Why One Source File Is Better Than Two

Peter Grogono
Department of Computer Science
Concordia University

Markku Sakkinen
Department of Computer Science and Information Systems
University of Jyväskylä

Abstract

Engineering practice requires that details of the representation of a software module should not be accessible to clients of the module. Developers respect this practice by splitting each module into two source files, an interface and an implementation. We call this the "two-file model". We discuss an alternative model, the "one-file model", that requires the developer to write and maintain one source file rather than two. We show that the one-file model provides better support for information hiding than the two-file model and that it has a number of additional advantages.

1 Introduction

Most modern programming languages support the development of a large program as a collection of modules. Some of these languages require the developer to maintain two files for each module: an interface file and an implementation file. In fact, the need to write two source files for each software module is taken for granted by many software developers. Engineering practice requires interfaces and implementations to be separated; writing separate source files for interface and implementation seems to be a natural and obvious way to proceed. However, it is not necessarily the best way. Interface files are read by other developers, who need to know how to use the services provided by the module, and also by the compiler, which needs to know how to generate the code that invokes these services. Since interfaces have two sets of readers with different requirements, they necessarily contain information that is needed by one set but not the other. In particular, they do not support but rather prevent complete information hiding.

We compare two models of software development: a two-file model, which separates interface and implementation files, and a one-file model, based on a single canonical document. We present reasons for preferring the one-file model. In Section 5, we describe a software tool that combines the one-file model and C++.

1.1 Terminology

We assume that the source code for a software product is split up into several parts. The parts are often realized as physical files but can also be stored in a database. We refer to each part as a *module* of the product. Our arguments apply to both modular programming languages, such as Modula-2 and Ada,[1] and to class-based object oriented languages, such as C++ and Java. If the implementation language is object oriented, a module typically consists of one or more classes.

When a module C uses another module S, for instance by invoking a procedure or function implemented by S, we say that C is a *client* of S and that S is a *supplier* of C. A software product is built and maintained by *developers*. Each module is associated with a particular developer, or perhaps a team of developers, that we refer to as its *owner*. After a module has been released, the owner becomes a *maintainer*. Since ownership of a module is often transferred at this stage, the maintainer is not necessarily the original author of the module. For this and other reasons, it is important to maximize the intelligibility of the source code of modules.

We assume that a software module consists of *features* which may be functions, variables, constants, or perhaps other entities. A feature may be an *attribute* (data member in C++) or a *method* (member function in C++). A *declaration* introduces a new feature and may also indicate properties of the feature such as its type and parameters. For some features, notably methods, there must also be a *definition* that provides details needed by the compiler to generate code for the feature. A *public* feature of a module is accessible to clients of the module; a *private* feature is not accessible.

2 The Models

In several current programming languages, including Ada and C++, the source code of a software module consists of two parts: an *interface file* and an *implementation file*. We call this the *two-file model* (although the actual number of source files associated with a module may in some cases be greater than two) and discuss it in Section 2.1. In other current programming languages, including Eiffel, Java, and Oberon, the source code for a module is contained in a single file. We call this the *one-file model* and discuss it in Section 2.2. Section 2.3 includes a short example to illustrate the difference between the approaches.

2.1 The Two-File Model

A developer using the two-file model writes two source files: an interface file and an implementation file. The interface is usually coded first and should be changed only rarely thereafter. It is particularly important to minimize changes to the interface after the module has been released. The implementation is usually coded after the interface has stabilized and changes as often as its owner finds necessary. If the interface was properly designed, changes to the implementation file should not affect the interface of the module.

The two-file model is usually justified as a means of incorporating Parnas's [15] principle of information hiding into software development. The interface file is available to all developers but the implementation file, which contains the "secrets" of the module, is seen only by the owner of the module. This separation facilitates the development of complex software systems,

[1]Ada supports both modular and object-oriented programming styles.

consisting of many modules, by individuals or teams who communicate partly by means of interface files and partly by means of other forms of documentation.

Some languages specify distinct syntax for interfaces and implementations. For example, an Ada interface has the form

```
package WIDGET is
   -- declarations of public features
private
   -- declarations of private features
end WIDGET;
```

and the corresponding implementation has the form

```
package body WIDGET is
   -- declarations and definitions
end WIDGET;
```

Each of these is a *compilation unit*. Ada has other kinds of compilation units that we do not describe here. A file submitted to the compiler must contain one or more complete compilation units. Ada does not provide mechanisms for combining arbitrary chunks of text.

C++ also uses the two-file model. In contrast to Ada, a C++ compiler imposes minimal constraints on the syntactic structure of source files. Most programmers follow standard conventions, distinguishing "header" files from "implementation" files, and using the directive `#include` for the sole purpose of making header text (consisting mainly of declarations) visible in implementation files.

In C++, it is not necessary for the implementation of a class to be a single compilation unit. While a class is being developed and tested, it may be convenient to compile each method by itself. On the other hand, a single file may contain declarations and method definitions for several classes.

2.2 The One-File Model

In the *one-file model*, the owner of a software module is responsible for a single source file, which in this paper we call the *canonical document*. The canonical document contains all of the information about the module. The compiler and other tools on the developer's workbench pass the canonical document through various filters to obtain different versions of the module. The programming environment may materialize these versions into files or generate them on the fly from the canonical document. Whereas the description of the two-file model in Section 2.1 corresponds closely to actual practice, the one-file model, as described here, is somewhat idealized. Languages such as Eiffel, Java, Oberon, and Smalltalk reflect different aspects of the one-file model.

Since the output of a filter is a subset of its input, we refer to the output of a filter as a *projection* of its input. If the output of a filter is intended for a human reader, we call it a *view*; if it is intended for a software tool (usually but not necessarily the compiler), we call it an *interface*. Note that this use of "interface" differs from that of the previous section: in the two-file model, interfaces are read by both developers and software tools.

We can immediately identify two useful projections of a module:

- The *Client View* contains all of the information needed by the developer of a client module to use the module. In particular, it contains the declarations and documentation for each public feature of the module. It does not contain any information about private features or the representation or implementation of the module.
- The *Client Interface* contains all of the information needed by the compiler to compile clients of the module. It contains the declarations of all public features and as much information as the compiler needs about private features.

If the language has inheritance, there are two more obvious projections.

- The *Subclass View* or *Child View* of class `C` contains all of the information needed by the developer of a subclass of class `C`.
- The *Subclass Interface* or *Child Interface* of class `C` contains all of the information needed by the compiler to compile a subclass of class `C`.

There may be other possible projections that depend on specific languages features. For example, in Java, a *Package View* could be provided to reflect package-level visibility.

Documentation is unnecessary in interfaces because they are not intended to be read by people. Furthermore, the information in a interface can be encoded or compressed to save space and processing time. Saving space is especially useful if the interface is to be transmitted over a network.

In Eiffel, the class developer has very fine-grained control over the export of features; a feature can be made visible only to one or more explicitly listed classes and their descendants. This implies that different client classes may need their own views of a particular supplier class, a feature that Eiffel does not currently provide. There is an opposite principle in many modular languages: the visibility of suppliers and their features is controlled by import clauses in clients. A view of a module should include all of the features that are provided by the module, even though a client may not actually import all of them.

2.3 An Example

Figure 1 shows a C++ header file that declares the class `RemoteGauge`. The declaration contains a `private` section that should not be visible to clients but is needed by the compiler. It also contains implementations of the functions `getTemp()`, `getPressure()`, and `convert()`;

again, these implementations should not be visible to clients, but the compiler needs them in order to generate inline code. The corresponding implementation file, which we do not show here, would contain implementations of the constructor `RemoteGauge()` and the function `readTemperatureAndPressure()`.

Figure 1: Header file for class `RemoteGauge`

```
class RemoteGauge
{
public:
    RemoteGauge ();
        // Constructor for gauge.
    void readTemperatureAndPressure (int mode = 0);
        // Read temperature and pressure.
    double getTemp () { return tempFahrenheit; }
    double getPressure () { return pressure; }
private:
    void convert ()
        // Convert Celsius to Fahrenheit
    {
        // Algorithm due to Anders Celsius (1850)
        tempFahrenheit = 1.8 * tempCelsius + 32.0;
    }
    double tempCelsius;
    double tempFahrenheit;
    double pressure;
};
```

Figure 2 shows the canonical document for the same class as it would appear in the one-file model. The document contains implementations for all methods although, to save space, two of them appear here as "`...`". The default visibility is `private`; accessible features are labelled `public`. Features are grouped for convenience of maintenance. For example, features concerning temperature appear in a group, as do features concerning pressure. Later, in Section 5, we present projections of this canonical document.

3 Comparison of the Models

In this section, we compare the one-file model and the two-file model with respect to various criteria. Our objective is to provide support for our opinion that the one-file model is superior to the two-file model. For balance, we address potential disadvantages of the one-file model in Section 4.

Figure 2: Canonical Document for class `RemoteGauge`

```
class RemoteGauge

public void RemoteGauge ()
    // Constructor for gauge.
{ ... }

public void readTemperatureAndPressure ()
    // Read temperature and pressure.
{ ... }

// Temperature processing.
double tempCelsius;
double tempFahrenheit;
inline void convert ()
    // Convert Celsius to Fahrenheit
{
    // Algorithm due to Anders Celsius (1850)
    tempFahrenheit = 1.8 * tempCelsius + 32.0;
}
public inline double getTemp ()
{
    return tempFahrenheit;
}

// Pressure processing.
double pressure;
public inline double getPressure ()
{
    return pressure;
}
```

3.1 Encapsulation

If the public interface of a module contains no representation information, we say that the module is *encapsulated*. There is widespread agreement today that encapsulation is a good thing because it prevents clients from making use of representation information and thereby introducing undesirable dependencies into the software. In practice, however, the situation is complicated by the existence of several degrees of encapsulation. Saying that a module is encapsulated might mean:

1. client developers cannot make use of representation information; or
2. client developers do not have access to source code that describes the representation;
3. the compiler cannot make use of representation information.

As an example, consider a private data member of a C++ class. It is encapsulated in sense (1) but not in sense (2), because the declaration of the data member is part of the class declaration, and it is not encapsulated in sense (3), because the compiler uses private data declarations to determine the size of an instance.

For client developers, encapsulation in sense (1) is desirable and is usually provided. Encapsulation in sense (2) is also desirable but is usually not provided by the two-file model.

Encapsulation in sense (3) may be undesirable because the compiler, acting on behalf of the client, does need information about the representation. Consequently, *there is a built-in conflict in the two-file model between the needs of developers and the needs of the compiler.* Client developers read the interface file in order to use the corresponding software module. They do not need, and should not see, information about the representation of the module.

In Modula-2, for example, the types exported by a module are declared in the interface file. Consequently, the representations of these types are exposed to clients as well as to the compiler. Modula-2 provides a workaround: an "opaque type" is declared as a pointer in the interface file; the declaration of the data structure corresponding to the pointer is hidden in the implementation file. This trick provides encapsulation but forces a particular implementation; the design of the language is distorted by implementation requirements.

Similarly, class declarations in C++ must include declarations for private and protected features as well as for public features. Abstract classes provide a way of eliminating these declarations, but only at the cost of additional class definitions and, in many cases, a significant performance penalty.

In summary, the two-file model cannot provide encapsulation because the interface file is performing two incompatible roles: information that should be hidden from the developer is exposed because it is needed by the compiler.

The one-file model provides views for the developer and interfaces for the compiler. The filters can be implemented to ensure that each audience receives appropriate information, no more and no less than is needed. In particular, the one-file model can support encapsulation.

3.2 Dependency Analysis

The compilation process usually involves a dependency analysis performed by an integrated development environment or by a stand-alone tool such as the UNIX utility `make`. The analysis is typically rather simple: if file A depends on file B in any way, and B has been modified since A was last compiled, then A is recompiled. Since compilation time for large systems is measured in hours or days rather than seconds or minutes, unnecessary compilation can be an expensive overhead.

For the two-file model, the policy is safe but often inefficient: any change in the interface file leads to recompilation of all client files. Many changes to interface files, however, except perhaps early in development, are actually implementation changes, for example, additions or changes to private features, that do not necessarily affect the client. Thus many modules are recompiled unnecessarily. Uncertainty about the need for particular header files exacerbates the problem in large C++ programs.

Although the one-file model does not in itself avoid unnecessary recompilation, it can be applied in such a way that some unnecessary recompilation is eliminated. The software tool that generates interfaces from the canonical document is run each time the canonical document is changed, but it creates a new projection of the machine interface only if that interface has changed. Editing comments, reordering declarations, or adding declarations that do not appear in the interface do not alter the interface. Clients are recompiled if the machine interface has changed but they do not need to be recompiled if only the developer interface has changed.

3.3 Duplication

In the two-file model, the signature of each feature must be written twice, once in the interface file and again in the implementation file. Each time a signature is changed, two files must be updated.

The two signatures are similar but not necessarily identical. Figure 1 contains the function declaration:

```
void readTemperatureAndPressure (int = 0);
```

The corresponding prototype in the implementation file might look like this:

```
void readTemperatureAndPressure (int mode) {
```

Admittedly, the difference between these declarations is not great. Nevertheless, the duplication is a source of occasional errors and much irritation.

In some older languages, the duplication problem was exacerbated by the requirement of import and export lists for individual features. To appreciate the role of a function, a maintainer might have to look for its name in the body of the interface, in an import list (to see if it came from somewhere else), and an export list (to see if it is visible to clients). A single change to a module might require textual alterations in three or four locations.

The one-file model avoids the problem of duplicated information. The canonical document contains a single definition for each feature of the software module. The definition describes all of the properties of the feature; selected properties are copied to projections by filters.

There are a few situations in which duplication cannot be avoided entirely. For example, when an inherited feature is renamed in a subclass, duplication is inevitable.

3.4 The Big Inhale

Interface files are read by many developers and maintainers, most of whom were not involved in developing the module. Good documentation is essential if an interface file is to be useful. Good documentation tends to be verbose: a one-line declaration may require several lines of comments. Consequently, the two-file model has a built-in conflict: interface files should be well-documented but, since a significant fraction of compile time is spent in reading interfaces, there is a powerful incentive to remove most of the comments. The result is that source code and documentation become separated and consistency is likely to suffer.

Developers are told to keep modules to a manageable size, but even a simple software module may need functions from a number of libraries. Since the interfaces of these libraries may contain thousands of lines, the compiler may read 10,000 lines of declarations before compiling 100 lines of useful code. Although the average ratio of interface lines to implementation lines is not always as high as this, a significant proportion of the build time for an application with many modules is devoted to reading interface files. Moreover, common interfaces are read repeatedly during the compilation of a system with many modules.

The big inhale problem can be mitigated somewhat by precompiling the interfaces, a technique introduced by Mesa and subsequently adopted by Modula-2, Ada, and other languages. The C++ directive `#include`, inherited from C, is required only to switch the compiler input stream from the current source file to another source file. Nevertheless, some integrated program development platforms do maintain a repository of compiled header file information. The effect is that recompiling after changing an implementation file is faster than with text headers, but recompiling after changing a header file may be slower because the repository must be updated.

The one-file model does not completely solve the big inhale problem, because the compiler must read all dependency information. The volume of data read, however, will be smaller than in the two-file model because comments have been removed from interface files. It can be reduced still further by encoding or pre-compiling.

Automatic dependency analysis (discussed above in Section 3.2) tends to reduce the big inhale. In C++ and other languages in which the interfaces of suppliers must be explicitly included in client code, it is hard to avoid including include header files that are not actually needed, and these may transitively include further header files.

3.5 Declaration and Use

Developers and maintenance programmers need to refer to declarations of features, a task that is simplified if uses of features are close to their declarations. Most programmers find that function definitions are easier to read if the language allows variable declarations in local scopes. Function definitions are usually fairly short, however. The separation of declaration and use is more significant when a name is declared in one source file and used in other source files.

In the two-file model, the features of a module are declared in the interface file. The signature of a method is repeated in the implementation file, at the head of the method definition, but attribute declarations are not usually repeated in the interface file. A maintainer who is modifying an implementation needs easy access to both the interface file (for attribute declarations) and the implementation file (for attribute uses). This is true even when the maintainer has no intention of changing the interface.

Maintainers using the one-file model work with one file only. Of course, they need to refer to views of other modules, but this is true for both models. Moreover, the canonical document can be organized in any way that suits the maintainer. The order of features is logical without constraints imposed by visibility. Figures 1 and 2 provide a simple illustration.

3.6 Inlines

Inline functions introduce another conflict into the two-file model for languages that provide them: the implementation of an inline function musts be accessible to the compiler but should be hidden from the client developer. This does not present a problem for the one-file model because the body of the function can be included in the client interface but not in the client view. Developers of a client module do not have to know whether or not a function is inlined. If a developer changes the body of an inline function, the client view will not change but the client interface will change, triggering recompilation of clients.

3.7 Associating Features with their Properties

There are two ways of attaching a property to a feature. One way is to mark the feature itself; for example, the keyword "`static`" in C++ applies only to the variable or function that follows it. The other way is to partition the code into sections and to associate a particular property with all of the names in the section. For example, the phrase "`public:`" introduces a section with specific visibility into a C++ class declaration. Although in principle a programmer could write "`public:`" in front of each public feature of the class, the directive is not used this way in practice. In Ada, modules are partitioned into `public`, `private`, and `limited private` sections.

The advantage of sections is that each group of names is immediately visible to readers. However, the separation of public and private features is necessary only because the interface file contains declarations that should not be there in the first place. For example, a C++ class declaration has up to three sections: `public`, `protected`, and `private`. The division helps the developer pick out the `public` functions quickly, but the other two sections are there only because the compiler needs them.

From the point of view of the owner or maintainer of a module, sections are a nuisance because they tend to interfere with the natural ordering of features. For example, if a public method of a class uses two private "helper" methods, it would be natural to put the three declarations in a single group. But this cannot be done if the class declaration is divided into public and private sections.[2] Similarly, it is sometimes useful to associate attributes with particular methods.

There is no need for sections in the one-file model. Each feature can be labelled according to its accessibility. In Oberon, for example, names flagged with an asterisk are exported and names not so flagged are private. Developers may choose whether to organize the class by accessibility, for instance, by placing all public features before private features, or by logical roles, for instance, by declaring private data features near the public functions that use them. In terms of language design, accessibility and logical organization should be orthogonal.

3.8 Comment Skew

It is notoriously difficult to maintain up-to-date and reliable documentation for large software projects. The problem is made worse by the existence of a vicious circle: since documentation is often out of date, experienced developers tend to ignore it and rely only on the code. The less they read the documentation, the less likely they are to write it.

The two-file model exacerbates the documentation problem. Comments must be written in two places rather than one, and changes to interface comments are likely to cause unnecessary recompilation. The result is all too often "comment skew" — comments that are much older than the code and that do not describe the code accurately.

In the one-file model, developers maintain only the canonical document. Comments can be placed appropriately and have no overhead, since they are not seen by the compiler.

4 Critique of the One-File Model

The arguments in the previous section claim advantages of the one-file model over the two-file model. However, the one-file model has been criticized. This section includes responses to some of the objections that might be made to the one-file model.

4.1 The Development Process

We must first dispose of an important and often-cited objection to the one-file model: Good practice dictates that the software developers agree on a complete set of public interfaces before starting to work on the implementation of those interfaces. An interface may be changed after

[2]A class declaration in C++ must declare all member functions, including private member functions.

implementation has started, but such changes should be kept to a minimum. The two-file model provides a natural way to do this: write the interface files first, then the implementation files. In Ada, packages are written before package bodies. In C++, header files are written before implementation files.

Contrary to popular belief, the same approach can readily be used with the one-file model. In the first phase, developers prepare canonical documents that describe only the public interface of each module. The owner of a canonical document may include comments that give hints for implementation but these comments do not appear in the machine-generated public interface. In the second phase, the owner completes the canonical document by filling in implementation details. At all stages, the specification can be extracted from the canonical document by a software tool.

Developers can use the compiler to validate the design at any stage of the development. The compiler can check inter-module dependencies ("does module `foo` provide function `bar`?") in the absence of function definitions although, of course, it cannot generate object code.

Practice is moving in the direction of "growing" software [3, page 201] and away from the phased, or "waterfall" approach [17]. The one-file model is fully compatible with software growth. The canonical document for each module is gradually refined as development spirals through specification, design, and implementation. At any stage, projections can be obtained automatically to show the current status of the module for clients.

4.2 Comments

Systems that require automatic processing of source text tend to have problems with comments. Conscientious programmers take great care to organize their comments for maximum readability and are justifiably upset when their carefully designed layouts are mangled by a careless formatting tool.

The tools that process the canonical document should not have to alter the layout of comments in any way. Their only problem is to decide which comments to include in a file that is generated for a human reader. They can make most distinctions on the basis of simple syntactic conventions. For example, the opening brace of a function definition in C++ can be used to separate interface and implementation documentation. In the following extract from Figure 2, the first comment is intended for clients and the second for maintainers.

```
inline void convert ()
    // Convert Celsius to Fahrenheit
{
    // Algorithm due to Anders Celsius (1850)
    tempFahrenheit = 1.8 * tempCelsius + 32.0;
}
```

An alternative approach that is already used by code generators and other tools is to provide different syntax for each kind of comment. It is then straightforward to ensure that each comment is seen only by its intended readers.

Figure 3: Interface for class `RemoteGauge`

```
class RemoteGauge
{
public:
    void RemoteGauge();
    void readTemperatureAndPressure();
    double getTemp() { return tempFahrenheit; }
    double getPressure() { return pressure; }
private:
    void convert()
    {
        // Algorithm due to Anders Celsius (1850)
        tempFahrenheit = 1.8 * tempCelsius + 32.0;
    }
    double tempCelsius;
    double tempFahrenheit;
    double pressure;
};
```

5 From One File to Two Files

Even if your favorite programming language, or the language that your employer requires you to use, assumes the two-file model, all is not thereby lost. Given a language X, and a compiler that requires separate interface and implementation files coded in X, we can define a closely-related language Y, based on the one-file model, and write a preprocessor that reads a module coded in Y and generates the files required by the X compiler. As a bonus, the pre-processor can also provide well-documented, human-readable interface files, as outlined in the description of the one-file model above.

We have developed a simple preprocessor of this kind for C++ [8]. It is called VIG, (View and Interface Generator). For obvious reasons, VIG cannot generate an arbitrary C++ program, but it is capable of generating a program that consists of class declarations, method definitions, and certain kinds of additional text. Comments are processed using the convention described in Section 4.2.

Figure 3 shows the interface generated by VIG from the canonical document of Figure 2. It is a conventional C++ header file with most of the comments omitted. The function `convert` does have a comment: this is because VIG does not parse function bodies but rather copies them directly to the output stream. Most function definition go into the implementation file, but definitions of inlined functions must be included in the header file.

Figure 4 shows the client view generated by VIG. The keywords `public` and `private` are omitted because only public features appear in the view. Other implementation details, such as inlining,

Figure 4: View for class `RemoteGauge`

```
class RemoteGauge
    void RemoteGauge();
        // Constructor for gauge.
    void readTemperatureAndPressure();
        // Read temperature and pressure.
    double getTemp();
    double getPressure();
```

are also omitted from the view. The view, however, does contain documentation that pertains to the use of public features.

The proposed approach does not eliminate the need for the compiler to read interface files, although it does reduce overhead by removing comments. The overhead of reading interface files can be reduced further by compiling them. This method was used in the Mesa project at Xerox PARC [4], in implementations of Modula-2 and Ada, and in recent integrated development environments for C++. Borland C++, for example, uses precompiled header files because "the compiler can spend up to half its time parsing header files" [2, page 221].

Unfortunately, preprocessors have a well-known drawback: errors reported by the compiler refer to the files created by VIG rather than to the files the developer wrote. An effective implementation of a preprocessor should include a tool that scans the compiler's error messages and links them to the canonical document.

6 Related Work

The one-file model is not new. For various reasons, however, it has failed to permeate the mainstream. (Java may prove to be an exception to this rule.) In the Smalltalk programming environment, the developer maintains each class as a single document that is analogous to the canonical document described here [5]. The browser presents various views of the class, including its name and position in the class hierarchy, a list of the functions it provides, and the complete source text with function and variable definitions.

The organization of Eiffel is very similar to the one-file model described in this paper [12, 13]. The developer of a class maintains a single file that contains all of the information about the class. The utility `short` extracts a human-readable interface from the class document and the utility `flatten` shows all the features provided by a class independently of their origin in the class hierarchy [12, pp. 541–543]. In Eiffel development environments, the environment provides the appropriate views when requested by the developer. In most cases, a developer can put all of the information pertaining to an Eiffel feature (member) in a single location in the source file. However, Eiffel has advanced features, such as selective renaming and repeated inheritance, that sometimes preclude this.

The rationale for the one-file model was outlined by Grogono [7] and used in the implementation of the object oriented programming language Dee [6]. The Dee compiler reads the canonical document and generates interfaces and views for clients and descendants. Interfaces are written using plain ASCII for the compiler; views are written using plain ASCII or $\LaTeX$.

Modula-2 requires separate interface and implementation files [20]. Its successor, Oberon-2, requires an implementation file only [19]. The interface file is extracted automatically from the implementation file by the development environment. Mössenböck describes this as "significant progress over the Modula-2 approach" [14, page 23].

Stroustrup points out that many people think that because they *can* put the representation of an object in the private section of its class declaration they *have* to put it there [18, page 279]. In fact, it is possible to provide an interface to an abstract base class, which has no representation and therefore requires no representation information, and to implement the module with a concrete derived class. While this solves the problem of information hiding, it introduces unnecessary run-time overhead and it does not solve all of the problems discussed in Section 3 above. In fact, it adds to the problems by providing yet more files that the developer has to maintain in a consistent state.

Java uses the one-file model and also provides facilities for structuring documentation [1, Chapter 11].

Blue is an object oriented language and environment designed for introductory programming courses [10]. The owner of a class maintains a single file, analogous to the canonical document, and the environment presents various views of this file [11].

Literate programming can be seen as a variant of the one-file model [9]. In Web, Knuth's original formulation, a "canonical document" is "tangled" into source code or "woven" into a formatted document that describes the program. Web was not used widely in its original form, but it spawned many imitations, of which the best known and most widely-used today is probably `noweb` [16]. A `noweb` user writes a single source document, and the software tool generates a human-readable document and one or more machine-readable source code files from the original document. Although `noweb` appears to be most suitable for describing implementations, it could be adapted to provide separate interface and implementation documents.

7 Conclusion

The distinction between one file and two files may seem slight, even trivial, at first sight. But a small difference can have a significant effect if it impacts the software developer's daily work patterns. The one-file model provides developers with the means to maintain well-organized canonical documents without exposing implementation details to clients or sacrificing run-time efficiency. The one-file model eliminates duplication and reduces the opportunities for careless errors while carrying out mechanical tasks that should not be necessary.

Acknowledgments. The research reported in this paper was supported in part by the Natural Sciences and Engineering Research Council of Canada.

A shorter version of this paper was presented at the IASTED International Conference on **Software Engineering and Applications (SEA 2000)**, Las Vegas, 6–9 November 2000.

References

[1] Ken Arnold and James Gosling. *The Java Programming Language.* Addison-Wesley, second edition, 1998.

[2] Borland. *Borland C++ Programmer's Guide.* Borland International, Inc, 1996.

[3] Frederick P. Brooks. *The Mythical Man-Month.* Addison-Wesley, anniversary edition, 1995.

[4] C.M. Geschke, J.H. Morris, Jr., and E.H. Satterwhwaite. Early experience with Mesa. *Comm. ACM*, 20(8):540–553, August 1977.

[5] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[6] Peter Grogono. The Dee report. Technical Report OOP–91–2, Department of Computer Science, Concordia University, January 1991. http://www.cs.concordia.ca/∼faculty/grogono/dee.html.

[7] Peter Grogono. Issues in the design of an object oriented programming language. *Structured Programming*, 12(1):1–15, January 1991.

[8] Peter Grogono and Markku Sakkinen. A view and interface generator for C++. In *Joint Modular Languages Conference*, September 2000. Submitted for publication.

[9] D.E. Knuth. Literate programming. *The Computer Journal*, 27:97–111, 1984.

[10] Michael Kölling. Blue—language specification, version 1.0. Technical Report TR97–13, School of Computer Science and Software Engineering, Monash University, November 1997.

[11] Michael Kölling. *The design of an object-oriented environment and language for teaching.* PhD thesis, Basser Department of Computer Science, University of Sydney, 1998.

[12] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall International, 1988. Second Edition, 1997.

[13] Bertrand Meyer. *Eiffel: the Language.* Prentice Hall International, 1992.

[14] Hanspeter Mössenböck. *Object-Oriented Programming in Oberon-2.* Springer-Verlag, 1993.

[15] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(12):1053–1058, December 1972.

[16] Norman Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, September 1994.

[17] W. W. Royce. Managing the development of large software systems: Concept and techniques. In *1970 WESCON Technical Papers, Western Electric Show and Convention*, pages A/1–1–A/1–9, 1970. Reprinted in *Proceedings of the 11th International Conference on Software Engineering*, Pittsburgh, May 1989, pp. 328–338.

[18] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[19] N. Wirth and J. Gutnecht. *Project Oberon: the Design of an Operating System and its Compiler*. Addison-Wesley, 1992.

[20] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, 1982.