

OMG IDL Style Guide

ab/98-06-03

June 17, 1998

This document has been approved by the Architecture Board as a style guide for the IDL contained in technology submissions to the OMG.

It is an AB requirement that submissions adhere to the mandatory parts of this style guide.

Copyright (c) 1998, OMG.

CORBA, CORBA services, CORBA facilities, and Object Request Broker are trademarks of Object Management Group.

Prefaceiii
0.1 IDL Style Guide Purpose	iii
0.2 Conformance	iii
0.3 Relationship to Other OMG Documents	iv
0.4 An Example of Style: CORBAservices IDL	iv
0.5 Typography Conventions	v
 1 Modules and Files	 1
1.1 Use of Modules	1
1.2 Submission of IDL Files.....	2
1.3 File Names	2
1.4 Include Conventions.....	3
1.5 File Structure	3
1.5.1 File Internal Identification	4
1.5.2 Guard	4
1.5.3 Required Contents	5
1.5.4 Complete File Example	5
 2 Identifiers	 7
2.1 Identifiers	7
2.1.1 Mixed Case, Beginning Upper, No Underscores	7
2.1.2 Lower Case with Underscores	8
2.1.3 Upper Case with Underscores	8
2.2 Global References	8
 3 Formatting.....	 9

Preface

0.1 IDL Style Guide Purpose

Most submissions to the OMG contain IDL. Being written by different authors, submissions use different styles in their IDL. Having some level of consistent style eases the processes of specifying IDL in submissions and of using that IDL in programs. Two aspects of consistent style are addressed in this guide. The first and most important consideration is **portability**: how IDL files should be named and how inclusion of other files should be done. The second consideration is **naming conventions**, less important for portability, but very important for readability and consistency.

This guide presents style and usage conventions in several areas, in order of importance:

- File Structure
- Identifier naming
- Global references
- Formatting (appearance)

All future OMG-approved specification are expected to adhere to the first three of these. The fourth will aid readability.

The level of “style” suggested is much lower than that required for many frameworks and patterns and is not intended to supercede or sidetrack those efforts.

0.2 Conformance

This document contains both *requirements* and *recommendations*.

Requirements are identified by the noun “requirement” or the verb “shall”. No deviations from requirements are permitted.

Recommendations are identified by the noun “recommendation”, the clause “if possible”, or the word “should”. Following recommendations will add to the consistency of OMG IDL and ease of use for OMG specifications. However, a specification may deviate from a recommendation for a good reason. Submitters need not justify their decisions concerning recommendations.

Reason – A paragraph identified in this format (that is, identified as Reason, set off from other paragraphs by horizontal rules) contains the motivation for a requirement or recommendation.

0.3 Relationship to Other OMG Documents

Parts of this style guide are taken from approved OMG specifications. Parts are taken from the general requirements for an RFP response that were in effect at the time that this guide was written. It is the intent to keep all three (specifications, RFP template, Style Guide) mutually consistent. Conflicts and interpretations shall be referred to the OMG Architecture Board, which is the owner of this document and of the general RFP template and requirements. The AB may update its own documents directly and may recommend that adopted specifications be changed by the appropriate Technical Committee.

0.4 An Example of Style: CORBAServices IDL

In addition to this guide, there is a large example of IDL that is structured according to this style. This concrete source code and file structuring may be easier to understand than an abstract guide. The example IDL is the CORBAServices IDL (as of March, 1998) modified to match this guide. The files cited below will continue to be valid as a style guide, even if the CORBAServices IDL changes.

This example IDL is provided in two separate documents, in the formats recommended to accompany a submission

1. **ab/98-04-02.** A text (ASCII) file containing the concatenation of all the module files.
2. **ab/98-04-03.** A structured, compressed file of files, containing the same text as the text file. The important feature of this file is not the particular compression used but the structuring that allows a decompressor program to reconstitute the separate IDL modules as separate files.

In the example style, the CORBAServices IDL identifiers (names) are not changed from the original services documents because to do so would cause incompatibilities with existing implementations at the programming level. *That is, non-conforming identifiers have been ‘grandfathered’ for existing services.*

File names (implicit or explicit) were changed in some cases to match the style guide conventions. Adopting such a file naming convention doesn’t have to cause programming incompatibility. Differences in file names can be taken care of by using the updated file name for new applications and leaving those in current use intact.

(That is, provide two versions of the file with the same contents. While this usually isn't a good practice, the compatibility advantage overrides the disadvantages. The disadvantage might be ameliorated by having the older file contain a single line: an **include** of the new file.) As it is, it is likely that vendors had to take matters into their own hands for file names, so these requirements add to portability.

0.5 Typography Conventions

Typogrography conventions are the same as those in published OMG documents.

- Ordinary text appears in this font.
- **IDL statements appear in this font as do IDL references embedded in the text.**
- **File names and programming language code appear in this font.**

Modules and Files

1

This chapter describes the file structuring for the IDL contained in a specification published by OMG.

1.1 Use of Modules

All declarations of OMG-approved IDL shall be contained in modules. No declarations of interfaces and definitions shall appear in the global scope.

Nesting modules is a useful technique when dealing with large namespaces to avoid name clashes and clarify relationships. A module nested within another module shall not have the same name as a top-level module in any other OMG specification.

Reason – Global definitions of any kind are a common source of error. Accidentally leaving off a scope or misspelling a reference might result in an unintended use of the global definition. Accidents of this type don't occur if all IDL is in a module.

Restricting a nested module name to not have the same name as another OMG top-level module avoids confusion. For example, a nested module by the name of CORBA would be exceedingly confusing. References to such a module from within the containing module, while valid and syntactically unambiguous, are an invitation to error.

Clearly, in an environment less strictly controlled than OMG specifications, enforcing such a rule is untenable. A user cannot be expected to know every top-level module name that might be used in the future. Products from vendors will contain IDL and the names of such modules can't be foreseen. Program generation tools may automatically produce nested modules with little or no control over their names. Confusion is possible. In the case of OMG, however, there is absolute control of the adopted IDL: future OMG specifications will be required to adhere to this rule.

1.2 Submission of IDL Files

If a specification contains IDL, that IDL shall also be available in a file separate from the specification. That is, a specification consists of at least two parts: the text and an IDL file. The IDL file shall be suitable for processing by an IDL compiler, according to the rules below.

In OMG-specified IDL, each module shall occupy a separate file. If a module is defined in multiple parts then each part shall occupy a separate file (see the Note and the file name rules below).

If there is a single module (in a single IDL file) for a specification, that single file shall be provided to the OMG as a separate document, in text (ASCII) form, as part of the submission .

If there are multiple modules in a specification (hence multiple files), they shall not be submitted as separate OMG documents but shall be submitted as a single *IDL distribution file*. The distribution format is a single text file composed of a concatenation of the separate IDL files. The rules below for file structure unambiguously state where file boundaries are to be found in such a concatenated file (at the **//File:** line.).

If possible, a submission should provide a second distribution file using some bundling technique that can automatically unbundle that distribution file into separate files. The widely used **.zip** and **.tar** file formats are examples suitable for this second distribution file.

Note – Use of multiple files for a single module is allowed in order to take advantage of the IDL feature for a module that allows it to be “re-opened”. That is, the second time a compiler sees a module with the same name, it treats the contents as a continuation of the module definition already processed. The reason for such use is to keep logically related interfaces in the same name space (module) while avoiding circular references. A circular reference can occur when two modules need to reference information provided in one of the modules. Without the “re-opening” feature, there would be an infinite recursion of module includes or, if guards are used, undefined interfaces. The re-opening feature allows the relevant portions of the module to be placed in a third file that is included by each of the other files.

1.3 File Names

The name of a file shall be the same as the name of the module contained in that file, followed by the four characters “**.idl**”.

When a module is defined in multiple parts, the first such file shall be named using the rule above. Each subsequent file shall be named by appending a sequence number starting at 2 to the module name, before appending the **.idl**. Thus if module **foo** is in three files, the file names will be **foo.idl**, **foo2.idl**, and **foo3.idl**.

Note – This file naming convention guarantees as much uniqueness among OMG-specified file names as possible since it is a goal that all OMG specifications can be used together. Such use implies that all OMG specification use IDL with unique module names; hence, this rule guarantees unique file names.

For Corba Core, the file name is **orb.idl**. (This is an admitted exception to the rule, for compatibility with previous versions.)

OMG specified services and facilities shall use prefixes for modules and file names as follows.

- For CORBA services, “**Cos**”
- For CORBA facilities, “**Cf**”
- For domain services, “**Ds**”
- For domain facilities “**Df**”

Reason – The prefix convention is intended to avoid collisions with non-OMG module names and files. Except for (a finite number of) previously approved specifications, users can safely avoid collisions with CORBA specifications by not starting modules with the letters above. A submission for an OMG specification is responsible for ensuring the uniqueness within OMG by consulting the OMG adopted specifications.

CORBA services and CORBA facilities, as described in the OMA, provide interfaces that are expected to be useful to a wide range of applications. The line between CORBA services and CORBA facilities is somewhat indistinct, being primarily a judgment of complexity. Domain provided specifications are also at different levels of complexity but are applicable within a narrower range of applications. For both common and domain offerings, the relevant Technical Committee is charged with determining the appropriate category: service or facility. For uncertainties concerning commonality or domain specificity, the Technical Committees should consult with the Architecture Board.

1.4 Include Conventions

OMG IDL shall be available in the “system” directory. Therefore, an included file is specified using the **< >** form of **#include**. For example,

```
#include <CosLifeCycle.idl>
```

1.5 File Structure

A file shall contain an identifying line, a guard, and a **#pragma prefix**, as detailed below

1.5.1 File Internal Identification

The first line of the file shall contain “**//File:**” followed by a single space followed by the name of the file. For example,

```
//File: CosNaming.idl
```

Additional comment lines are recommended to identify further the purpose and source of the IDL contained the file. They can appear before or, preferably, after the guard.

1.5.2 Guard

An IDL file shall use a *guard* to avoid multiple definition errors.

Reason – A file is included more than once if it has a **#include** referencing it from separate IDL files in the same compilation. When such a file is processed by a compiler the second time, the IDL compiler typically reports multiple definition errors because the identifiers declared are defined more than once.

A guard is easier to show than explain. Figure 1-1 shows an example of a guard for a file named **FileName.idl**.

Figure 1-1 File Guard Example

```
#ifndef _FILE_NAME_IDL_  
#define _FILE_NAME_IDL_  
    ...remainder of the IDL  
#endif // _FILE_NAME_IDL_
```

A guard comprises three preprocessor lines that ensure that IDL contents are not processed more than once by a compiler. The *guard name* is formed by uppercasing all characters in the file name, prepending and postpending underscores, adding an underscore before any word that was capitalized in the middle of the file name, and replacing the period of the **.idl** with an underscore. (If part of the module name is an acronym, submitters may choose how that is reflected in the guard name.)

The first line of the guard (starting with **#ifndef**) directs the preprocessor not to process the IDL if it has done so already, that is, if the guard name has already been defined to the preprocessor during this compilation. If the guard name has been defined, the preprocessor skips to the last line of the guard, ignoring the lines (the IDL) inside the guard.

If the guard name has not been defined, the contents of the file inside the guard is processed. The second line of the guard is reached only if the guard name has not been defined. The second line (starting with **#define**) defines the guard name to the preprocessor so that a subsequent **#include** of the file will not process the IDL. The last line of the guard contains an **#endif** to end the guard and, for documentation, a comment containing the guard name (using either the **//** or **/*...*/** form of comments).

1.5.3 Required Contents

If any other files are to be included, the **#include** statements come after the guard.

After any **#include** lines and immediately before the module statement, the following line shall appear:

```
#pragma prefix "omg.org"
```

1.5.4 Complete File Example

```
//File: ExampleNameOrSomething.idl  
#ifndef _EXAMPLE_NAME_OR_SOMETHING_IDL_  
#define _EXAMPLE_NAME_OR_SOMETHING_IDL_  
  
// (optional) This module is part of the ...  
  
#include <CosNaming.idl>  
#include "ThatOtherThing.idl"  
  
#pragma prefix "omg.org"  
module ExampleNameOrSomething  
{  
    //...the creative and beautiful part: the IDL  
}  
#endif // _EXAMPLE_NAME_OR_SOMETHING_IDL_
```


This chapter describes the requirements for forming identifiers and using global references.

2.1 Identifiers

This section describes the formation rules for IDL identifiers. These are rules for capitalization and use of underscores.

Reason – There is no absolute justification for these or any other particular rules. It is advantageous to have rules, however arbitrary, to help users quickly recognize categories of identifiers. Any consistently followed set of rules would suit the purpose. These are the rules that have been in place, explicitly or implicitly, since the earliest days of CORBA, so they are the CORBA rules.

2.1.1 *Mixed Case, Beginning Upper, No Underscores*

The following categories of identifiers follow the *Mixed Case, Beginning Upper, No Underscores* rules. No underscores appear in the name and all words begin with an upper case letter with the remaining letters being lower case.

- Module
- Interface
- Typedef
- Constructed types (struct, union, enum)
- Exception

For example, **CosLifeCycle**, **Big**, **NotSoBigButNotTooSmallEither**, **NameComponent**.

2.1.2 Lower Case with Underscores

The following categories of identifiers follow the *Lower Case with Underscores* rules. All letters are lower case and words (if more than one) are separated with underscores.

- Operation
- Attribute
- Parameter name
- Structure member

For example, **meters**, **a_partridge_in_a_pear_tree**

2.1.3 Upper Case with Underscores

The following categories of identifiers follow *Upper Case with Underscores* rules. All letters are upper case and words have underscores separating them.

- Enum value
- Constant

For example, **RED**, **TAG_SEC_NAME**.

2.2 Global References

References shall not use the “file scope” convention of a leading double colon. For example, reference to an identifier uses just the scoped name:

```
#include <CosNaming.idl>
interface A {
    void B( ::CosNaming::Name a_name); // non-conformant
    void B( CosNaming::Name a_name); // conformant
};
```

Reason – “File scope” is intended for situations in which there is a conflict between an identifier within the same scope as the reference and an identifier in a different scope, one reachable only from file level. For example, consider an **interface foo** at the file level and another **interface foo** within **module oof**. Within **oof**, a reference to **foo** alone is interpreted by IDL rules as **oof::foo**. If the intent is to reference the interface at file level, the syntax would be **::foo**. A similar situation might occur when modules are nested. An identical identifier in a nested module might hide a similar name in a module with disjoint scope. The solution is the same: use file scope to escape to the file level and then qualify down to the appropriate identifier in the target module.

Since this Style Guide’s requirements for OMG-approved specifications do not allow file level interfaces, only the possibility of nested modules arises. OMG approved specifications that include nested modules must follow rules that do not require the use of file scope. Use of the “file scope” notation is thus not necessary in OMG specifications. Its use is disallowed because it is confusing when used unnecessarily. Its use implies a need for explicit scope resolution when none exists.

This chapter describes the recommended appearance (formatting) for the IDL contained in a specification submitted to OMG. Formatting consists of rules for indentation and spacing. Opinions on these matters show little sign of convergence, with strongly held views varying widely. As a result, these are merely recommendations. If you are looking for some rules, consider these.

The first IDL statement should be flush left, containing no leading white space (spaces or tabs).

For constructs using `{` and `with` as delimiters, the `{` can be on the the same line as its beginning keyword. If the `{` appears on the next line it should appear with the same indentation as the keyword. The closing `}` should appear alone on a line at the same level as the keyword. Code within such a construct should be indented one level.

If a single statement is to be continued on another line, it should be indented at least one level from the previous line. It is often useful to indent enough to make parts of one or more statements line up for readability, as in parameters to an operation or identifiers in a **struct**.

A useful indentation amount is four characters (if using a fixed width font, likely in the text file for the IDL in a submission). Each indentation level should be created by a tab character. (Four characters is a convenient indentation in many editors used in programming tools.)

Figure 3-1 is an example of these recommendations.

(Note. Because of the typographic conventions of the document system for this report, merely cutting the IDL from the figure and pasting it into a programming language editor will probably not produce the same alignment as appears below. Specifically, OMG publishing style mandates that IDL appear in a variable-width rather than the fixed-width font of programming editors and tab stops in OMG documents do not start at the left margin. Thus, a pleasing alignment after the first non-blank character is difficult to achieve in both the publication and the IDL text file. Sad but true.)

Figure 3-1 Block and Indentation Conventions

```
module CosFoo {  
    typedef long Big;  
    exception Me { long i_am_short; };  
    interface Bar  
    {  
        void nada( in  Big nothing,  
                   out Big zilch );  
        long adam_and( in short eve)  
                    raises( Me );  
    };  
};
```