

Bringing Portability to the Software Process

James D. Mooney

West Virginia University

Dept. of Statistics and Computer Science

PO Box 6330

Morgantown, WV 26506-6330 USA

+1 304 293-3607 X3504

jdm@cs.wvu.edu

ABSTRACT

Portability is recognized as a desirable attribute for the vast majority of software products. Yet the literature on portability techniques is sparse and largely anecdotal, and portability is typically achieved by *ad hoc* methods.

This paper proposes a framework for incorporating portability considerations into the software process. Unlike reuse, portability can be effectively attained for individual projects, both large and small. Maximizing portability, however, is more than an implementation detail; it requires reexamination of every phase of the software lifecycle. Here we identify issues and propose guidelines for increasing and exploiting portability during each of the key activities of software development and maintenance.

Keywords

Software portability, software reuse

INTRODUCTION

Portability is recognized as a desirable attribute for the vast majority of software products. In an age of ubiquitous computing and fast moving technology, there are few software products that cannot benefit from implementation in multiple environments over their total lifetime. Mass market products must leverage their cost through implementation on as many different platforms as possible. Software of any type or scale may benefit from the ability to migrate to newer and better systems as they become available.

Software engineering texts and literature universally identify portability as a desirable quality attribute (e.g. [8],[9]). Today portability is recognized and accepted as a goal even for software classes which present special difficulties, such as real-time systems and high-performance parallel applications. However, there is a scarcity of

literature addressing portability problems and strategies in a systematic way. Much of the literature is limited to case studies, and most of the techniques actually used tend to be *ad hoc*.

In particular, commonly used development methodologies do not incorporate a comprehensive strategy for achieving portability. If portability is considered it may be viewed as an implementation issue only. There is little attention paid to portability of artifacts other than the main software product, and there are no generally-accepted portability metrics.

Much attention is being paid to increasing reuse during software development, and it may be argued that porting is a type of reuse. However, the type of reuse that occurs in porting (reusing complete products in new environments) is different from mainstream reuse activities (reusing components in new products). Both goals are desirable, but most reuse strategies are not directly applicable to the goal of portability.

The absence of portability as an explicit concern in software development methods is demonstrated in a study by Song and Osterweil [10]. This study examined a broad range of software design methods with the goal of developing a framework for description and comparison. Dozens of concepts were identified that are included in one or more of these methods, including problem concepts (*e.g.*, software complexity), solution principles (*e.g.*, information hiding), and element measures (*e.g.*, cohesiveness). None of these concepts relates directly to portability.

The purpose of this paper is to propose a framework for a systematic approach to portability in the software process. Use of a sound methodology is essential to achieve this goal, but this framework is applicable to a wide range of methodologies. Key activities such as specification, design, implementation, testing, etc. occur in all approaches, and portability strategies may be primarily associated with these distinct activities.

There is ample literature in support of the use of sound, systematic methodologies for software development. The importance of such approaches for large projects is self-evident. Recent work such as the Personal Software

Process [2] demonstrates the value of these techniques for small projects as well. This paper makes no attempt to repeat these arguments.

We do argue, however, that sound methodologies are even more important where portability is an objective. The goal of producing a total system which is as portable as possible raises concerns which must be systematically addressed during every activity associated with software development. At the same time, the orderly software process provides opportunities to enhance the portability of a software product and its associated artifacts at many stages.

We will explore the portability issues for the various development activities as follows:

- Analyze the principal development activities;
- Identify the key portability concerns and opportunities for each activity;
- Propose strategies for dealing with the concerns and exploiting the opportunities.

The next three sections of this paper briefly overview portability concepts, consider how portability may be measured, and examine portability in the context of the overall software lifecycle.

Following sections of the paper examine the major activities of software development and consider their relation to portability. In each section we identify the portability issues and opportunities, and propose appropriate strategies. The activities considered are analysis and specification, design, implementation, testing and verification, documentation, and maintenance.

The paper concludes with a brief look at project management issues, and some overall conclusions.

PORTABILITY CONCEPTS

This section briefly reviews some concepts related to portability, and introduces some preferred terminology. Most of the ideas in this section are discussed more fully in [3].

Defining Portability

Porting is the act of producing an executable version of a software unit or system in a new environment, based on an existing version. There are many definitions for the attribute of portability. We will use the following as a working definition:

A software unit is portable (exhibits portability) across a class of environments to the degree that the cost to transport and adapt it to a new environment in the class is less than the cost of redevelopment.

We use the term *software unit* to indicate an application program, a system program, or a component of a program. A software system is a collection of software units. In this paper we will sometimes use the term *product* with a similar meaning.

The term *environment* refers to the complete range of elements in an installation that interact with the ported software. This typically includes a processor and operating system; it may also involve I/O devices, libraries, networks, or a larger human or physical system. Many authors use the term *platform* with a similar meaning.

The portability of a software unit is only meaningful with respect to one or more environments. We are particularly interested in portability for a *class* of environments, since we do not expect to know all likely target environments when the software is developed.

It is important to remember that porting is an alternative to redevelopment. One alternative starts with an existing implementation, the other starts with a specification. When a new implementation is the goal, a critical choice must be made as to which approach will be most cost-effective.

A software unit would be perfectly portable if it could be ported at zero cost; this is never possible in practice. Instead, we characterize software by its *degree of portability*, which is a function of the porting and redevelopment costs, with respect to a specific target environment.

Aspects of the Problem

The primary entity to be ported is usually an application or a software system. In some cases auxiliary software such as tools, libraries, or databases may need to be ported as well. Production of documentation for the new implementation may also be viewed as a porting activity. The concept of portability can also be applied to human experience; both users and programmers can benefit if experience gained with one implementation can be reused in the new one.

The principal types of portability usually considered (for software) are *binary portability* (porting the executable form) and *source portability* (porting the source language representation). Binary portability offers several advantages, but is possible only across strongly similar environments. Source portability assumes availability of source code, but provides opportunities to adapt a software unit to a wide range of environments. In some cases porting of an intermediate form provides a third alternative.

In this paper we focus primarily on the porting of a central software product rather than its associated artifacts. Also unless otherwise stated, "portability" should be taken to mean "source portability."

The porting process has two principal components which we call *transportation* and *adaptation*. Transportation is physical movement; this may not be trivial since compatible media must be used and various types of representation conversion may be required. Adaptation is any modification that must be performed on the original version; we take this to mean both mechanical translation such as by language processors, and manual modification by humans.

Costs may be associated with the use of a portability-based strategy for software development; these could take the form of increased development costs and of possible reductions in some quality measures of the actual software (e.g., performance, or conformance to system-specific user-interface conventions). The corresponding benefits take the form of reduced costs to produce and maintain future implementations, as well as possible quality improvements in factors such as reliability.

Finally, we should remember that the goal of portability is not necessarily to produce multiple implementations that behave identically in all respects. Porting may involve internationalization and localization, or adaptation to the experience levels and user interface preferences of the new environment. Significant differences in the processor architecture, memory structure, or I/O devices may call for implementations which are visibly different.

Achieving Portability

A software unit interacts with its environment through a collection of *interfaces*. For example, typical applications may have interfaces to the processor, operating system, run-time libraries, I/O devices, users, etc. If all of the interfaces for a given software unit can be made to appear identical across multiple environments, portability will be achieved. In the ideal case, common interfaces may already be present in each target environment. Otherwise, some type of adaptation will be required during the porting process.

Interfaces may take on several representations during normal software development and use. They may begin, for example, as programming language statements, which are eventually transformed into machine code. Each representation and transformation provides a distinct opportunity to adapt the interface to achieve portability.

During a decade of research and teaching in the area of portability we have identified numerous strategies for increasing portability during software development. These strategies can be summed up in three key principles:

1. Control the interfaces
2. Isolate dependencies
3. Think portable

The first principle requires the developer to identify all interfaces to the environment, and cast them in a standard form wherever possible. The second principle calls for recognition of portions of a software unit which must be adapted, and isolating these portions, e.g., into a small number of modules. The third principle is the most fundamental of all; it asks for a constant awareness on the part of the developer of the likelihood of future porting, and the impact on portability of all design decisions.

These principles, and the strategies that flow from them, are the foundation of our analysis of the software process.

PORTABILITY AND THE SOFTWARE LIFECYCLE

Most software goes through a period of development followed by an extended period of use, during which many changes may occur. It is generally agreed that a sound economic view of software costs must be based on an understanding of the complete software lifecycle.

At the same time, there is usually pressure to focus on the current part of the lifecycle, rather than later parts (Get it out the door, fast and cheap). This leads to much greater costs during the “maintenance” activities that make up most of the lifecycle. Parnas [6] has observed that long life is the norm for much software, and that it will be an indication of the maturity of software engineering when we can focus on long-term viability, not just the “first release.”

Portability is a test issue for Parnas’ challenge. For much software the likelihood of eventual porting is high. A portable design up front can reduce porting costs, and often reduce total costs for the entire life of the product. However, developing portable software adds costs during initial development, in return for benefits which will come later. If the focus is on “getting it out the door”, portability will receive little consideration. If instead a more holistic view is taken, based on total lifecycle costs, portability should become a standard element of the software process.

A number of models of the software lifecycle are used both to understand the lifecycle and to guide the overall development strategy. Most widely known is the waterfall model, in which activities progress more or less sequentially through specification, design, implementation, and maintenance. Recently popular alternatives include rapid prototyping and the spiral model, with frequent iterations of the principal activities. Testing (and debugging) and documentation may be viewed as distinct activities, but are usually expected to be ongoing throughout the process.

Each of the principal activities of the lifecycle is associated with some distinct portability issues. However, the sequencing and interleaving of these activities, which distinguishes the models, does not substantially affect these issues. Thus our discussion is applicable across the usual models, but will focus primarily on the individual activities.

MEASURING PORTABILITY

There is an increasing focus on measurement in the software development process to evaluate the success of various techniques and document properties of the software product. To incorporate portability solidly into the process, metrics related to portability must be available for use.

Elsewhere [4] we have proposed some possible process and product metrics for portability. Standard cost estimation techniques, especially those that distribute cost over the various lifecycle activities, may be used to estimate the development cost penalties incurred by portable development. During the maintenance phase, the costs of

both porting and redevelopment may be estimated, given a specific product and a specific target environment. Analysis of porting costs involves analyzing the match between the interfaces of the software unit and those of the target. A figure for degree of portability can then be computed for a specific software unit and target (or target class):

$$DP = 1 - (\text{cost to port} / \text{cost to redevelop})$$

If this value is greater than zero, then porting is more cost-effective than redevelopment. Moreover, porting costs will be inversely proportional to the DP; a value of one represents “perfect portability.”

ANALYSIS AND SPECIFICATION

It is axiomatic that good software development should start with a careful process of requirements analysis, leading to the creation of one or more sound written specifications. In this section we assume that such specifications are being developed.

The purpose of the specification is to identify the functionality and other properties expected in the product to be developed. There are many proposed structures for such a specification, ranging from informal to fully formal, mathematical notations like Z. Formal notations in current use express the *functional* requirements of a software product, but are not designed to express non-functional requirements such as reliability, performance, or portability. If such requirements exist they must (presently) be expressed by less formal means.

There is little common agreement on the best form or organization for specification documents. A number of alternatives are described, for example, by Sommerville [9].

Our framework proposes two guidelines for the specification activity to maximize portability:

1. Avoid portability barriers
2. Specify portability goals explicitly

Avoiding Portability Barriers

Avoiding barriers means that statements in the specification should be checked to ensure that they contain no unnecessary system dependencies. For example, the following statement is fraught with dependencies:

An object may be selected by two clicks of the right mouse button no more than 1/2 second apart, within 10 pixels of the object center.

There is probably no reason to insist that the display screen have a certain resolution, that the mouse have two buttons, or even that the pointing device must necessarily be a mouse. A more appropriate statement, avoiding unnecessary detail, might be

An object may be selected by use of a suitable pointing device.

The majority of system dependencies here are likely to relate to the user interface, but not all. For instance, there

is no need to specify that all integers must be represented by 32 bits, if you require only that some values may range between zero and one million.

Stating Portability Goals

The specification provides an opportunity to identify portability as an explicit goal, and to balance this goal against other system requirements. There is no use to a statement such as:

This program shall be easily ported to new platforms.

This is a meaningless requirement which cannot be tested. Instead, we must characterize portability in a way which can be achieved and measured.

Portability is only meaningful with respect to particular target environments. One aspect of a portable specification must be a statement of expected target environments. This should not be a list of specific systems (Windows NT, Solaris, Apple Newton, etc.) Developing software for a set of targets whose details are known is not the same as developing portable software. Instead the target environments should be characterized as a class. For example, the following statement could characterize the class suitable for a visualization program:

This software may be ported to any personal computer or workstation environment supporting at least 16-bit color on a 15 inch display, achieving a SPECfp95 benchmark rating of at least 5.0, and having a data storage capacity of at least 8 MB.

Here we identify the essential requirements for a high-quality animated display with a large data set, but we do not specify details of the architecture or operating system.

A second aspect of portability goals is to determine how much portability is required. The goal of portability may be limited by conflicting goals such as high performance or exploiting the features of a specific environment. Development costs and timetables will be affected by this choice.

There is no commonly-accepted method for quantifying portability, but the Degree of Portability discussed above offers one possible approach. Given a target class of environments, we may specify a target value for such a metric.

A portable software unit is not finely tuned to a particular environment, and may exhibit lower performance or less efficient resource usage than a tuned implementation. It is not feasible to specify the degree of penalty that might be tolerated here, since this would require both a portable and a non-portable implementation for comparison. However, the specification may include performance goals that must be met *in spite of* portable design.

DESIGN

Design is the heart of software development. Here our understanding of what the software is to do, embodied in the specification, directs the development of a software architecture to meet these requirements.

A large software project may require several levels of design, from the overall system architecture to the algorithms and data structures of individual modules. A systematic design method may be used, such as Structured Design, SADT, JSD, OOD, etc. The various methods have widely differing philosophies, and may lead to very different designs. However, they share a common objective: to identify a collection of elements (procedures, data structures, objects, etc.) to be used in implementing the software, and to define a suitable partitioning of these elements into modules.

The resulting design (perhaps at various levels) has the form of a collection of interacting modules which communicate through interfaces. It is well understood that clear and careful interface design is a crucial element of good software design.

Ideally, a software design is independent of any implementation and so is perfectly portable by definition. In practice, the choice of design will have a major impact on portability.

Portability issues in design are focused on partitioning. The guidelines are therefore applicable to most design methods. However, some design methods may be more favorable to portable design. For example, object-oriented design provides a natural framework for encapsulating external resources.

External Interfaces

A software product is only complete when it is embedded in a particular environment; so a complete design must reflect interfaces to the environment as well as internal interfaces between modules. These interfaces are easy to overlook; they often take the form of ubiquitous calls to services such as file access, memory management, or user interface services which may be used freely throughout an implementation. If the form in which these services are presented varies across target environments, a substantial portability problem arises.

To avoid this problem, external interfaces must be promoted to first class status when evaluating the merits of potential designs. quality measures such as cohesion and coupling should be applied equally to external connections and internal ones.

To achieve the goal of a highly portable design, all external interfaces (even seemingly trivial ones such as print statements) should be *identified*, then either *standardized* or *isolated*.

Identification

To identify external interfaces for a given software unit we ask the question: What resources and services does this unit

access or rely on, which are outside of the system being developed? Obvious answers to this question will include any services explicitly called, such as file access, display operations, etc. Less obvious may be implicit expectations about the environment, such as floating point accuracy or storage capacity.

This question should be asked for the system as a whole and for each of its subsystems and modules at every level of partitioning.

Use of Standards

Each of the interfaces should now be examined to determine if there are appropriate standards which should be followed. A standard is an appropriate solution if it is one which is likely to remain stable and be supported, or easily supportable, in most anticipated target environments. A few standards that are good choices in a very wide number of cases include ASCII for character sets, POSIX for operating system interfaces, IEEE floating point, etc. In particular application domains or organizations, other standards may be obvious choices.

The choice of programming language is ideally deferred until implementation. However, it may be necessary to identify the probable language at this time, since languages vary widely in the interfaces which they cover. If at all possible a language should of course be chosen which is well standardized and widely available; the most likely candidates for general use at the present time are Ada, C, and C++. Language choice is further discussed in the next section.

This process should identify the most portable structure for some, but not all, of the external interfaces.

Isolation

Interfaces which cannot be expected to be supported in a consistent way across most targets may require adaptation. One strategy for reducing this problem is to isolate all accesses to this interface to a very small number of modules. For example, low-level operations on special I/O devices may be confined to a single module, which is accessed by higher-level calls throughout the system.

When accesses to an interface are not easily isolated, the alternative is encapsulation. This is a more general solution but may introduce small performance inefficiencies. The resource and its interface may be encapsulated in a module which defines its own structure for the required accesses. This structure becomes an internal standard for the program. System-dependent accesses to the native interfaces are again isolated in a single module.

IMPLEMENTATION

Implementation is concerned with transforming a design into a working software product. If good design practice has been followed, the design in most cases should not be platform-specific, even if it is not explicitly portable.

In most cases, the implementation targets one specific environment. Occasionally, versions for multiple environments are implemented simultaneously. During portable development, it is also possible to envision an implementation which has no specific target, but is ready for porting to many environments. We will not consider this case.

Developers who strive for portability most frequently concentrate their attention on the implementation phase, so the issues here are fairly well understood. The principal guidelines are:

1. Choose a portable language
2. Follow a portability discipline
3. Apply standards carefully

Choosing a Language

The appropriate language must be identified at this time if it has not been determined previously. There are many criteria for choosing a programming language, such as ease of use, safety, availability, suitability for the application, etc. Portability should be prominent on this criteria list. This criterion will favor the selection of a well-standardized, widely available language such as Ada, C, or C++ (or in some domains, FORTRAN or COBOL).

An additional criterion is whether the language will work well with the other standards identified during the design phase. Most interface standards require an auxiliary *language binding* to specify how to use them with a particular programming language. If appropriate language bindings for the chosen language have not been standardized, consistent syntax cannot be assured across multiple implementations.

Identifying the language is the first step; the process continues by specifying the portable subset of the language that will actually be used. All portable languages include elements which are recognized as portability problems; for example, many operations that blur the distinction between data types are legal in C but highly non-portable. Resources such as [1],[7] may be consulted to determine which elements should be avoided.

One element of the language that is easily overlooked is the use of compiler directives for purposes such as optimization or listing control. These are highly non-portable, unless the same compiler can be used for most target environments.

Portability Discipline

Beyond the specification of the language, there are many ways in which potential problems can be avoided by conscious attention to portability (thinking portable) on the part of the developer. There is not space in this paper to survey all of the issues that may arise. One example is the need to be aware of and stick to the least upper bounds that can be relied on for data type ranges, nesting depth, identifier lengths, etc. Another is to recognize further

constraints that may be imposed on external names by various linkers and system libraries.

Applying Standards

The standards identified during design, including internal standards, must be applied during implementation. The implementor must try to use only the structures defined in the standard when accessing the interfaces which it covers. All standards leave some questions unresolved, which may be discovered during implementation. The developer must identify these as likely system dependencies, and make an educated guess as to the interpretation that is likely to be consistent with the greatest number of target environments.

TESTING AND VERIFICATION

Testing is an essential activity for any type of software development. An increasing number of projects also make use of formal verification, to demonstrate a high likelihood of correctness by logical reasoning.

The goal of testing is to verify correct behavior by observation in a suitable collection of specific test cases. It is not possible to test all cases, but there are well-known techniques to generate sets of test cases which can cover most expected situations and lead to a reasonably high confidence level in the correct operation of the software.

The guidelines for the testing activity are:

1. Develop a reusable test plan
2. Learn from errors
3. Test portability itself

Reusable Test Plan

Good practice in testing calls for the use of a systematic written test plan. This plan is even more important when porting is anticipated, since it can be largely reused to test each new implementation. Differences will occur only when there are deliberate differences in intended behavior between implementations (such as user interface differences). In this case some of the tests will be system-dependent. In many cases system-dependent tests can be identified in advance; if so they should be isolated in a separate portion of the test plan. This will be the only portion of the plan that has to be redesigned for a new implementation.

Learning from Errors

When testing reveals errors, a debugging process must be invoked followed by repetition of earlier development activities as necessary to correct the problems. Careful records of the errors found, and the modules in which they occur, will be valuable in porting. It is reasonable to expect that ported modules, once successfully tested, which undergo no changes, are less likely to introduce new errors after porting.

Testing Portability

If portability has been specified as a required property of a software product, this attribute itself must be tested in some manner. This cannot be done by an execution test; it requires an analysis which compares the product to one

or more typical environments in the target class. This comparison can be used to estimate porting costs and so compute a Degree of Portability for the product.

Finally, formal verification raises the question of what portion of the verification must be repeated after porting to ensure that the conclusions still hold. This is an open area for continuing research.

DOCUMENTATION

Many types of documents are associated with a well-managed software process. Portability will have an impact on the documentation activity as well as the other development phases.

Portability guidelines for documentation are:

1. Develop portable documentation
2. Document the porting process

Portable Documentation

There are many types of documentation associated with a typical software project, including specifications, design documents, source listings, programmer's manuals, user's manuals, etc. For a portable software product, a large part of each type of documentation may apply without change to multiple implementations, while a smaller part is implementation-specific.

The principal task of portable documentation is to identify and separate system-dependent and independent portions into distinct sections or separate documents. For a new implementation, ideally, only the limited system-dependent documentation needs to be redeveloped.

This practice should be followed as long as it does not seriously impact the usability of the documents. In particular, some user manuals cause confusion by including numerous sections that apply to various specific systems, when the user wants to focus on just one system. Here is a case where portability may not be the primary consideration!

Documenting the Porting Process

A goal of portability calls for one new type of documentation: the porting process itself must be documented. The developer is aware of the probable system dependencies and the types of adaptation that may need to be performed. Instructions should be developed to guide the porting activities during the maintenance phase.

MAINTENANCE

The maintenance phase is the payoff for portable development. Each requirement to produce an implementation in a new environment should be greatly facilitated by the efforts to achieve portability during original development. Other maintenance activities, such as error correction and feature enhancement, will not be impeded by portable design and may possibly be helped.

The only complicating factor is the need to maintain multiple versions. Where possible, clearly, common code should be maintained via a single source, if the versions are

under the control of a common maintainer. Issues of multiversion maintenance are widely discussed in the literature and will not be repeated here.

In the rest of this section we discuss the porting activity itself. We assume that the software to be ported was developed for portability, but in principle the same ideas apply when porting "non-portable" software as well.

Porting the Software

When a new implementation is desired, the most fundamental question is whether to port or redevelop (hybrid approaches may also be possible). The Degree of Portability metric may guide us in this choice. If the DP is positive, and especially if it is fairly close to 1.0, porting is clearly the most cost-effective choice.

Computing the DP requires estimation of the anticipated costs of porting and redevelopment. This information will guide the planning process during the implementation.

The starting point for porting is an existing implementation, together with the porting documentation (and all necessary documentation for the target system).

The porting process is a subset of the normal software development process. Ideally, specification and design can be omitted, unless there is a clear need for visible design changes.

Implementation, guided by the porting documentation, should require limited manual adaptation to only a small subset of program modules, followed by automatic transformation by tools such as compilers.

Testing of the ported software can make use of the portable test plan. In general fewer errors may be expected with subsequent implementations. Good practice requires full regression testing, although in fact many tests would not be expected to produce different results.

Documentation for the new implementation may be derived from the portable documentation set already developed. The cost of this activity will be greatly reduced for new implementations.

A newly ported implementation will be maintained as an added version along with previous ones, if the same organization is responsible. This will add only moderate costs to the maintenance activity, unless this is the first additional version.

Finally, we should note that even if a piece of software is fairly portable, the option of making it more portable during reimplementations should be considered. This would require some reversion to earlier activities of the development process, but might yield payoffs during later maintenance.

MANAGEMENT ISSUES

Portability is widely seen as a desirable attribute for most software by computing professionals, but incorporating portability into a development project may require support from management (and customers).

Fortunately, portability is easier by far to incorporate than the related, more popular concept of reuse. Most successful reuse programs are institutionalized in a single development organization, and provide gradually increasing benefits over multiple development projects. Trying to practice reuse in a single isolated project is less likely to be successful.

By contrast, portability may be successfully incorporated in individual projects, large or small, or even in individual portions of a project. Even a small increase in portability is helpful when later implementations are required.

However, the cost distribution for portability is one which may seem unfavorable to managers and customers; the initial implementation may be later, more expensive, or even less perfectly tuned. The benefits will come at a later, less certain point in the life cycle. Only when we are fully committed to a "total lifecycle" view of the product and its costs will these objections be completely overcome.

CONCLUSION

This paper has reviewed portability issues for each of the principal activities of the software development process, and proposed guidelines for dealing with portability problems and taking advantage of the benefits of portability. These guidelines collectively provide a practical framework for incorporating portability more systematically into many software development projects.

The author maintains a Software Portability Home Page on the World Wide Web [5]. This page contains information and links related to the discussions in this paper.

ACKNOWLEDGMENTS

I would like to thank my colleague Murali Sitaraman and the students of the Software Portability Group at WVU, along with the many students of CS 374, for helping to develop and formulate many of these ideas.

Thanks also to the National Science Foundation and the State of West Virginia for their continuing support. This work is supported in part by NSF and the State of West Virginia through the EPSCoR program.

REFERENCES

1. Giencke, P. *Portable C++*. McGraw Hill, New York NY, 1996.
2. Humphrey, W.S. Using a Defined and Measured Personal Software Process. *IEEE Software* 13, 3 (May 1996), 77-88.
3. Mooney, J.D. Strategies for Supporting Application Portability. *IEEE Computer* 23, 11 (Nov. 1990), 59-70.
4. Mooney, J.D. Issues in the Specification and Measurement of Software Portability. Technical Report TR 93-6, Dept. of Statistics and Computer Science, West Virginia University, Morgantown WV, 1993.

5. Mooney, J.D. Software Portability Home Page. <<http://www.cs.wvu.edu/~jdm/research/portability/home.html>>.
6. Parnas, D. Software Aging. *Proc. 16th Int. Conf. on Softw. Engr.*, Sorrento, Italy, 1994, 279-287.
7. Rabinowitz, H., and Schaap, C. *Portable C*. Prentice Hall, Englewood Cliffs NJ, 1990.
8. Schach, S.R. *Classical and Object-Oriented Software Engineering (3rd ed.)*. Richard D. Irwin, Chicago IL, 1996.
9. Sommerville, I. *Software Engineering (5th ed.)*. Addison-Wesley, Reading MA, 1996.
10. Song, X., and Osterweil, L. Toward Objective, Systematic Design-Method Comparisons. *IEEE Software*, 9, 3 (May 1992), 43-53.

