourceFile
classSourceFileStem
classReleaseOrder
timeStamp  (the date and time the emitter was run)
```

If a metaclass is explicitly defined for the class, the following symbols are also defined:

```
metaName
metaIDLScopedName
metaCScopedName
metaComment
metaInclude
metaLineNumber
metaMajorVersion
metaMinorVersion
metaMods
metaReleaseOrder
metaSourceFile
metaSourceFileStem
```

Valid within the **baseIncludesS** and **baseS** sections:

```
baseName
baseIDLScopedName
baseCScopedName
baseComment
baseInclude
baseLineNumber
baseMajorVersion
baseMinorVersion
baseMods
baseReleaseOrder
baseSourceFile
baseSourceFileStem
```

Valid in the **methodsS, overrideMethodsS,** and **inheritedMethodsS** sections**:**

```
methodName
methodIDLScopedName
methodCScopedName
methodComment
methodLineNumber
methodMods
methodType
methodCReturnType
methodContext
methodRaises
methodClassName
methodCParamList
methodCParamListVA
methodIDLParamList
methodShortParamNameList
methodFullParamNameList
```

Valid in the **dataS** section:

```
dataName
dataIDLScopedName
dataCScopedName
dataComment
dataLineNumber
dataMods
dataType
dataArrayDimensions
dataPointers
```

Valid in the **passthruS** section:

```
passthruName
passthruComment
passthruLineNumber
passthruMods
passthruLanguage
passthruTarget
passthruBody
```

Valid in the **constantS** section:

```
constantName
constantIDLScopedName
constantCScopedName
constantComment
constantLineNumber
constantMods
constantType
constantValueUnevaluated
constantValueEvaluated
```

Valid in the **typedefS** section:

```
typedefDeclarators
typedefBaseType
typedefComment
typedefLineNumber
typedefMods
```

Valid in the **structS** section:

```
structName
structIDLScopedName
structCScopedName
structcomment
structLineNumber
structMods
```

Valid in the **unionS** section:

```
unionName
unionIDLScopedName
unionCScopedName
unionComment
unionLineNumber
unionMods
```

Valid in the **enumS** section:

```
enumName
enumIDLScopedName
enumCScopedName
enumComment
enumLineNumber
enumMods
enumNames
```

Valid in the **attributeS** section:

```
attributeDeclarators
attributeBaseType
attributeComment
attributeLineNumber
attributeMods
```

Valid in the **moduleS** section:

```
moduleName
moduleIDLScopedName
moduleCScopedName
moduleComment
moduleLineNumber
moduleMods
```

## 2. Symbols by entry class availability

The following symbols are established and defined for each object of the indicated entry class when the **somtSetSymbolsOnEntry** method is invoked on that object. (The **somtSetSymbolsOnEntry** method can be invoked on an entry object directly. It is also invoked automatically on the target class entry object and on its metaclass entry object by the **somtFileSymbols** method. It is also invoked automatically on each entry processed by one of the section-emitter or scanning methods of **SOMTEmitC**.)

### For SOMTEntryC

<*prefix*>Name — The unscoped name of the entry.

<*prefix*>IDLScopedName — The scoped name of the entry, using "::" as delimiters.

<*prefix*>CScopedName — The scoped name of the entry, using "_" as delimiters.

<*prefix*>Comment — The comment that follows the entry in the IDL specification.

<*prefix*>LineNumber — The line number where the IDL specification of the entry *ends*.

<*prefix*>Mods — The SOM IDL modifiers of the entry.

where <*prefix*> is replaced by the corresponding IDL syntactic unit being defined, either "module," "attribute", "constant," "typedef," "struct," "enum," "union," "class", "base", "meta", "method", "data", "passthru", or a user-specified prefix.

### For SOMTCommonEntryC

<*prefix*>Type — The type of the entry. For methods, the return type.

<*prefix*>ArrayDimensions — The array dimensions of the entry, if it is an array.

<*prefix*>Pointers — The pointer stars for the entry, if it is a pointer type.

where <*prefix*> is replaced by either "method," "data," or a user-specified prefix.

### For SOMTAttributeEntryC

attributeDeclarators — The list of attribute declarators.

attributeBaseType — The base type of the attribute(s), not including pointer stars or array dimensions, if any.

### For SOMTEnumEntryC

enumNames — The list of enumerator names of the enumeration.

### For SOMTClassEntryC

classMajorVersion — The class's major version number.

classMinorVersion — The class's minor version number.

classSourceFile — The name of the IDL source file.

classSourceFileStem — The file stem of the binding files to be produced from the input IDL file. If the input IDL has a "filestem" modifier, then its value defines the symbol. Otherwise, the symbol will be the filestem of the input IDL file.

classReleaseOrder — The release order list for the class.

classInclude — The expression to be used in include statements to access the appropriate file for this class (such as <somobj.idl>).

### For SOMTConstantEntryC

constantType — The type of the constant.

constantValueEvaluated — The evaluated value of the constant. For constants of type string or char, this value includes the quotes.

constantValueUnevaluated — The unevaluated value of the constant. Constants within the value expression are, however, replaced with their values. For constants of type string or char, this value does *not* include the quotes.

### For SOMTMethodEntryC

methodCReturnType — The C/C++ return type of the method.

methodClassName — For an overriding method, the class whose method is overridden. For new methods, the introducing class.

methodIDLParamList — The formal parameter list (including types) for the method, in IDL form (includes only *explicit* parameters).

methodCParamList — The formal parameter list (including types) for the method's procedure, in C/C++ form (including all parameters).

methodCParamListVA — The formal parameter list (including types) for the method's procedure, in C/C++ form (including all parameters), with any va_list parameter replaced by "...".

methodShortParamNameList — A list consisting of the names of the method's explicit parameters (excluding *somSelf, ev,* and *ctx*).

methodFullParamNameList — A list consisting of the names of all of the method's procedure's formal parameters (including implicit method parameters and *somSelf)*.

methodRaises — A list of the exceptions the method may raise.

methodContext — A list of the context string literals for the method.

### For SOMTParameterEntryC

parameterDirection — Whether the parameter is an in, out, or inout parameter.

parameterIDLDeclaration — The declaration of the parameter, including type, in IDL form.

parameterCDeclaration — The declaration of the parameter, including type, in C/C++ form. This may differ from the IDL declaration, particularly when the parameter is an out or inout parameter.

### For SOMTPassthruEntryC

passthruLanguage — The target language of the passthru, in upper case (for example, "C").

passthruTarget — The file type for this passthru; for example, "h", "ih"

passthruBody — The full contents of the passthru entry, including newlines.

### For SOMTSequenceEntryC

sequenceLength — The maximum length of the sequence, as declared in IDL, or zero if unspecified.

### For SOMTStringEntryC

stringLength — The maximum length of the string, as declared in IDL, or zero if unspecified.

### For SOMTTypedefEntryC

typedefDeclarators — The list of declarators.

typedefBaseType — The base type of the new user-defined type(s), not including pointer stars or array dimensions, if any.

## The section-name symbols

The Emitter Framework recognizes a set of special symbols known as "section-name symbols," which correspond to the various sections that can be emitted from an output template. The value of each section-name symbol is the name of a section to be emitted.

Each predefined section-emitting method in the Emitter Framework determines which section of the output template to emit based on the value of a predefined section-name symbol. For example, the **somtEmitProlog** method emits the section whose name is specified by the "prologSN" symbol. ("SN means "section name.") The default value of the "prologSN" symbol is "prologS." Thus, **somtEmitProlog** emits the "prologS" section of the output template by default.

The value of a section-name symbol can be changed to cause the corresponding section-emitting method to emit a section of a different name. For example, to have the **somtEmitProlog** method emit a section named "myPrologS" rather than "prologS," set the value of the "prologSN" symbol to "myPrologS," using the **somtSetSymbolCopyValue** method as described in the earlier section "Defining new symbols," before invoking **somtEmitProlog.**

The following table lists all symbol names, their initial value, and the method that uses them.

| Symbol Name | Initial Value (Section Name) | Used by Method |
|---|---|---|
| attributeSN | attributeS | somtEmitAttribute |
| attributeEpilogSN | attributeEpilogS | somtEmitAttributeEpilog |
| attributePrologSN | attributePrologS | somtEmitAttributeProlog |
| baseSN | baseS | somtEmitBase |
| baseEpilogSN | baseEpilogS | somtEmitBaseEpilog |
| baseIncludesSN | baseIncludesS | somtEmitBaseIncludes |
| baseIncludesEpilogSN | baseIncludesEpilogS | somtEmitBaseIncludesEpilog |
| baseIncludesPrologSN | baseIncludesPrologS | somtEmitBaseIncludesProlog |
| basePrologSN | basePrologS | somtEmitBaseProlog |
| classSN | classS | somtEmitClass |
| constantSN | constantS | somtEmitConstant |
| constantPrologSN | constantPrologS | somtEmitConstantProlog |
| constantEpilogSN | constantEpilogS | somtEmitConstantEpilog |
| dataSN | dataS | somtEmitData |
| dataEpilogSN | dataEpilogS | somtEmitDataEpilog |
| dataPrologSN | dataPrologS | somtEmitDataProlog |
| enumSN | enumS | somtEmitEnum |
| enumEpilogSN | enumEpilogS | somtEmitEnumEpilog |
| enumPrologSN | enumPrologS | somtEmitEnumProlog |
| epilogSN | epilogS | somtEmitEpilog |
| inheritedMethodsSN | inheritedMethodsS | somtEmitMethod |
| interfaceSN | interfaceS | somtEmitInterface |
| interfaceEpilogSN | interfaceEpilogS | somtEmitInterfaceEpilog |
| interfacePrologSN | interfacePrologS | somtEmitInterfaceProlog |
| metaSN | metaS | somtEmitMeta |
| metaIncludeSN | metaIncludeS | somtEmitMetaIncludes |
| methodsSN | methodsS | somtEmitMethod |
| methodsSN | methodsS | somtEmitMethods |
| methodsEpilogSN | methodsEpilogS | somtEmitMethodsEpilog |
| methodsPrologSN | methodsPrologS | somtEmitMethodsProlog |
| moduleSN | moduleS | somtEmitModule |
| moduleEpilogSN | moduleEpilogS | somtEmitModuleEpilog |
| modulePrologSN | modulePrologS | somtEmitModuleProlog |
| overrideMethodsSN | overrideMethodsS | somtEmitMethod |
| passthruSN | passthruS | somtEmitPassthru |
| passthruEpilogSN | passthruEpilogS | somtEmitPassthruEpilog |
| passthruPrologSN | passthruPrologS | somtEmitPassthruProlog |
| prologSN | prologS | somtEmitProlog |
| releaseSN | releaseS | somtEmitRelease |
| structSN | structS | somtEmitStruct |
| structEpilogSN | structEpilogS | somtEmitStructEpilog |
| structPrologSN | structPrologS | somtEmitStructProlog |
| typedefSN | typedefS | somtEmitTypedef |
| typedefEpilogSN | typedefEpilogS | somtEmitTypedefEpilog |
| typedefPrologSN | typedefPrologS | somtEmitTypedefProlog |
| unionEpilogSN | unionEpilogS | somtEmitUnionEpilog |
| unionSN | unionS | somtEmitUnion |
| unionPrologsN | unionPrologs | somtEmitUnionProlog |

## 1.7 Limitations

The Emitter Framework is intended to be used to develop emitters to be run on .idl input files, rather than on .csc input files containing OIDL interface definitions. (OIDL is the language previously supported by SOM for describing interfaces.) Many of the standard symbols set by the Emitter Framework for syntactic entities in an IDL specification are not set for the corresponding entities in an OIDL specification. For example, the **somtTypeObj** and **somtPtrs** attributes of **SOMTCommonEntryC** objects may be NULL for methods, parameters, and data items when an emitter processes a .csc input file. Likewise, the **somtGetFirstArrayDimension** and **somtGetNextArrayDimension** methods may not be reliable when processing .csc input files.