# The Impact of Software Engineering Research on Modern Progamming Languages

BARBARA G. RYDER
Rutgers University
MARY LOU SOFFA
University of Virginia
and
MARGARET BURNETT
Oregon State University

Software engineering research and programming language design have enjoyed a *symbiotic* relationship, with traceable impacts since the 1970s, when these areas were first distinguished from one another. This report documents this relationship by focusing on several major features of current programming languages: data and procedural abstraction, types, concurrency, exceptions, and visual programming mechanisms. The influences are determined by tracing references in publications in both fields, obtaining oral histories from language designers delineating influences on them, and tracking cotemporal research trends and ideas as demonstrated by workshop topics, special issue publications, and invited talks in the two fields. In some cases there is conclusive

This article has been developed under the auspices of the Impact Project. The aim of the project is to provide a scholarly study of the impact that software engineering research—both academic and industrial—has had upon practice. The principal output of the project is a series of individual papers covering the impact upon practice of research in several selected major areas of software engineering. Each of these papers is being published in ACM TOSEM. Additional information about the project can be found at http://www.acm.org/sigsoft/impact/.
This article is based upon work supported by the U.S. National Science Foundation (NSF) under award number CCF-0137766, the Association of Computing Machinery Special Interest Group on Software Engineering (ACM SIGSOFT), the Institution of Electrical Engineers (IEE), and the Japan Electronics and Information Technology Association (JEITA). Any opinions, findings and conclusions or recommendations, expressed in this publication are those of the authors and do not necessarily reflect, of the NSF, ACM SIGSOFT, the IEE, or JEITA.
Authors' addresses: B. G. Ryder, Division of Computer and Information Sciences, Rutgers University, 110 Frelinghuysen Road, Piscataway, NJ 08854-8019; email: ryder@cs.rutgers.edu; M. L. Soffa, Department of Computer Science, School of Engineering and Applied Science, University of Virginia, 151 Engineer's Way, P.O. Box 400740, Charlottesville, VA 22904-4740; email: soffa@cs.virginia.edu; M. Burnett, School of Electrical Engineering and Computer Science, Oregon State University, Corvallis, OR 97331-5501; email: burnett@cs.orst.edu.

data supporting influence. In other cases, there are circumstantial arguments (i.e., cotemporal ideas) that indicate influence. Using this approach, this study provides evidence of the impact of software engineering research on modern programming language design and documents the close relationship between these two fields.

Categories and Subject Descriptors: D.2.0 [**Software Engineering**]: General; D.3.0 [**Programming Languages**]: General; D.3.3 [**Programming Languages**]: Language Constructs and Features; D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.1.7 [**Programming Techniques**]: Visual Programming

General Terms: Design, Languages, Reliability, Security

Additional Key Words and Phrases: Software Engineering, Programming Languages

## 1. INTRODUCTION

This article is part of the larger NSF-sponsored Impact project that is documenting the influence of software engineering research on software practice. In this article, we focus on the impact of software engineering research on the design of programming languages from a historical perspective. That is, the article is not a survey of software engineering accomplishments, but instead focuses on documenting software engineering research that has influenced the design of programming languages. The influence was determined by examining publications, oral reports, and the time sequence between significant advances in software engineering and programming language design.

The production of software requires software engineering techniques, such as specification, design, implementation, testing, and maintenance. Essential to performing the last three phases of software development is the selection of a programming language(s) as an implementation vehicle. The programming language(s) chosen needs to offer the application programmer the power to naturally express the task at hand in a disciplined manner. The requirements of expressiveness and software engineering methodology, evolving over time, clearly impact the design of programming language features.

Thousands of programming languages have been designed to address the unique demands of application areas. Many have been developed purely for research exploration, while others have been targeted for production use. Some of the languages are general purpose while others were developed for particular domains. To name a few programming languages used in practice today, $C++$, Java, Ada,[1] Perl, Visual Basic, and COBOL all are widely-used across many application domains. Software engineering and the design of programming languages enjoy a synergistic relationship, each influencing the other. This relationship holds with regard to both research and practice in these two fields.

The main goal of this report is to document how fundamental research in software engineering has been a resource for programming language features commonly used today. To achieve our goal, it is necessary to trace the ongoing interrelations between software engineering research and programming language design. In examining these interrelationships, we have found symbiotic

[1]We use the term *Ada* to refer to *Ada 83* in this article; we will use *Ada 95* to refer to the later version of the language.

interactions between these two fields that have both influenced and strengthened each other. Thus, in this report we track research both in programming languages and software engineering.

In accomplishing this goal, there are a number of challenges that must be addressed. The synergy between software engineering research and programming language design renders attribution of some specific contributions difficult. Initially, there was no distinction between the software engineering and programming languages research communities. Research in programming languages design, specification, programming methodology, and software engineering was being done by members of a single amorphous field. This field had the same conference publication venues until the late 1970s (e.g., IFIP Working Group 2.3 formed in 1969, *NATO Software Engineering Conferences* 1968, 1969). Initial research contributions in both fields were essentially shared contributions to this original, single software field. In the middle of the 1970s, this community began to split into the fields of software engineering and programming languages. The first in the series of the *ACM SIGACT/SIGPLAN Symposia on Principles of Programming Languages (POPL)* was held in 1973, while the first *IEEE/TCSE ACM/SIGSOFT International Conference on Software Engineering (ICSE)* was held in 1975. These two conferences mark the beginning of the differentiation between the two communities. As the fields began to evolve into distinct communities, researchers who had published in the same conferences and journals now began to publish in both software engineering and programming languages conferences, and thus, the classification of them as either a software engineering or a programming languages researcher is difficult. Interestingly, these two communities have continued to grow further apart.

Another challenge is differentiating between the contributions of software engineering research and software engineering practice, since both have influenced the design of programming languages. Questions to be considered include the following: *Was a feature of a programming language changed due to a new research idea or due to the experiences of practicing software engineers? What influenced the wide acceptance of object-oriented programming that current languages support? Does this show the influence of practitioners' experiences or the adoption of a new paradigm explored by researchers or both?* Because of the interrelatedness of languages, software engineering practice and software engineering research, these questions are difficult to answer.

Finding primary sources to document the influence of software engineering research presents an additional challenge. Research publications are needed to determine this impact; examination of the topics of meetings and workshops held at specific times also yields some information as to software engineering concerns. The oral and written history of how a programming language evolved is also needed to complete this study. Often the communities feel that certain concepts were *in the air* [Gabriel personal communication] in that everyone agreed these were important ideas, but no one really knew where they originated, or who was instrumental in developing them.

Clearly, studying the evolution of the concepts and features of all programming languages that have been developed presents an intractable problem. The challenge here is to select specific languages to study and their significant

features, to render the problem feasible. The existence of software process, language tools, and environments adds more complexity to the study because they also have had influence on research and practice in both software engineering and programming languages. Thus, we focused only on the software engineering concepts that made their way into programming languages; we did not address software process, tools, and environments.

The final challenge is the determination of an effective research methodology for this study. One possible approach is to examine this question from a history of science perspective; that is, focusing on practice at a given time and place, asking why and how ideas originated and evolved. Our approach has been to examine results of research as evidenced in published works and to infer influence from these results. We also have conducted interviews with programming language designers that explore what they were thinking and how they were being influenced at the time of accomplishing their designs.

This article discusses the impact of software engineering research on a selected set of widely used languages: Java, *C++*, Ada, and Visual Basic, and some of their significant concepts. Concepts considered include abstractions, visibility, reuse, exception handling, types, concurrency, and visual programming features. The research methodology used to study the historical interactions between software engineering research and programming language design takes into account the iterative evolution of programming languages and the influence of software engineering research on this evolution.

The research methodology is described in Section 2. Section 3 traces the development of individual programming concepts. Section 4 recounts the development of visual programming and its relationship to Visual Basic and to software engineering research. Section 5 summarizes interviews conducted with programming language designers, in which they were questioned about what influenced their designs. Finally, Section 6 summarizes the findings.

## 2. RESEARCH METHODOLOGY

Given the focus of this study, several languages, selected because of their importance to current programming practice, were examined for the constructs and the concepts they expressed.

Based on the constructs found in many languages, a number of languages were considered as possible choices for the focus of this study. Neither COBOL nor FORTRAN was selected because, although they remain in wide use today, they are not used as general purpose programming languages. COBOL is, and has been, used for business applications; FORTRAN is the language of choice for scientific computation. Likewise, C is used mainly as a systems programming language.

Ada, *C++*, and Java are widely used for general purpose applications. Visual Basic is a language widely used for fast prototyping, relying primarily on Basic plus graphical interfaces to leverage programmer productivity. We included Visual Basic in its own section of the report as a representative of visual programming in practice and because of its widespread use. Thus, the focus of this study centers on general purpose languages: Ada, *C++*, Java, and Visual Basic.

This language selection is not an attempt to be definitive about current practice, nor does it represent a value judgment about the importance of languages in specific application areas. These language choices have many of the features expected in today's general purpose languages and also enable the tracing of historical software engineering influences.

### 2.1 Software Maturation Models

In 1985, Redwine and Riddle [1985] presented a maturation model of software technology, which discusses the steps for transition from a research concept to practice. The goal of the work was to discover the process, principles, and techniques useful in transitioning modern software to widespread use. They identified six major phases.

—**Basic research:** investigate ideas and concepts that eventually evolve into fundamentals and identify the critical problems and subproblems.
—**Concept formulation:** key idea or problem is identified and the ideas are informally circulated, convergence of a set of ideas, and publications start to appear that suggest solutions to subproblems.
—**Development and extension:** present a clear definition of a solution in either a seminal paper or a demonstration; preliminary use of the technology; generalize the approach.
—**Internal enhancement and exploration:** extend approach to another domain, use technology to solve real problems; stabilize and port technology; development of training materials.
—**External enhancement and exploration:** similar to previous phase but performed by a broader group, including outside the development group.
—**Popularization:** show substantial evidence of value and applicability; develop production-quality version, commercialize and market.

In a keynote address at *ICSE 2001*, Shaw [2001] used this model to describe the *coming of age* of software architecture. The first phase, the basic research phase, involved the exploration of the advantages of specialized software structure for software process. This phase also cataloged systems to identify common software architectural styles, leading to models for explaining architectural styles. In the concept formulation phase, architecture description languages were developed and architectures were classified. In the development and extension phase, languages were extended and new languages developed, as the concepts were refined. Meetings, journals and conferences that were devoted solely to software architecture were seen. In the next phase, formal analysis of real systems was performed. In the external enhancement and exploration phase, Unified Modeling Language™ (UML) was developed, as an example. Finally, in the popularization phase, standards were developed, demonstrating that software architecture is accepted and used in software engineering.

We developed a similar maturation model for programming language design and software engineering and used this model to determine the interactions of programming languages and software engineering. Our model has five phases

followed by iterative refinement using the last two phases. They are:

—**Basic research:** identify a software engineering problem and explore basic ideas and concepts.
—**Concept formulation:** circulate ideas informally to develop a research community and publish solutions to the problem or subproblems.
—**Introduce into a programming language:** frame a solution to the problem in a programming language, which is used for a period of time.
—**Refinement:** as a result of using the language feature as well as exploring other facets of the problem, solutions are refined and extended.
—**Extend the programming language or develop a new one:** based on the refinements, an existing language is either changed or a new one is developed.

The last two phases are repeated until software engineers and programming language designers find an acceptable solution.

### 2.2 Methodology and Resources Used

To provide focus, we used the model to explore concepts in object-oriented and imperative languages that are commonly used; that is, we examined modern programming languages and identified a set of core features, as was done in Shaw et al. [1978]. We used these core features to demonstrate the relation between software engineering research and programming languages.

An extensive survey was performed in both the software engineering and programming language research literature to find references to the constructs and concepts. Historical papers describing the evolution of particular programming languages were also examined. Key to the approach was, for each concept, performing a reverse chronological search through the materials gathered to look for interactions, both cited and implied,[2] between software engineering research and programming language design. Of greatest interest were the origin, development, and evolution of these concepts to their current form. The papers found through this search were then read for evidence of influence on programming language design. We used time lines to help determine the interrelationships between the fields and their potential impact on one another. In many cases, we used the date that an article was published as an indication of possible influence.

We also conducted interviews with language designers to acquire firsthand historical information. With the help of Mike Mahoney, History of Science Professor at Princeton University, we developed a set of five questions and emailed the questions to the language designers. We gave them the option of answering the questions by email or by phone. In some cases, we had further contact. A section summarizing the responses appears in this article in Section 5. The language designers participating in the interviews were Professor Niklaus Wirth (Pascal, Modula), Professor Boris Magnusson (Simula), Dr. Bjarne Stroustrup

[2]As an example of an *implied* interaction, we may infer a relation between the temporal precedence of a published software engineering result and the subsequent appearance of a seemingly related programming language construct.

(*C++*), Dr. Jean Ichbiah (Ada), Tucker Taft (Ada 95), Tim Lindholm (Java) and Dennis Canady (Visual Basic).

The resources used included both journal and conference papers in software engineering and programming languages. These written documents were acquired through the usual digital and hard copy library sources as well as through personal communication. In addition, we used first person interviews that were published in various venues. With the development of later languages came rationales about why language features and their forms were included; these provided another resource. Last, because many of the ideas that impacted software engineering and programming languages were in the air, we used the results of the interviews that we conducted for insight and further references. We also used the existence of workshops and conferences in different time periods to indicate significant issues being considered by researchers.

In the next section, we apply this technique to several language features as they evolved to their usage in Java, *C++*, and Ada. These include control and data abstractions, inheritance, types, exception handling, and concurrency.

## 3. LANGUAGE FEATURES

In tracing the interaction between software engineering and programming languages, we begin with the design of abstractions, including control, procedural, and data, which demonstrate a clear symbiotic relationship between research in the programming language design and software engineering communities, to the benefit of software practice.

### 3.1 Abstraction

With the development of increasingly more complex applications, abstraction has become a critical mechanism to control complexity in software. Thus, it is a major part of programming language design and software engineering methodology, necessitating interaction between the two communities. Modern programming languages support abstraction by enabling the user to express what is significant and suppress what is not [Shaw 1984]. There are basically three types of abstractions used when constructing software, namely, *data, control*, and *procedural abstractions*. Most widely used programming languages today include a number of different constructs to support these various forms of abstractions. The abstraction mechanisms in languages have been affected by past experiences in software engineering research and practice.

3.1.1 *Control Abstraction.* Control abstraction describes a well defined operation that expresses the execution sequence of statements (actions) in a program and includes intraprocedural control structures such as *if-then-else*, *while*, and *switch* statements. Typically all languages have built-in control abstraction. During the 1960s, indiscriminate transfer of control was believed to cause difficulties and errors in writing and understanding programs. In the late 1960s and early 1970s, two important software design methodologies emerged that stressed the importance of properly organizing and structuring programs. One method, called *stepwise refinement* or *top-down design*, initially created an

abstraction of the program without details about the data structures or operations [Wirth 1971a]. With each step, the program is successively refined until the final code is constructed. Thus there is a separation between the abstract definition of structures, and procedures and their implementation.

About that same time, there was a debate about the harmful effects of *go to* statements [Dijkstra 1968b; Rubin 1987]. Intertwined with the discussion of *go to*s was the structured programming movement [Dijkstra 1969; Dahl et al. 1972]. Consequently, the languages developed after the 1970s had structured control constructs and deemphasized or eliminated *go to* statements. However, there were a few cases where escaping from statements was felt to be necessary [Hopkins 1972], and languages have either included a *go to* or put in other constructs that enabled a break out of a statement, including breaking out of loops and case statements.

Both top-down design and structured programming are mechanisms to make software simpler, more understandable, and easier to modify. They enable abstractions to be specified and then implemented. Control flow patterns in a program are instrumental in developing and understanding a program. From the software engineering research on structured programming and top-down design, control structures with a single entry point and a single exit point made programs more understandable. This led to a set of accepted intraprocedural control constructs [Shaw 1984]. All commonly used languages feature the single-entry, single-exit (almost) design in their control structures.

Another example of programming methodology influencing programming language design comes from *generators* in Alphard which became instantiated as iterators in CLU. These structures expressed the concept of iterating through a set of elements or objects [Shaw et al. 1977; Liskov et al. 1977]. The idea was to separate the abstraction of the concept of iteration from the implementation. This idea is found in object-oriented languages, which have the concept of iterating through a collection of objects.

Thus, the research of the early combined software community on developing, understanding, and modifying programs influenced the design of structured programming language constructs.

3.1.2 *Procedural Abstraction.* *Procedural abstraction* deals with organizing and decomposing a program into modules that typically have specific purposes. Procedural abstraction involves the definition of an interface and an implementation of the unit. Units were composed of local data and code that was able to access nonlocal data as well as local data. FORTRAN introduced procedural abstraction through its subroutine and function program units. These could be developed independently of the programs that call them and could be separately compiled. Simula 67 also had procedural abstraction through the class structure with functions. Neither FORTRAN or Simula 67 had any separation between the interface and the implementation, nor were the types in the interface checked when used.

In 1971, Parnas wrote a most influential paper that formally introduced the importance of modularity. The paper describes a method and need for decomposing a program into modules [Parnas 1971, 1972]. The idea of *information*

*hiding* was also introduced in this paper by showing that the main benefit of hiding implementation was to allow changes to the implementation to be confined to its module. Parnas characterized a module by its knowledge of a design decision, which is hidden from all other modules. For example, in the paper a symbol table implementation was visible to its module but invisible to the rest of the system. The publication of this seminal paper represented a transition from software engineering research to concept formulation to programming language design. The module structures that were eventually developed by others differed in how the interface and implementation were managed, whether the interfaces were type checked, and the visibility rules for accessing the entities of a module.

A module enables more structure than a procedure. Modules allow programmers to group related items together, including types, data, and routines. Modula-2 was one of the first languages that allowed the separation of the interface from the body of a module [Wirth 1977]. In Modula-2, the interface and body can be separate in the same file or even appear in different source files.

In the mid-eighties, modules were refined to software components to enable reuse. Software reuse was seen as possibly "increasing software productivity by an order of magnitude or more" by Horowitz and Munson [1984]. Their notion of software reusability was broad, including not only code reuse, but also the reuse of software design. In their discussion of reusable code, they cite McIlroy's article from the 1968 NATO conference that described factories for building software components [McIlroy 1976]. At the time, this idea was considered impossible to achieve because of a lack of tools to create, index, search and compose components efficiently; also, the ability to describe the semantics of a component to a user so that the user could assess its utility to the task at hand was limited. After reviewing several approaches that offered software reuse through high-level systems that produce programs from specifications and user-supplied parameters, Horowitz and Munson praised Ada as a language with features to support software reuse, including the Ada package (i.e., an abstract data type) and the generic procedure [Horowitz and Munson 1984]. Thus, Horowitz and Munson believed that the programming language design community was providing tools to accomplish the software engineering research goal of software reuse. The designs of Ada and the research language CLU, both demonstrated an emphasis on modularity and the ability to reuse code. CLU was intended for programming in the large, emphasizing program readability and understandability over ease of writing [Liskov 1993]. Liskov believed that it was possible to develop a programming methodology through designing a programming language so that problem solutions were similar to programs in that language. This principle guided the design of CLU, which emphasized data abstraction and encapsulation as well as reliability using exceptions. Liskov stated that a driving force behind her language design was the desire to write correct programs, using language constructs to either help avoid errors or help find them automatically. She described many design decisions in terms of a goal of program safety, including the decision on compile-time type checking within a module (and bounds type checking for arrays) [Liskov 1993].

Ada was one of the first widely used languages to feature separation of specification and implementation as part of the language design, which enabled separate compilation of modules, and still provided checking of the interfaces. Ada was designed specifically "to support development of large programs composed of reusable software components" [Wegner 1984]. Ada separated the interface and implementation through its package module [DOD 1980]. An Ada package can contain procedures, functions, types and variables. The specification of a package is through its interface, which is declared separately from its implementation.

An important component of the module concept is the visibility rules for the interface and implementation. The early languages (e.g., Pascal, Algol 60) used textual scope to define the interfaces between modules (e.g., subroutines, procedures). Wulf and Shaw detailed the problems with using scope for interfaces. They enumerated some desirable properties of an interface, including (1) the scope of a name should not automatically be extended to inner blocks, (2) access rights to a structure and its substructures should be decoupled, and (3) it should be possible to distinguish different types of access [Wulf and Shaw 1973]. As mentioned previously, Modula-2 [Wirth 1971a] contains the idea of separating the interface and implementation; the implementation is not visible to users. However, there is no way to hide entities in the interface. The package specification in Ada does enable parts of the interface to be hidden, as well as the implementation. Ada packages also allow information to be left out of the declaration and provided in a separate file not visible to users of the abstraction [DOD 1980]. However, Ada allows the header of a package to be divided into public and private parts. In Ada, data types expose the type name and operations allowed on the type. The data structures used and the implementation of operators are not visible to users. Thus, the ideas of information hiding were eventually implemented in programming languages by controlling access through the interface, including the introduction of access rights such as *private, public,* and *protected*. The access rights grew out of data abstraction research, which is discussed in the following section.

3.1.3 *Data Abstraction.* The concept of *data abstraction*, or *abstract data type*, has had a profound influence on both software development and programming language design. The major thrust of programming languages and software engineering research activity in the 1970s was to explore issues related to data abstraction and visibility control [Ghezzi and Jazayeri 1998].

Abstract data types give programmers the capability of safely defining their own types. Modern day concepts of abstract data types include (1) encapsulating or enclosing data and the related operations on the data, (2) naming the data type, (3) placing restrictions on the use of the operators, (4) having rules that specify the visibility of the data, and (5) separating the specification from the implementation.

The notion of data abstraction was first introduced in the Simula 67 language by associating abstract operations with the entities for which they were defined [Dahl and Nygaard 1967; Birtwistle 1973]. The entities and operations were encapsulated in a class structure. However, accessing the data entities in a class

was not restricted to the class operations. It should be noted that the creators of Simula 67 did not think about restricting the use of class entities [Rowe 1980]. The idea of a class in Simula 67 paved the way for the later development of data abstraction, classes and object-oriented programming. For various reasons, including, (1) the price of the compiler, (2) the perception that Simula 67 was a simulation language, and (3) the lack of resources to publicize it [Nygaard and Dahl 1978], Simula 67 did not receive the attention it rightly deserved until much later. Finally, in 2001, Ole-Johan Dahl and Kristen Nygaard, the developers of Simula 67, received the Turing award for ideas fundamental to the emergence of object-oriented programming, through their design of Simula 67. Unfortunately, they both died in the same year and were not able to personally receive the award.

The idea of using abstract data types as an abstraction technique for reliable software design was presented by Liskov [1972]. In this paper, Liskov describes various types of abstractions that provide the infrastructure to produce reliable software that is easier to prove and/or test. One of the ideas presented was to develop an abstraction based on the use and structure of data and limiting the visibility of functions in the abstraction to what they need to know. Thus information hiding was also included as part of abstraction concept. The data for the functions constituted the resources of the abstraction.

In the 1970s, the software engineering community recognized the need to organize programs into modules by localizing implementation details as much as possible. This led to the development of language support for abstract data types. The first research to incorporate data abstraction into a language was CLU, through its cluster data type [Liskov and Zilles 1974].

About that time, a paper by Gannon and Horning [1975] presented language design issues to improve the reliability of programs. One concept that had an impact on the design of languages was the idea of an explicit interface, and providing rights to the creator of objects and the users of objects. The encapsulation mechanisms of Simula 67 were refined in the 1970s by the developers of CLU [Liskov and Zilles 1975; Liskov et al. 1977] and Alphard [Wulf et al. 1976; Shaw 1981]. CLU and Alphard both provided a data abstraction construct that enforced protection of the data entities, allowing access only to the operations defined for them. The design of CLU [Liskov et al. 1977] was very much driven by the desire to support software engineering methodology developed by means of program decomposition, similar to step-wise and structured programming [Wirth 1971a; Dijkstra 1969]. A goal for Alphard was to reduce the cost and increase the quality of software by developing a program methodology based on abstraction and verification [Shaw 1981].

Besides CLU and Alphard, Euclid [Popek et al. 1977] also introduced concepts of information hiding. In both CLU and Euclid, the declaration and definitions of a data abstraction always appear together with the header that states which of the module's names are to be exported. In CLU, a module (called a *cluster*) implements a single abstract type [Liskov et al. 1977]. It has an interface specification, which is the code to create instances and definitions of operators. Alphard also has the concept of operations to be performed on objects of the cluster type. Assignment and equality are default operations on "forms." The

language has explicit rules about accessing a variable of a cluster in the immediately enclosing procedure. The environment, or outer scope, was visible only by a direct request.

A set of visibility rules for data abstractions was also an important component of research in the early 1980s. *C++*, developed by Bjarne Stroustrup, had as one of its primary goals the support of data abstraction and object-oriented programming [Stroustrup 1987]. Not only did it provide the encapsulation but also limited the use of data entities. It used the model of *private*/*public* to limit the access of entities in classes. The access right *protected* was introduced for inheritance in object-oriented languages.

Cardelli and Wegner defined object-oriented languages by three necessary characteristics: (i) they contain objects that are data abstractions with a defined interface and hidden state, (ii) objects have types, and (iii) attributes can be inherited from super-types [Cardelli and Wegner 1985]. This definition clearly shows the influence of data abstraction and modularity (i.e., defined interface) on this programming language paradigm. They cite Smalltalk as an example of object-orientation with inheritance, and described the standard Smalltalk hierarchy as an example of reuse that imposes a coherent structure on the types in the system. *C++* and Java added visibility control to the notions of inheritance and attributes (i.e., fields) already included in Smalltalk.

The abstract data type research results entered mainstream programming languages in combination with research results on inheritance in object-oriented software design and implementation. The goal of classes and inheritance is to enable reuse as well as code and data sharing. Inheritance allows new abstractions to be defined as refinements or extensions to existing ones. Dynamic binding allows a new version of an abstraction to display dynamically defined behavior.

Data abstraction plus inheritance and dynamic binding define important characteristics of classes and object-oriented design. All three fundamental concepts of object-oriented programming—encapsulation, inheritance and dynamic binding—have their roots in Simula 67.

The Simula 67 designers observed from their simulation applications that processes often share a number of properties, including both data and actions [Nygaard and Dahl 1978]. These processes were structurally different in some respects and therefore needed their own declarations—parameterization could not provide the flexibility needed. They then came up with the idea of subclassing, another name for inheritance. The term *inheritance* was defined later.[3] In the Simula 67 language, subclassing was done by prefixing. The conceptual view of prefixing is concatenation of the parent's class declarations with the child's. Thus, if class A is a prefix of class B, B's objects would include A's formal parameters, specification of the formal parameters, class body declarations, and the class body as well as its own declarations. Thus, not only were the data and

[3]Collins and Quillian defined *is-a* relationships, introducing them in semantic hierarchies in 1969 [Collins and Quillian 1969]. In Smalltalk-76 [Ingalls 1978] subclasses are described as a useful mechanism to accomplish code inheritance and allow code reuse and specialization through overriding.

functions of the class inherited but so was the body of the class, which contained statements that were executed when the class was created.

Simula 67 also had dynamic binding through its *virtual* concept. Thus, the method actually used depended on the object's subclass. Smalltalk also incorporated the notion of object orientation with binding [Ghezzi and Jazayeri 1998; Goldberg and Robinson 1983] at run time. In the Smalltalk message-based programming model, each object carries a type at run time.

With the introduction of inheritance, object-oriented languages needed to supplement the scope rules of objects. Ada, *C++*, and Java all have rules for visibility of inherited attributes. Visibility issues included the visibility of private and public members to derived classes.

The issue of multiple or single inheritance has been addressed in both programming language design and software engineering. Subclassing in Simula 67 was designed to be single inheritance; that is, an object of a class can only have one base class. *C++* initially had single inheritance because this was sufficient in a large majority of cases [Stroustrup 1987]; however, Stroustrup indicated that there were important concepts for which multiple inheritance is required. These concepts required a directed acyclic graph to represent their relationships rather than a tree, the single inheritance structure [Stroustrup 1987]. Thus, *C++* introduced multiple inheritance, in which a class can inherit from a number of base classes. However the controversy of single versus multiple inheritance did not end. Cargill [1993] argued against multiple inheritance by indicating that is it complicated to learn, write and read. He demonstrated, using examples of programs with multiple inheritance in the literature, that multiple inheritance is either improperly used or not used at all. Waldo [1991], in the same journal, makes a case for multiple inheritance by describing different forms of multiple inheritance and presenting examples where it is useful. When Java was developed, it used single inheritance with the extra facility of *interfaces*, which can be viewed as a form of multiple inheritance. This type of inheritance is called mix-in inheritance [Scott 2000].

A time line graph of the interactions among abstractions and programming languages is given in Figure 1. The time line indicates when a particular abstraction concept first appeared in the literature or in a programming language, subsequent interactions with other software engineering concepts, and its final incorporation into object-oriented programming languages.

### 3.2 Types, Polymorphism, and Generics

Strong typing is currently accepted as a fundamental feature of modern programming languages, regardless of whether validated by compile-time checking, type inference or run-time checking. According to Cardelli and Wegner [1985], in a strongly typed programming language "all expressions are guaranteed to be type consistent although the type itself may be statically unknown." In other words, a program in a strongly typed programming language is guaranteed, through a combination of compile-time and/or run-time type checks, to be *type safe*; this means that the program is free from type errors during execution. Strong typing ensures the reliability of code and provides an additional

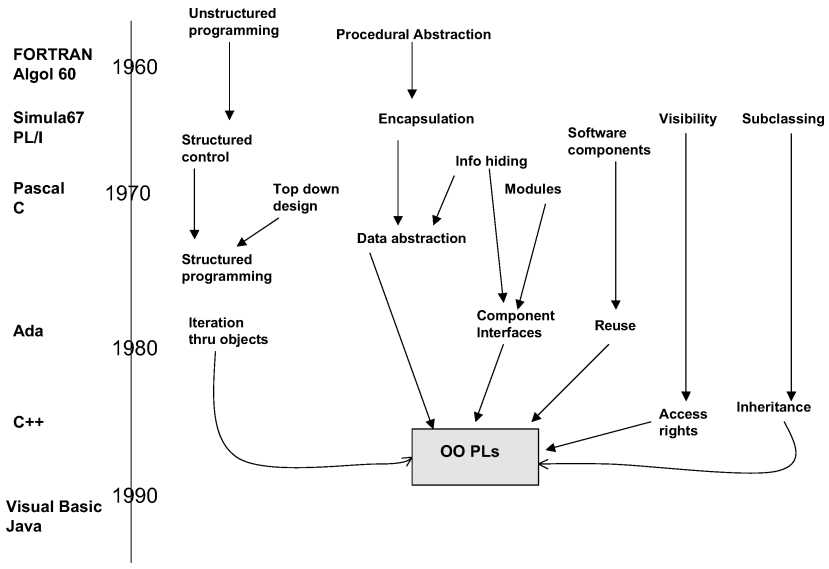


Fig. 1. Time line for data abstraction.

layer of semantic checking of a program. Development of strong typing in programming languages happened cotemporally with software engineering research on reliability and later, software reuse. It seems clear that the concepts of strong typing and software reliability reinforced each other, especially since the early pioneers in imperative language design were also active in software engineering research. Further, as type systems for programming languages developed notions of genericity and polymorphism, these were directly related to issues of software reuse.

The main imperative languages designed in the 1960s and 1970s—Algol 60, Pascal, and PL/I—were designed at a time when the software engineering community was worried about the reliability of code. In the mid-1970s, Barry Boehm defined *software engineering* to be "the practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate and maintain them" [Boehm 1976]. Peter Wegner [1984] talked of the 1950s as a time of stand-alone programs and the 1960s as a time of development of operating systems and databases. He stated that the 1970s saw the birth of software engineering, referring to similarities in the construction of large software systems and large physical structures such as bridges. The 1980s, in Wegner's view, saw the development of interface technologies and the personal computer revolution. The 1990s was a time of knowledge engineering, the use of intelligent components to build systems (e.g., adaptation). In this temporal framework, the 1970s—the era of Algol 60 and Pascal—was a key time for software engineering, so that the design of strongly typed languages, which were type safe, coincided with the beginning of the discipline of engineering software.

C.A.R. Hoare, a member of the *IFIP Working Group in Algol (WG 2.1)* active in the original design of Algol 68 [Bergin and Gibson 1996], was the keynote

speaker at the first Principles of Programming Languages (POPL) conference in Boston in 1973. In that talk he spoke of types as "eliminating low-level programming errors when checked by compiler" [Hoare 1974a; van Wijngaarden et al. 1968]. Furthermore he claimed that types are essential to program design and documentation. He stated that denoting a declared name and structure or range of values for every variable is the "most important aspect of clear programming" [Hoare 1974a].

In 1977, John Gannon performed an interesting experiment showing that declared types and type checking could increase the fraction of programming errors caught before production of the final program [Gannon 1977]. In addition, the programmers he studied who were the least experienced, benefited the most from compile-time typing. In an earlier paper, Gannon and Horning [1975] had explored issues of language design for reliability through experiments with students. They stated that declared types provided useful "redundancy" that caused errors to transform valid programs into invalid programs; they also noted that type checking caught logical errors, not syntactic errors.

The first widely used programming language to feature explicitly typed variables was Algol 60. In designing Pascal, which was based on Algol 60, N. Wirth intended the programming language "to make available a language suitable to teach programming as a systematic discipline" and "to develop implementations of the language that were both reliable and efficient on presently available computers" [Wirth 1971b]. It was the first language to introduce user-defined types, including records and subranges. However, Pascal's subrange types caused some problems in its type system, since operations on subranges could not be checked fully at compile time. In addition, Pascal extended the data structure constructs in Algol 60 (e.g., arrays) by introducing record structures with heterogeneously typed fields. Pascal allowed variants of records in which the value of a tag field determined the current type of the variant; however, most Pascal compilers offered no run-time type check of these tags, so the use of variant records was *type unsafe*. Pascal also introduced the use of heap storage with pointer-valued variables that had heap addresses as their values.

Habermann [1973] criticized the subrange construct because there can be expressions whose operands are subrange type values, but whose result has no type! Habermann also argued that structures are not types, primarily because operations on structures are actually operations on their constituent elements, not on the structure itself; therefore, a characteristic property of a type, that is, being a value as well as operations on that value, is not fulfilled [Habermann 1973]. These critiques of types in Pascal can been seen as addressing issues of program understandability and readability as well as good programming language design.

The unsatisfying lack of orthogonality in design was clearly addressed by Algol 68 [Tanenbaum 1976; van Wijngaarden et al. 1968], which offered a rich system of user-defined types. The fact that all types could be used in all contexts rendered Algol 68 easier to learn, although some argued that Algol 68 programs were difficult to understand. The designers of Algol 68 achieved their goal of being able to validate type safety of programs fully at compile-time, even while allowing constrained type unions, which safely offered the functionality

of the variant records in Pascal. Unfortunately, Algol 68, although novel, never attained popularity or general usage outside of the research world.

Cardelli and Wegner [1985] stressed that a type "protects an underlying untyped representation from arbitrary or unintended use." They differentiated the goal of *static typing* (i.e., complete type checking at compile-time) used by the early programming languages from the goal of *strong typing* combining compile-time and run-time type checking to achieve type safety. They characterized these early programming languages with declared types in terms of their ability to achieve the goal of strong typing. Some of the problems with array types and variant records in Pascal have been mentioned previously; in addition, Pascal failed to require the complete type of procedures passed as parameters to be specified.

During the late 1960s and 1970s, the emphasis on data abstraction in languages such as Simula 67 and Modula-2, enabled richer notions of types including inheritance (in Simula 67, already discussed in Section 3.1.3), modules with opaque types at the interfaces to facilitate information hiding (in Modula-2), and parametric polymorphism (in ML) [Cardelli and Wegner 1985]. During this same period, the influential *International Conference on Reliable Software* was held in 1975[4] and discussions on types were prevalent.

In 1980, there was a workshop on *Data Abstraction, Databases, and Conceptual Modeling* attended by researchers from software engineering, databases, and artificial intelligence [Rowe et al. 1980]. The transcript of a panel considering the topic *What is a type?* from the viewpoint of each of these research communities, features wide-ranging discussions of this question. A central theme was the value and practical viability of abstract data types, as a construct with separate specification and implementation, and with encapsulation of values. Panelist Mary Shaw's remarks emphasized the needs for types to convey information about constraints on values and how they can be used. Ira Goldstein, another panelist, explained the concept of a class with methods from Smalltalk, (in which he was programming at the time), a world of hierarchical types. Ted Codd, a panelist, associated a notion of type with ideas in relational databases. All of these researchers agreed that a notion of type adds structure and definition to the use of values in their respective subfields of computer science.

The emphasis on building software from components in the software engineering community during the 1970s, was reflected in programming language research in data abstraction, object-oriented programming and modularity [Wegner 1984].

At this same time, the designers of Smalltalk were providing a practical demonstration of how abstract data types with encapsulation might be integrated into a programming language. Smalltalk uses run-time type checking (type by use). Some research projects defined compile-time checkable types for Smalltalk [Graver and Johnson 1990]. This goal could be accomplished only by separating subclassing from subtyping rather than considering them as equivalent and thereby restricting inheritance. These research results were

[4]The conference was subsequently held annually from 1976–1979, demonstrating the emphasis on reliability in the research community at that time.

not incorporated in the definition of Smalltalk, but may have affected the definition of inheritance in *C++* and Java, and the addition of strong typing to these programming languages.

Ada used strong typing to check consistency of definition and usage of parameters in packages. However, Ada components (or packages) had some weaknesses as well. Wegner [1984] describes how Ada components can communicate not only through interfaces, but also through global variables and pointers into the heap. In addition, packages and processes in Ada did not integrate well; for example, Ada packages were not protected against concurrent access by tasks. These aspects of Ada hinder it as a basis for software component technology, according to Wegner. He discussed software reuse in terms of how it relates to software productivity; most significant are reuse of software components in different applications and reuse of components in different versions of a given program [Wegner 1984].

Both Ada and CLU offered generic templates for data abstractions, parameterized by the type of the data. These embodied the notion of generating specific software components of the sort needed for the application, while only maintaining one copy of the code. These languages offered *parametric polymorphism* in that an explicit type parameter was used to instantiate the function argument; this kind of polymorphism was described by Cardelli and Wegner [1985] as "macro-expansion driven by the type of the arguments." Wegner remarked that the emphasis on reuse embodied in Ada packages, also implied an emphasis on portability (i.e., reuse of software across machines), modularity (i.e., reuse of software across programs), and maintainability (i.e., reuse of unchanged portions of code after other portions are changed) [Wegner 1984]. Frankel [1993] discussed how Ada generics "encourage and assist the implementation of reusable software." He cited Ada's strong typing and separation of package specification from implementation as language features supporting abstraction, and necessary for reuse to be accomplished in an efficient manner. Frankel emphasized that building a generic version of code isolates the reusable portion from the part that is unique to each user. Both of the kinds of reusability he discussed—*as-is* and *with-modification*—can be realized with generics.

Research in the programming language design community at this time examined the relation between subtyping and subclassing, and their influence on possible choices for object-oriented programming language design [Madsen et al. 1990]. Madsen et al. contrasted the two uses of subclassing, for establishing a type system through specialization and for code sharing. They discussed a type system as a "means for detecting certain program errors, type errors" and considered combinations of compile-time and run-time type checking to achieve type safety, also including the checking needs of parameterized types. They expected a type system to allow desirable "early error reporting" when compile-time checking is possible. This paper is an example of many research papers in the programming language design community of the time, concerned with building type systems that would enhance program reliability in the new language paradigm.

The emergence of object-oriented programming languages in the 1980s and 1990s was cotemporal with increasing concern in the software engineering

community about how to design and build large, modular systems from pieces with clearly defined interfaces using teams of programmers. The ability to ensure software reliability was questioned and more emphasis was placed on reuse of tested or provably correct software.

Designed in the mid-1980s, *C++* was the first widely used object-oriented language for commercial software. *C++* offered the creator of a new type (or class) the ability to define implicit type conversions between that type and another; these are used by the compiler for resolving types in uses where coercion is necessary. Murray [1988] discussed problems with this language feature that can occur when too many conversions are defined, leading to ambiguities that require casting to fix. In addition, this feature may make it possible for new declarations to break existing code. This discussion was in the context of *C++* libraries, whose usage was growing at the time. Translating these concerns to software engineering terms, as a programming language expert Murray was worrying about effective software reuse, support for program understanding, and code extensibility.

Wegner [1984] viewed inheritance in classes as structuring components so that they are easily reused. He compared libraries in Ada, which are flat in structure, with libraries in Smalltalk, which have a tree structure. He suggested the idea that object-oriented language mechanisms can embody semantic knowledge that can aid in software reuse.

Reuse research in the late 1980s focused on libraries, reusable components, and reusable support environments [Prieto-Diaz 1993]. There are many different types of reuse considered—ideas, artifacts, processes—but the connection with programming language design is best viewed from the perspective of *ad hoc* and *compositional* reuse. Ad hoc reuse involves reuse of libraries with user-friendly retrieval mechanisms; compositional reuse refers to composing existing components to furnish parts of new systems. Both of these involve reuse of code, with (black box), or without (white box), modification [Prieto-Diaz 1993].

The standardized interfaces of the *C++* STL and the Java JDK can be seen as providing tools for these sorts of reuse. Stroustrup stated that *C++* templates were intended to be added to the language when their design and implementation issues were fully explored. The need for standardized libraries in *C++* was clear in 1988 [Stroustrup 1993] when he first presented templates at the USENIX *C++* conference. He chose to use generics because they provided static type checking and cited the provision of good libraries as desirable, and almost necessary, for acceptance of an object-oriented programming language. Looking at *Dr. Dobb's Journal* and the *C/C++ Users Journal* from the 1990s, we find several articles about programming with the standard template library (STL), evidence of language design responding to reuse issues [Plauger 1995; Keffer 1995]. The addition of generics to Java 1.5 and the strong interest expressed in the generics offered by *C++* are further evidence that a concern with reuse issues affected programming language design.

Strong typing was added to programming languages to enhance software reliability, at a time when this was a focus of the software engineering research community. The development of user-defined packages and generics were direct responses to concerns about software reuse. The time line graph showing

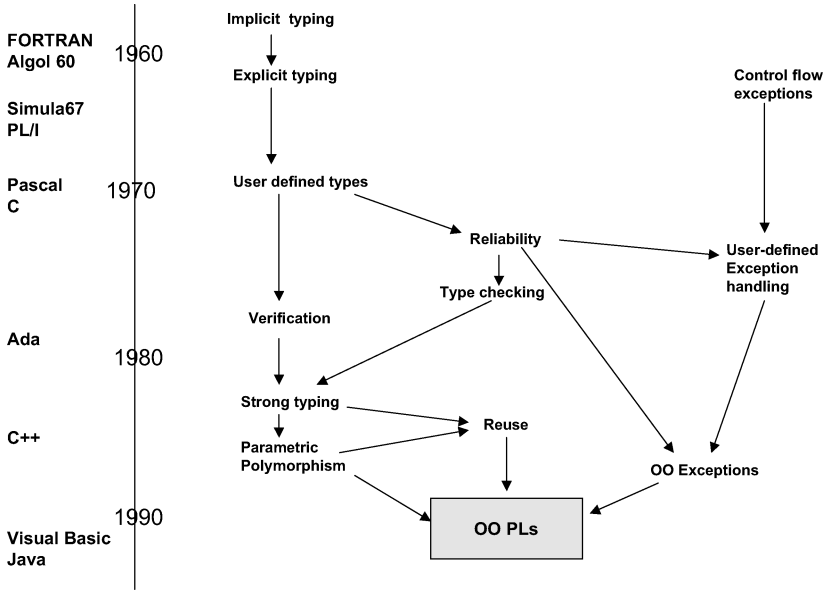


Fig. 2. Time line for Types and Exceptions. The arrows in the diagram on the right show relations between programming language and software engineering concepts, and the times of their first appearance. The programming languages listed to the left of the time line are shown at their inception year. Some of the concepts on the right appear in some of the programming languages on the left.

the interactions between types in programming languages and the software engineering research related to types is given in Figure 2.

### 3.3 Exceptions

*Exceptions* are features that were added to programming languages to provide the programmer the capability to specify what should happen when unusual execution conditions occur, albeit infrequently. Originally in programming languages, such atypical conditions resulted in control being relinquished to the operating system that then aborted the execution of the program in a *forced termination*. But when such conditions, however infrequent, can be anticipated, the programmer can write code to react to them and to gracefully *handle* them. Exceptions and exception handler codes are the mechanism provided by modern programming languages to address this problem.

The historic introduction and development of exception mechanisms in programming languages is intertwined with considerations of software reliability and fault-finding in software engineering. The first precursor of an exception-like construct is found in Lisp 1.5, a language designed by John McCarthy in the mid-1950s [Gabriel personal communication; McCarthy et al. 1965]. This language featured a function named `errset` that allowed the Lisp interpreter and compiler to gracefully exit from an error when one occurred. This function used a mechanism for counting the number of *cons* operations performed within a loop in order to stop an unbounded computation. The `errset` construct

allowed suppression of error statements and a program restart under certain conditions after an error occurred [McCarthy et al. 1965].

PL/I, a programming language developed by IBM and representatives of its user community (i.e., SHARE [Radin 1981]) concurrently with System 360 in the mid-1960s [Radin 1981], included some facilities for dealing with control flow that were atypical for a high-level language. Previous languages had had few facilities for dealing with exceptional conditions during execution, such as *end of file, overflow, bad data,* and so forth. The PL/I language designers wanted to give programmers the ability to write reliable and safe programs totally in their language [Radin 1981]. The *ON condition* feature was introduced into PL/I to allow specification of the actions to be taken when one of a set of 23 unusual, but anticipatable situations occurred during execution. User-defined ON conditions were also allowed. There were several problems with this mechanism: (i) an ON unit was dynamically associated with its invocation by an exceptional condition occurrence, rather than being associated lexically with the excepting statement or operation, and (ii) global variables were often used to communicate data to the ON unit code [Library 1970; MacLaren 1977; Liskov and Snyder 1979].[5] The construct proved difficult to use, in part because the fix-up actions were to take place in the state that pertained when the ON statement was executed. Nevertheless, the philosophy of programming language design for reliability demanded that this facility (or something like it) be included in the programming language definition.

By the mid-1970s, software reliability was a strong concern in both the software engineering and programming languages communities. In March 1977, SIGPLAN, SIGOPS and SIGSOFT cosponsored the *Conference on Language Design for Reliable Software* in Raleigh, NC. The *Communications of the ACM* featured a special issue on language design for reliable software in August 1977.

The exception mechanisms in today's languages were influenced by research in software engineering. In the *Communications of the ACM* in December 1975, John B. Goodenough discussed issues in exception handling, classifying the types of exception usages as (i) dealing with domain or range failure of an operation (ii) indicating the significance of a result, or (iii) permitting monitoring of an operation. When range failure is indicated, the operation may need to be aborted, retried, or terminated, yielding partial results. A domain failure requires modification of the input or an abort of the operation. Exception type (ii) is not a range failure, but requires that additional information be passed back to the user about the operation. Exception type (iii) is usually resumed after the user examines the information about the operation passed back by the exception. Not all of these are error conditions; Goodenough [1975a] described them "as a means of conveniently interleaving actions belonging to different levels of abstraction." After reviewing some existing exception mechanisms of the time, Goodenough discussed requirements for good exception handling, which should "help prevent and detect programmer errors" according to Gannon and

[5]Note: no parameters were allowed in ON conditions; standard condition built-in functions were provided to query information such as the name of the procedure in which the condition was experienced [Library 1970]. See also http://www.kednos.com/pli/docs/USERS_GUIDE/6292pro_023.html.

Horning [1975]. He argued the effectiveness of lexical (or static) association of handlers with operations that may throw exceptions. These included both language-defined and user-defined (subroutines) operations, with explicit declaration of those exceptions that may be thrown as the result of a call. He advocated compile-time checking of the completeness of exception handling. His model of exception handling included the possibility of resuming execution as well as termination, and allowed the use of default exception handlers.

Goodenough's [1975a] paper codified the ideas presented in his earlier paper [Goodenough 1975b] at the second *Symposium on Principles of Programming Languages (POPL)* (January 1975), the major programming languages research conference at that time. This is an example of how software engineering researchers, concerned about issues of software reliability, directly contributed to programming language design.

In addition to Goodenough's contributions to exception handing, during this same year Brian Randell described his own construct for error detection and recovery [Randell 1975] in a paper delivered at the *International Conference on Reliable Software*. He defined a structured mechanism called *recovery blocks* which were to be used when an unanticipated fault occurred. Horning described exception handlers as "useful" and "intended to cope with particular anticipated, but unusual, situations" [Horning 1979]. Horning suggested that recovery blocks and exceptions could coexist in the same code as what he termed "an attractive compromise."

During the period of these software engineering discussions on how programming languages should be designed to facilitate the handling of faults, a new programming language, *CLU* [Liskov and Zilles 1975; Liskov et al. 1977; Liskov 1993] was designed based on the ideas of abstraction and specification. CLU was designed to enable good program construction of moderate-sized (by modern standards) codes, emphasizing the use of software engineering techniques such as data abstraction and program verification, as well as programming language theory in terms of formal (algebraic) specifications. In the chapter on exceptions, Liskov and Guttag [1986] described the need to "program defensively," that is to write each procedure "to defend itself against errors." A "robust program" is "one that continues to behave reasonably even in the presence of errors" [Liskov and Guttag 1986]. This is a strong emphasis on this aspect of programming language design being motivated by software engineering considerations; this fits well with Goodenough's [1975a] emphases.

Liskov and Snyder [1979] discussed the design of the exception handling mechanism in CLU and gave credit to Goodenough specifically for influencing them through his paper [Goodenough 1975a]. They discussed their design with regard to other previous approaches, as well. For example, the CLU model expects that exceptions not handled locally where they are raised, will be handled by the immediate caller of a procedure. (Of course, the handler code can rethrow an exception to pass it upwards through the call chain, one link at a time.) Liskov and Snyder pointed out that this is consistent with Goodenough's model, but not with the definition of exception handling in PL/I or Mesa. Their discussion of the *resumption* versus *termination model* of exception handling, explicitly referenced Goodenough's model in his 1975 paper [Liskov and Snyder

1979]. Thus, it is clear that for CLU, programming language design was directly influenced by software engineering research, although the researcher (Goodenough) presented his ideas to both communities [Goodenough 1975a, 1975b].

The Ada programming language was defined in the late 1970s according to specifications set by the US Department of Defense, as a modern programming language that was to be the universal software *lingua franca* for their projects. The *Rationale for the Design of the Ada Programming Language* [Ichbiah et al. 1979] described the evolution of the Green candidate language written as part of the process of defining what was to become standardized as Ada. The *Rationale* was written to record design decisions and influences on the Ada language. The chapter on exception handling in the *Rationale* references the earlier Goodenough article [Goodenough 1975a] in two ways. It uses Goodenough's classification of exceptions as either allowing program execution termination or resumption after handling, but rejects the resumption model of handling in favor of always terminating execution.

The *Rationale* also references a technical memo by Bron et al. [1975] that discussed desirable properties of an error-handling mechanism to provide for program termination under exceptional conditions. These desirable properties were formulated so that operational semantics could be defined for those programs requiring this construct. The suggested programmer-controlled termination mechanism was associated with a block of operations; it allowed programs to be written with the usual input assertions, with the handling of bad input not affecting program structure, and resulting in a cost only to those blocks where it might be used. These ideas influenced the design of Ada exception handling.

The *Rationale* also refers to the *Bliss* language [Wulf et al. 1971] developed at DEC in the mid-1970s as influential in the definition of Ada exceptions. The Rationale emphasizes that the designers wanted to being able to prove the correctness of programs [Luckham and Polak 1980] and to optimize programs with exceptions. These properties led to the rejection of the resumption model, which renders both of these difficult.

Perry in his *ICSE'89* award-winning paper entitled *The Inscape Environment* discussed the specification and design of exception handling as an integral part of system development when large systems are built by many developers. His model for specifying exceptions is based on an extension of Hoare's input/output predicates [Hoare 1969; Perry 1989]. Perry examined the Larch specification language [Guttag et al. 1985], but preferred a nonalgebraic approach. It is clear from his paper that Perry was influenced by programming language technology in his choice of how to specify exceptions and their handling in Inscape, whose goal was to provide an integrated software development environment for large groups of developers building large software systems. He described a design in which module interface specifications include descriptions of both exceptions to be handled and handling strategies. His program construction tool checked that exceptions are handled as specified in the constructed code. Changes in exception handling are considered by the change evolution manager component of his system. The Inscape environment shows that the treatment of exceptions in programming languages in the late 1980s influenced

software engineering research on the design of programming development environments.

The subject of exception handling and how it should be added to *C++* was a major topic of debate at the 1990 *USENIX C++ Conference*. Koenig and Stroustrup [1993] argued for their model of exceptions as objects (which was eventually incorporated in *C++*) in their 1990 *USENIX C++ Conference* paper. They also referred to the termination model of PL/I, CLU and Modula-3 [Harbison 1992] as preferable to any resumption model. Essentially, *C++* exception handling seems an outgrowth of the techniques defined in CLU and Modula-3. Statements are executed within a *try* block, which has associated *catch* blocks (typed exception handlers) that are handlers for some of the exceptions that can be thrown from within the *try* block. This design was influenced by the work on fault-tolerant systems by B. Randall [Ellis and Stroustrup 1990]. Again we see programming language design being influenced by software engineering research. Exceptions are handled locally or by walking up the call chain to find the first appropriately typed handler. The set of exceptions possibly thrown by a function (directly or indirectly) can be listed as part of the function declaration. Violations of this exception specification are dealt with at run-time, not compile time as in Java. This decision to avoid compile-time checking was in part due to the ability to link to C, functions that have no explicit exception constructs. When a function with an exception-specification throws an exception not on its list, then the function `void unexpected()` is called and execution is usually halted.

More recently, aspect-oriented programming describes how crosscutting concerns in an object-oriented program can be addressed by new compositional mechanisms in addition to inheritance. At *ICSE 2000*, this new paradigm was used by having aspects [Kiczales et al. 1997] express the detection and handling of exceptions [Lippert and Lopes 2000]. The main idea was to reduce the amount of redundant handling code in a program. The specific language used, *AspectJ*, allowed abstract crosscuts (i.e., templates) that can be instantiated in many different locations where exception handling required the same actions. In their case study using a large Java application containing 750 classes (including 150 test classes), the authors reduced the size of the exception handling code from 10.9% of the total lines of code to 2.9% lines of code on average. This represented a significant reduction in *catch* statements over the original program. This research paper is indicative of the strong interaction between software engineering and programming language research and researchers. In this case, a paper describing how to code exceptions (a programming language mechanism) to ensure program reliability (a software engineering desiderata) through the use of aspects (a programming language mechanism) was presented at the premier annual software engineering conference.

As another example of the close tie between the disciplines, Robillard and Murphy [2000] discussed in their paper why the design of exception handling in an application is so difficult. The focus is software design, namely how to regularize the exceptions that are passed between components of a software system, but intertwined in the discussion is the essential character of exceptions and their handling in Java. The technique applied is *software compartmenting*,

first described by Litke [1990] at the *TRI-Ada 1990 Conference*. This technique divides software into compartments, defines precise and complete exception interfaces for each compartment, and automatically verifies the conformance of the actual program to compartment specification [Robillard and Murphy 2000]. The paper described a case study of software compartmenting using the authors' own Java tool as data. The authors also developed guidelines for Java exception usage. In their discussion, they referred to exception handling both in CLU and *C++*. This is another software engineering research project that builds on programming language design and research.

Figure 2 provides a time line for concepts related to types and exceptions.

### 3.4 Concurrency

The theoretical foundation of the concurrency that is expressed in programming languages today was laid in the 1960s and early 1970s. Initially, the need for concurrency in programming languages was driven by applications, such as simulation in Simula 67 [Dahl and Nygaard 1967]. The initial concurrent construct that appeared in Simula 67 was a *coroutine*, which allowed a quasi-parallel execution of the program. In 1965, Dijkstra published an important paper in which he described how to use concurrency to express the solution to the problem of sharing data among concurrent processes. In 1968, Dijkstra [1968a] showed how to use semaphores to solve a variety of synchronization problems and introduced the famous *dining philosophers* problem.

At the same time, Algol 68 [van Wijngaarden et al. 1968] was the first programming language to allow expression of true concurrency by the programmer through the *parbegin-end* construct. Later, [Courtois et al. 1971] introduced the readers/writers problem and formulated solutions to it using semaphores.

During the 1970s, most concurrency was expressed at the operating system level, through time-sharing operating systems with multiprogramming. In his HOPL-II article on *Concurrent Pascal*, Brinch Hansen relates how "the idea of *monitors* evolved through discussions and communications among E. J. Dijkstra, C.A.R. Hoare, and me (Brinch Hansen) during the summer and fall of 1971" [Hansen 1996]. Influenced by the work of Dahl in Simula 67, the concurrency model of Concurrent Pascal, developed in the mid-1970s, refined the definition of monitors into a key language construct [Hansen 1972, 1975]. Brinch Hansen describes a monitor as "a set of shared procedures which can delay and activate individual processes and perform operations on shared data" [Hansen 1996]. Conceptually, monitors can be viewed as an extension of the idea of an abstract data type to a concurrency setting.

C.A.R. Hoare later described his implementation of *monitors* [Hoare 1974b] built with semaphores, for shared memory concurrency models in operating systems. According to Brinch Hansen, Hoare's ideas were available in public presentations in 1972 and 1973. Thus, these two researchers codeveloped the monitor concept. Later, in his seminal paper in 1978, Hoare described notational tools for programming languages to express concurrency, that is, *communicating sequential processes* (CSP) for message-passing concurrency models [Hoare 1978]. Clearly, there was close interaction between programming language

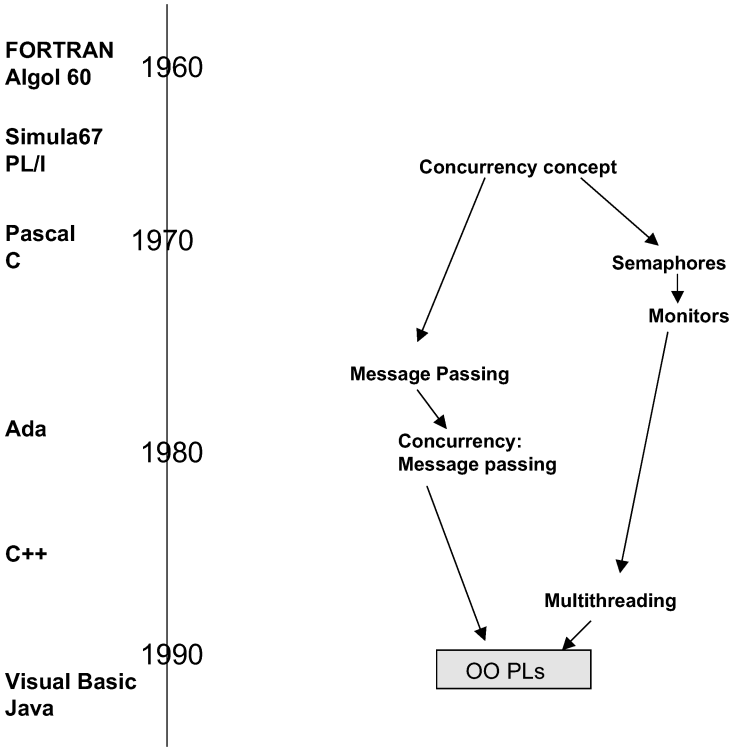


Fig. 3. Time line for Concurrency. The arrows in the diagram on the right show relations between constructs related to concurrency, with the times of their first appearance. The programming languages listed to the left of the time line are shown at their inception year. Not all the programming languages shown contain concurrency constructs.

designers and software engineering researchers in these early days when the two disciplines were beginning to distinguish themselves from one another.

Ada [DOD 1980], also designed in the mid-1970s, was the first widely used programming language designed with language constructs for concurrency, based on Hoare's message-passing CSP. Java [Gosling et al. 1996], introduced in the mid-1990s, advanced the monitor concept by encapsulating access to shared data by synchronized methods and code blocks.

As is evidenced by this historical narrative, the early unified software engineering and programming language community heavily influenced the concepts and the notation used to express concurrency in programming languages. There is also evidence that the more recent software engineering research has focused on building tools and techniques for managing and analyzing concurrent programs. This includes work in program analysis, testing, anomaly detection, replay and debugging of concurrent programs. Surprisingly, except for the early research in the 1960–1970s, there has been little influence as evidenced through published works of software engineering research on the design of concurrent constructs in programming languages, although language constructs for concurrency continue to be developed [IEEESW 1989].

The time line for concurrent constructs is given in Figure 3.

## 4. VISUAL PROGRAMMING: FROM RESEARCH TO PRACTICAL USE

This report thus far has focused on the impact of software engineering research on programming languages that have textual syntax, but visual syntaxes for programming have also become important in recent years, as the popularity of Visual Basic helps to demonstrate. Although software engineering research generally aims to improve the quality of software and the processes, languages, and tools used to create this software, most of this work focuses only indirectly on the human programmers who create the software. In contrast to this majority, research on visual programming aims directly at the human aspects of programming. Therefore, in this section, we focus on the evolution of visual programming properties that are becoming mainstream, as they relate to the cognitive problem-solving needs of human programmers.

*Visual programming* refers to any system that allows a program to be specified using two- (or more) dimensional expressions, such as diagrams, icons, color coding, multi-dimensional annotations, and/or even graphical actions themselves [Burnett and McIntyre 1995]. When multi-dimensional expressions, called *visual expressions*, are the syntax of a new programming language, the system is called a *visual programming language*. For example, Prograph [Cox et al. 1989] was a research (and later, commercial) object-flow visual programming language in which components of the classes were specified by placing icons, and whose method syntax consisted of dataflow diagrams. Another example is Peridot [Myers 1990], a research visual programming language for programming user interfaces, part of whose syntax was the demonstration of actions on data objects. When the visual expressions are used to provide a supporting environment for a preexisting textual programming language, the system is a *visual programming environment*. For example, Pecan [Reiss 1984] was a research visual programming environment for Pascal. Snapshots of visual programming approaches of the 1980s, when many of these approaches emerged, are shown in Figure 4.

In general, the aim of visual programming is to reduce the cognitive burden on human programmers, through devices that reduce the mental effort required to access and make sense of the available information. Although visual programming research spans multiple subdisciplines of computer science, a large portion of it has taken place within the software engineering community.

The impact of visual programming research upon modern programming languages lies in two areas. The first area of impact is the use of visual communication devices to illustrate programs and their behavior in software development environments for professional programmers. Today visual devices that descended from early visual programming research can be found in many modern software development environments. The second area of impact is the design of visual programming languages that are viable for audiences who are not expert programmers in traditional programming languages. Visual Basic, which is of great practical importance given its wide usage, demonstrates both of these impact areas.

In other sections of this article, we have discussed software engineering research culminating in programming language constructs that enable language

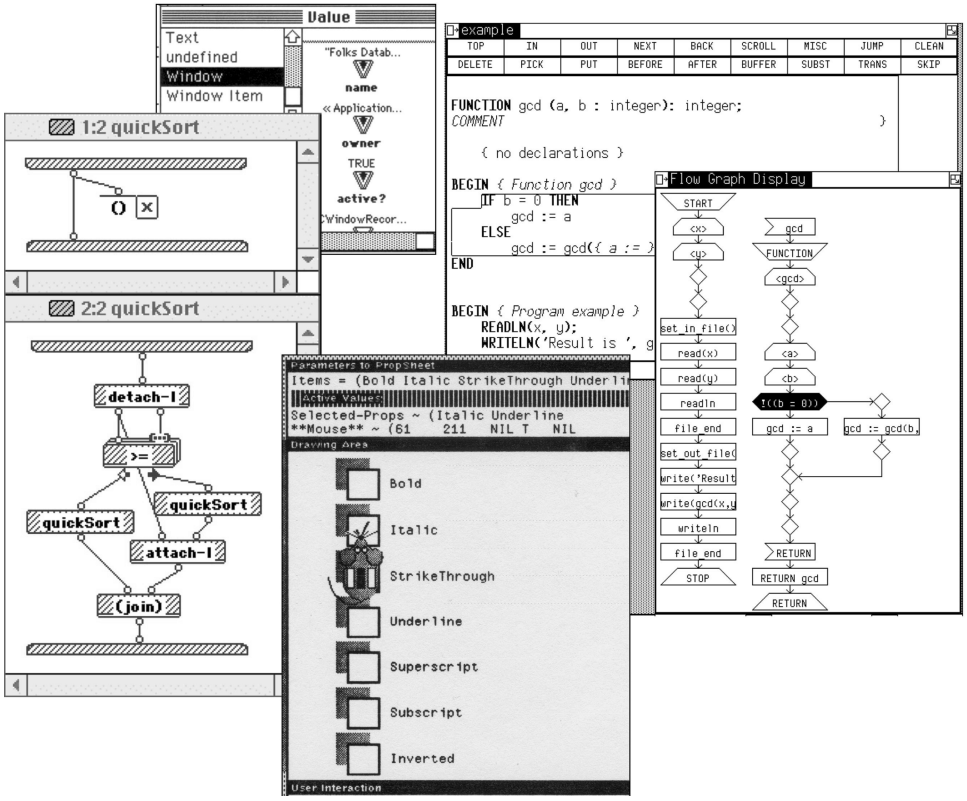


Fig. 4. Snapshots of 1980s era visual programming systems: Clockwise from left: Prograph programming was done by drawing dataflow diagrams (left; figure re-created from Cox et al. [1989]), and Prograph objects arriving at the ports on the dataflow diagram could be inspected by double-clicking to reveal the object structure and values shown (top; figure is from TGS Systems [1990]). PECAN (right) featured a variety of graphical and textual views of Pascal programs, and visually highlighted the portion of the code being executed in both of the views shown (figure is from Reiss [1984]). Peridot allowed programmers to specify user interfaces by directly placing the GUI objects as desired and demonstrating the objects' responses to GUI events (bottom; figure is from Myers [1990]).

processing systems to automatically detect or prevent certain reliability issues in a program. For example, abstraction constructs allow some inappropriate data accesses to be prevented by the system, and type checking allows the system to detect type errors. However, the primary influence of the visual programming work was less on new language constructs *per se*, and more on language properties that are realizable by visual programming mechanisms. Further, these properties are tied to enabling human programmers, rather than language processing systems, to detect or avoid certain reliability issues in a program. We begin by considering four properties that emerged from the visual programming research done in the software engineering community prior to 1991: concreteness, directness, explicitness, and immediate visual feedback. We then show examples of how these properties have been instantiated in Visual Basic, which emerged in 1991.

### 4.1 Emergence of Visual Programming Properties

Although there were a few pioneering works in visual programming in the 1960s and 1970s (e.g., Sutherland [1963, 1966]; Smith [1977]), in the 1980s visual programming gathered momentum as a research area in the programming language and software engineering communities, initially following two approaches. One concentrated on mapping each traditional programming construct one-to-one to an icon (e.g., Glinert and Tanimoto [1984]). The other concentrated on new visual approaches to programming that deviated significantly from the traditional methods, such as programming by wiring together constraints (e.g., Borning [1981]) or by demonstrating desired behaviors on the screen (e.g., Rubin et al. [1985]). Many of these systems had advantages that seemed exciting and intuitive when demonstrated with toy programs, but ran into difficult problems when attempts were made to extend them to more realistically sized programs. These problems led to an early disenchantment with visual programming, causing some to believe that visual programming was inherently unsuited to *real* work.

To address these problems, visual programming research moved in two directions. The first direction involved the incorporation of visual expressions into integrated programming language environments, for those specific parts of software development in which visual expressions demonstrated obvious advantages for human programmers. The second direction, involving the development of domain-specific languages, increased not only the kinds of applications suitable for visual programming from start to finish, but also broadened the set of people who could program.

4.1.1 *Visual Programming via Programming Environments.* Starting in the early 1980s, integrated programming environments such as Smalltalk [Goldberg 1984], the Cornell Program Synthesizer [Teitelbaum and Reps 1981], Cedar [Teitelman 1984], Pecan [Reiss 1984], and Gandalf [Habermann and Notkin 1986] began incorporating visual expressions. (For a history of the increasing use of visual expressions during the evolution of language environments, see Ambler and Burnett [1989].) The synergy of visual language research with language environment research led to the advent of practical visual programming environments for traditional languages. The visual expressions were used for selected aspects of software development (e.g., for GUI programming), for explicitly depicting relationships and flow, and for visually combining textually-programmed units to build new programs (e.g., Hirakawa et al. [1990]). Eventually, this merging combined syntax-directed editing and grammars for both textual and visual expressions (e.g., Chang et al. [1989], Crimi et al. [1990], and Helm et al. [1991]), which further increased the viability of incorporating visual expressions into programming environments.

The successes and failures from these early approaches led to investigation into more general properties of visual programming, and how these properties were linked with supporting human programmers' software development efforts. For example, one critical reason for the success of incorporating visual expressions into programming environments was that doing so promoted the property known as *directness* or *closeness of mapping* [Green and Petre 1996]:

allowing the programmer to express solutions to a subproblem (e.g., GUI layout) using a notation similar to the problem itself (e.g., by drawing the desired layout). Directness is important because it avoids the cognitive burden and associated error potential of translating from one way of describing ideas to a different way.

Other properties supporting human problem-solving are explicitness, concreteness, and immediate visual feedback. *Explicitness* in depicting relationships and dependencies, such as via dataflow diagrams, eliminates the error-prone work of tracking down these relationships manually. The property of *concreteness* denotes working with concrete data values to express or explore program logic. Consider a function or procedure definition, which is abstract in the sense that it does not include any actual data from a specific instance of invocation. When writing such a function, programmers have to model (in their heads or on paper) the actual data that will be present when the function is invoked. This modeling process is an additional attention cost that concrete sample values can help to eliminate. A system's ability to provide *immediate visual feedback* about program semantics, such as immediately showing the calculated result on values, is facilitated by concreteness. This feedback feature has been shown to be heavily used in problem-solving by both novice and expert programmers [Green and Petre 1996]. As visual programming research matured, it became evident that the support visual expressions can lend to properties such as those described above, not the use of visual expressions *per se*, is the aspect of visual programming important to human productivity in problem solving and programming.

4.1.2 *Domain-Specific Visual Programming Languages.* The other direction followed by visual programming researchers was to increase the kinds of applications suitable for visual programming from start to finish, through the development of domain-specific languages. These researchers incorporated the previously discussed properties. In addition, because many domain-specific languages are aimed at specific types of audiences as well as specific types of applications, these researchers began to focus more carefully on the audiences for whom their languages were intended. Thus, visual programming became an enabling mechanism for a phenomenon known as *end-user programming*, in which people not trained as programmers develop their own applications.

Designing a visual language for use in a specific problem domain by a specific audience is an example of language research that aims at supporting the directness property. It allows people to program directly in their domain-specific notations using visual expressions (e.g., icons and diagrams) reflecting the particular abstractions, diagramming traditions, and vocabulary specific to that domain, instead of requiring people to translate their vocabularies to traditional programming language terminology. This approach produced a number of successes, first in research and then in the marketplace. For example, today such audiences include teachers creating educational simulation programs by demonstration (e.g., Roschelle et al. [1999]) and laboratory scientists, who can graphically wire measurement data through icons representing summarization and visualization tools using LabView, a commercial visual programming

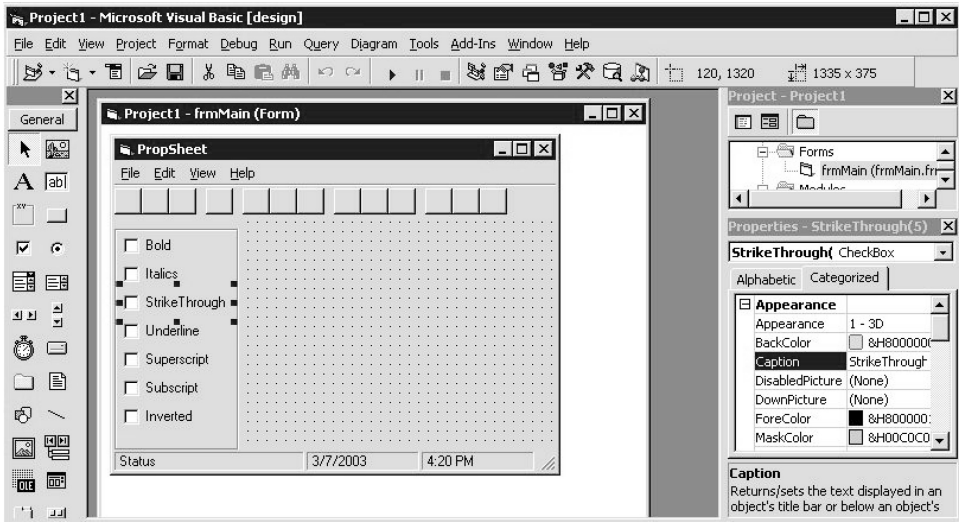


Fig. 5. Visual Basic. The center window shows Visual Basic's direct placement of GUI objects in the desired GUI layout. The StrikeThrough subobject has been selected, causing its type to be displayed along with its "properties" (lower right of the screen shot), i.e., the object's data members' names and values.

language for the domain of laboratory measurement [Baroth and Hartsough 1995].

### 4.2 Visual Basic's Instantiations of Visual Programming Properties

The properties described in the previous section can be seen in three of Visual Basic's features: its support for GUI programming, its use of visual programming in its integrated language environment, and its programming-by-demonstration features (found in the language subset known as Visual Basic for Applications or VBA).

The precursor to Visual Basic first took shape during 1988–1990 as a visual programming language for Windows shell programming not associated with Basic [Arnson et al. 1992; Cooper 1996]. Microsoft purchased the concept, married it to Basic, and Visual Basic made its debut in 1991. In the 1980s and early 1990s, the visual programming properties discussed previously had not yet become widely recognized as such; rather, researchers were still experimenting with specific features instantiating those properties. Thus, it is not surprising that Visual Basic's instantiations of these properties correspond to the specific features of the pioneering systems such as those in Figure 4.

For example, the directness property figures prominently in one of the best known features of Visual Basic, namely the ability to specify a GUI layout by directly manipulating the visual properties of GUI objects (e.g., their placement and size). The version of this feature in Visual Basic is shown in Figure 5. The center window shows the support for programmers to be able to directly place objects in a GUI as desired. This feature was pioneered in the Peridot visual language [Myers 1990], as can be seen by comparing this window with Peridot's version (bottom of Figure 4).

An instantiation of the immediate visual feedback property can also be seen via the GUI objects in Figures 4 and 5. As these figures show, in Visual Basic (as well as Peridot), moving and resizing the source code (i.e., elements of the drawing on the screen) is immediately reflected in how the GUI appears, since the representation of the source drawing and its ultimate result are one and the same.

The immediate visual feedback property is due in part to the use of concreteness in Visual Basic in the same manner as in Peridot. As Figure 5 shows in its center window, programming is done via specific instances, not just abstract variables, in performing manipulations of GUI objects. Also, the StrikeThrough subobject has been selected, causing its properties to be displayed in the properties pane (lower right of Figure 5), which lists the object's attributes/data members in a column on the left along with their values in the right column. The programmer can access and manipulate these properties, a capability reminiscent of the Values window of Prograph (top of Figure 4). In Prograph, as later in Visual Basic, any value (object) can be opened via direct manipulation, causing the object's type to be shown (highlighted top left of Value window), along with its attributes'/data members' names (below the icons) and values (above the icons).

The VBA subset of Visual Basic is embedded in several Microsoft products, including Excel, PowerPoint, and Word. VBA has been the language to support end users in the construction and editing of macros. Although the user can create these macros by typing in VBA directly, the more usual approach is to begin by demonstrating the desired logic via example (as in Peridot). For example, in Excel a user can demonstrate a macro by recording the actions they perform on Excel objects (cells, formulas, spreadsheets, etc.). VBA's support for programming-by-demonstration is another instantiation of the directness property.

The explicitness property can also be seen in VBA. In VBA, when a user chooses to step through execution one line at a time, the editor highlights the lines being executed, a feature found in other modern visual environments as well. This feature dates back to the Pecan visual programming environment of 1984-1985 [Reiss 1984], which graphically highlighted the portions of code being executed, as shown at the right side of Figure 4. This is an instantiation of the explicitness property, since the system is explicitly depicting control flow as the system executes.

Visual Basic has, of course, also incorporated numerous language features that are not related to visual programming. For example by 1992, Visual Basic included the concept of objects, which eventually led to incorporation of a form of classes in 1996.

### 4.3 Software Engineering Trends in Modern Visual Programming Research

Three recent software engineering trends have emerged in visual programming research. First, software engineering research regarding language features such as abstraction, exception handling, and types, discussed in Section 3, has begun to have significant impacts on visual programming language features

in the last few years. In order for visual programming languages to scale to increasingly large software projects, researchers have worked to devise ways to incorporate more powerful language features without loss of the properties that assist human problem solving. Burnett et al. [1995] surveys the beginnings of visual programming research in these directions.

Second, with the advent of end-user programming, emerging projects have begun to consider the concept of *end-user software engineering* [Burnett et al. 2003]. This research draws from results of software engineering research into how to support phases of the software lifecycle beyond the coding stage. Combining this research with visual expressions, machine learning techniques, and emerging HCI research, researchers are now developing new software engineering methodologies to help end-user programmers reduce defects by supporting them beyond the coding stage, such as with incremental visual testing [Rothermel et al. 1998], semi-automatic defect detection [Bottoni et al. 1997; Miller and Myers 2001; Raz et al. 2002], and tightly integrated assertion mechanisms [Burnett et al. 2003].

Third, as visual programming research has matured, researchers have learned that visual programming research is not a matter of learning whether visual expressions are overall superior to text-only notations: every notation has strengths and weaknesses [Green and Petre 1996]. Rather, the essence of visual programming research is to learn how to harness visual programming's expressive power to support particular properties that assist human cognition, such as the closeness of mapping, concreteness, explicitness, and immediate semantic feedback properties discussed above. Progress in this direction has been made largely through multidisciplinary work that draws not only from software engineering and language research but also from human-computer interaction (HCI) research, with a strong emphasis on empirical work. One example of this direction is the work of Pane et al. [2002], who devised a domain-specific programming language only *after* performing empirical work with the intended audience, to elicit the principles and constructs that were eventually used in the design of the language. This increasing emphasis on drawing from and contributing to empirical foundations is having an important impact on the measurable effectiveness of visual programming. Further, by demonstrating the impact of empirically-based research involving humans on the genuine effectiveness of software engineering techniques, an approach long advocated by several software engineering researchers [Gannon 1977; Basili et al. 1986; Tichy 1998] but too rarely followed, visual programming research makes a valuable methodological contribution.

The time line graph for visual programming is provided in Figure 6.

## 5. INTERVIEWS WITH PROGRAMMING LANGUAGES DESIGNERS

To obtain firsthand historical information about the influence of software engineering research on programming language design, a set of language designers were contacted by email or phone and asked about influences on their work. Designers who responded to us were Professor Boris Magnusson (Simula 67), Professor Niklaus Wirth (Pascal, Modula), Dr. Jean Ichbiah (Ada),

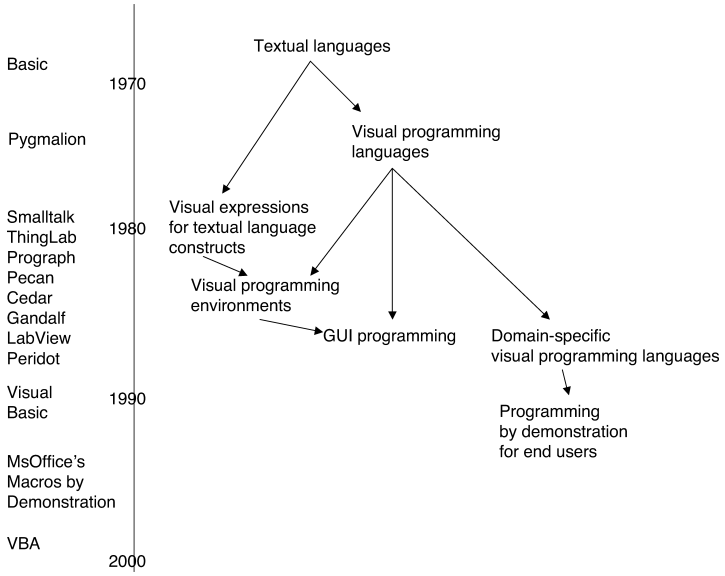


Fig. 6. Time line for visual languages.

Dr. Brian Kernighan (scripting languages), Dr. Bjarne Stroustrup (*C++*), Dennis Canady (Visual Basic), Tim Lindholm (Java), and Tucker Taft (Ada 95). Each of the participating designers was sent the list of questions below and then either answered them by email or phone.

The following questions were asked about the influences on the interviewees' language designs, where **xxx** stands for the particular programming language with which the designer was involved.

(1) What were the problems that you were addressing in your design for PL xxx?
(2) What programming practices or software engineering research do you think influenced your language design, positively or negatively?
(3) Can you recollect any of the papers you were reading, the conferences you were attending and the people you were talking to when you were designing your language xxx?
(4) Were there specific applications that drove your design?
(5) If you were to design a programming language today, how would it differ from current programming languages?

Dr. Kernighan, designer of *awk*, indicated that he thought there was little influence of software engineering research on scripting languages. Kernighan remarked, "I think there's been only minimal back and forth between scripting languages and software engineering, though perhaps it's increasing. Speaking specifically for *awk*, there was no influence at all from any 'software engineering' perspective . . . as for the influence of scripting languages on software engineering, I don't see much there either." Thus, he declined to answer the questions we posed.

In this section, excerpts from these interviews are presented, in roughly chronological order of the language they designed, to show relevant influences between software engineering research and programming languages, as recalled by the language designers.

**Professor Boris Magnusson (Simula 67):**

Professor Magnusson has personal knowledge of the early days of Simula 67. We were grateful for his participation, especially given the recent untimely deaths of Nygaard and Dahl, who jointly won the ACM Turing Award in 2001 for the development of Simula 67.

**1. What were the problems that you were addressing in your design for Simula 67?** Complexity of large software systems. The experience behind Simula 67 came from developing large discrete event simulation models, programs that indeed tend to get large, complex and involved. These models also include a notion of concurrency, although it is not necessarily reflected in the execution of the model.

**2. What programming practices or software engineering research do you think influenced your language design, positively or negatively?** The notion of a safe language inherited from Algol is perhaps in this category, the strive to find trivial errors in a program, errors in respect to the language definition, at an early stage. The inclusion of automatic garbage collection, available in LISP systems at the time, was motivated by the same reason. Tony Hoare's work on record handling influenced the formation of the class concept.

[Ed. note: There is no separate response in this interview to question 3.]

**4. Were there specific applications that drove your design?** No! It (Simula 67) is a general purpose programming language.

**5. If you were to design a programming language today, how would it differ from current programming languages?** I would put more stress on supporting building of notions of concurrency in the language. This point is sadly missing in modern languages. Introducing ONE notion of concurrency in a language, like *Thread* in Java, is too limiting. If you, for example, want to write a program that animates a simulation, you need two different time bases and scheduling mechanisms at the same time.

**Professor Niklaus Wirth (Pascal, Modula):**

Professor Wirth was awarded the ACM Turing Award in 1984 for developing several innovative computer languages, including Modula and Pascal.

**1. What were the problems that you were addressing in your design for Pascal and Modula?** Two purposes stood in the foreground: 1.To obtain a language suitable for system programming. . . 2. To obtain a language suitable for teaching the fundamentals of programming in a lucid, systematic manner, without undue references to particular computers and implementations.

**2. What programming practices or software engineering research do you think influenced your language design, positively or negatively?** Software Engineering was not yet a subject of research; even the term did not exist. Programming methods was the name of the subject. Of great influence to Pascal was Structured Programming, put forth by E. W. Dijkstra. This method of proceeding in a design would obviously be greatly encouraged by the use of a

Structured Language, a language with a set of constructs that could freely be combined and nested. The textual structure of a program should directly reflect its flow of control.

Another key idea stemmed from the fact that the computing profession was split into scientific computing and commercial data processing, and from my attempt to unify their bases .... Of negative influence was the need to win programmers for a new language. This need forced me to retain constructs and facilities that programmers were used to and did not want to miss, although I knew they would have to be left out in the interest of safer programming. Examples are the *go to* statement, the variant record (data overlay), and incomplete parameter type specification.

**3. Can you recollect any of the papers you were reading, the conferences you were attending and the people you were talking to when you were designing Pascal and Modula?** By far the biggest influence came from the *Report on Algol 60*. My interest in programming languages was increased by Prof. A. van Wijngaarden, who spent a sabbatical semester at Berkeley, where I was working for my doctorate. I was brought in contact with the *Algol Working Group 2.1* of IFIP, and there the most influential colleagues were E. W. Dijkstra, C. A. R. Hoare and P. Naur. We attended several work meetings and exchanged many ideas, which first led to Algol W (1966, implemented at Stanford), then to Pascal (1970, implemented in Zurich).

Modula-2 followed in 1979 as intended successor to Pascal, and it grew out of the need to further develop Pascal to cater for the needs of large systems (software engineering). Several ideas came from the language Mesa (B. Lampson, J. Mitchell, J. Morris, Ch. Geschke), to which I was introduced during a sabbatical year at the Xerox Palo Alto Research Center in 1976/7. My main contribution was (as to some extent also in the case of Pascal) combining, molding all constructs into a single, harmonious framework, and in simplifying them, extracting the essentials and discarding the bells and whistles.

**4. Were there specific applications that drove your design?** In the case of Pascal, the need for a decent language for teaching and for system programming. In the case of Modula-2, the need for a language adequate for modular system building, and the desire to finally get rid of antiquated constructs (such as *go to*).

**5. If you were to design a programming language today, how would it differ from current programming languages?** If you mean Java and C# by current languages, then my answer is: Look at Oberon (1988). Most of their features had been present in Oberon, my successor to Modula-2, some 7–10 years earlier. The chief difference to Java and C# is its size, in terms of number of syntactic rules, of number of pages of definition, and in the size of its compilers. The Report was only 16 pages long, and the entire Oberon compiler took about 45 Kbyte, was very efficient, and due to its transparency, highly reliable. Oberon was Modula-2 with object-orientation.

The sad thing about new developments is that they always seem to turn out more complex rather than simpler. Yet it is known that progress in (mathematical) sciences had mostly been through simplification and unification of concepts.

**Dr. Jean Ichbiah (Ada):**

Dr. Ichbiah was the principal designer of Ada.

[Ed. note: There is no separate response in this interview to question 1.]

**2. What programming practices or software engineering research do you think influenced your language design, positively or negatively?** Doing programming language design in the mid-1970s, we were addressing a set of objectives derived from software engineering, definitely. These were the years when hardware prices were coming down significantly and machines were becoming much cheaper. It was possible to run much larger applications, but previous programming methods didn't scale. It was like *constructing a small hut versus a skyscraper*. Trying to scale methods of developing programs was difficult; the costs were increasing in exponential fashion rather than linearly with program size. . . we learned how to scale with linear cost, by introducing modularization and packaging.

Dijkstra had a profound idea, that programs should become closer to text. If you could read a program and understand it, then it was a good program. You should be able to read a program like a book; this was a notion of *linearity*. The initial ideas were expressed in Dijkstra's *Go tos considered harmful* paper, but this title was a misnomer. Dijkstra was laying the foundation of how to understand a program through linear thinking. Twenty-five years later we can see the absolute necessity for this idea, that the complexity of code be linear in its size. This was essential to allow systems to grow in size. The key emphases were to make programs readable, maintainable, and reliable. Readability is by far the more important issue as it controls the other two.

Dijkstra saw an infinite space of possible programs and used a constructive approach to extract an infinite subset of programs that you can read easily and can convince yourself that they are correct. . . . As an example of this in the design of Ada, I am proud of the textual separation of specification and implementation, as it lets you know what you need to know to use a module, without getting into details of implementation.

We systematically designed PL features in a manner like structural engineering; that is, we would look for failure patterns and then try to make something that would not break.

**3. Can you recollect any of the papers you were reading, the conferences you were attending and the people you were talking to when you were designing Ada?** There were several working groups that met several times a year. Two were the *IFIP Working Group* 2.4 *on System Implementation Languages* [IFIP 2004] and the *Purdue Europe LTPL-E committee*. There was a conference on system implementation languages in 1973 or 1974 in Trondheim. This was a not too big conference and at its end, about 10 researchers met and formed an IFIP working group that provided collegial feedback and stimulation to participants about 3–4 times a year. Participants included Bill Wulf and Jim Horning. Another influential group was the IFIP 2.3 group on *Algorithmics*. These groups gave researchers a chance to meet multiple times a year, present what they were working on and to discuss it with peers who could comment and criticize.

**4. Were there specific applications that drove your design?** The Ada PL was fairly general, however one new aspect it addressed was real-time programming. The metaphor used at the time, to talk about applications that would use this new paradigm, was to think of airplanes as *software with wings*. Another new aspect of computing addressed by Ada was programs designed to be executed on distributed computers.

**5. If you were to design a programming language today, how would it differ from current programming languages?** If I were to design Ada today, it would not be much different in a way. I would try to make it even simpler. Although I do not want to design another PL now, I would take Ada and trim it. For example, I would eliminate parameterized types (i.e., variant records) Generics also appear in Ada but may be less useful than originally thought .... Inheritance seems to me to be a low-level concept; this is a detail of implementation rather than a conceptual thing.

When we were inventing modularity in the 1970s, we were solving the problem of scalability. This resulted in making software that scales linearly. Now there is a full industry of components that uses this philosophy, across many programming languages. Improvements in programming languages today are second order effects.

**Dr. Bjarne Stroustrup (*C++*):**

Dr. Stroustrup was the designer and original implementor of C++.

**1. What were the problems that you were addressing in your design for *C++*?** Initially, I simply needed a tool to help me with a project to distribute the Unix kernel across multiple machines. I saw two areas of need: To express the logical partitions within the kernel code (the kernel was—and is—written in C, and C doesn't provide facilities for directly representing logical partitions and their communication paths) and to write simulations to determine the effects of different communication patterns resulting from different software configurations and different types of hardware support.

**2. What programming practices or software engineering research do you think influenced your language design, positively or negatively?** Ideas for operating system organization and application building, which I was exposed to at Cambridge (notably the CAP computer and its use of hardware protection) and Newcastle (the *C++* notion of exceptions and their use) was directly inspired by the work on reliable systems by Brian Randell's group.

The "object-oriented" ideas from Simula 67, and appreciation of static type checking from Simula 67 and Algol 68 (this appreciation wasn't widely shared among people building operating system kernels).

**3. Can you recollect any of the papers you were reading, the conferences you were attending and the people you were talking to when you were designing *C++*?** I went to SIGOPS, USENIX, and a distributed systems conference. ... However, my main influence was my colleagues in the Bell Labs Computer Science Research Center, including Sandy Fraser, Steve Johnson, Brian Kernighan, Doug McIlroy, Bob Morris, Gregg Chesson, Stu Feldman, Dennis Ritchie, and Steve Bourne. This was in the very early years where the

key notions of C++ were formed. I think that it is significant that the systems people dominated over the language people.

**4. Were there specific applications that drove your design?** It started with the distributed operating system kernel, which I never completed, and the simulations. For years after, my bread and butter work in the Labs related to various forms of simulation (network traffic, telephone use, board layout, processor register design, etc.). I also worked with embedded systems and a variety of systems programming tasks. I remember that the application that finally caused me to introduce operator overloading wasn't the obvious mathematical uses of operators, but modeling of bits, registers, and signals in a processor design.

At the time, the UNIX system and networking loomed large in the collective imagination of the Lab, as did telecommunications applications (not just huge switching applications, but also small switches, embedded control applications, and hand-held gadgets usually weighing a few pounds). In the 1980s, Bell Labs was an environment very rich in diverse real-world applications and people with both research training and real-world experience.

**5. If you were to design a programming language today, how would it differ from current programming languages?** A language design—like all design—should arise from a need. I don't clearly see a problem with current programming practice that would best be solved by a new language. Note that when I designed C++, I based it on an existing language, C, to build on existing strengths to have a complete and useful tool very early on. Should I design another language, I'd probably do something similar again.

**Dennis Canady (Visual Basic):**

Dennis Canady was a member of the original Microsoft Visual Basic design team.

**1. What were the problems that you were addressing in your design for Visual Basic?** Visual Basic was shaped during the emergence of Windows. Since Windows itself was just emerging, there naturally weren't many Windows applications, but applications are needed to make an O/S useful. So, we wanted to make Windows applications easy to write. For one thing, we wanted an interactive development environment in which the performance of the language would be similar to that of compiled code, but that would have the interactivity and other nice features that come from an interpreted environment. Another goal was to give developers of business applications a way to develop Windows applications without having to write all that window-oriented code by hand.

**2. What programming practices or software engineering research do you think influenced your language design, positively or negatively?** Interactive debugging practices were an influence. In fact, Visual Basic's design was influenced by our desire for an execution model that could allow a good interactive debugging environment. For example, we wanted to provide quick feedback about syntax errors, and to point to identifiers in the source. Turbo Pascal came out about that time, and had an influence on our design. Forth was also an influence, which led to a threaded P-code interpreter. Of course, other versions of Basic, including QuickBasic and Microsoft's MacBasic, were also

influences. We also looked at visual programming languages, but at the time most of them were dataflow-based, which did not seem like a good fit. I think we saw ThinkPad, but not many other programming-by-demonstration languages. Object models (C++, Ada, Smalltalk) were also influences. But creating an application in Smalltalk was too large, too isolated from Windows, and too slow. We asked "How can we do better?" Then Ruby came along [Ed note: this was the code name for the visual programming language for Windows shell programming mentioned in Section 4.2.]. We purchased the concept, and used it for Visual Basic.

**3. Can you recollect any of the papers you were reading, the conferences you were attending and the people you were talking to when you were designing Visual Basic?** I went to some OOPSLAs around that time. I was also following the *C++* world, and Bertrand Meyer's work on Eiffel.

**4. Were there specific applications that drove your design?** Windows business applications.

[Ed. note: There is no separate response in this interview to question 5.]

**Tim Lindholm (Java):**

Tim Lindholm is a Distinguished Engineer for the Java Software group at Sun Microsystems. He was an original member of the Java project at Sun and remains the architect of the Java virtual machine.

**1. What were the problems that you were addressing in your design for Java?** The language that became known as Java was initially called Oak, and was originally used for programming an embedded device, a sort of cross between a large PDA and a super-duper remote control. The next use of Oak was as the language for a video-on-demand project, the project in which I was first involved. (Early in 1994, Lindholm joined the project.) When the market for video-on-demand was created in 1994, a portion of the team went off to investigate how to use Oak as a safe way of deploying applications over the World Wide Web. It is interesting that in the design of Oak, programming language concepts were selected that had in some cases been around for 20 years. Even so, the first two applications of the technology went nowhere. . . One of the main goals of Oak was to enable programs to safely move around a network of heterogeneous computational units. When moving Oak to the Web the team added a small number of key technical innovations, among them class loaders and verification of program type safety. . . . Then known as Java, the language *hit the wave* of popularity just right: while HTML could be downloaded to browsers on various machines, it only provided static content; Java applets could make Web content dynamic, and *people grokked the idea*. . . . There were 2 million people working with HTML and considering themselves to be programming the Internet. At that point we offered them Java as a power tool. . . .

**2. What programming practices or software engineering research do you think influenced your language design, positively or negatively?** At this time, people were starting to be upset with *C++* because it was hard to write programs you could understand and it was not safe, compiling to platform-specific executable code. Gosling called Java, *C++ without guns and knives*. The Sun team could not use *C++* for the applications they needed to write. When

asked "Why was $C++$ inadequate?" he replied, $C++$ programs compiled to native binaries, which made it impossible to safely move them about the Internet, and at least hard to make them run on different architectures. Also, we felt that $C++$ was sufficiently hard to program in that less-skilled programmers, who we hoped to reach, would have difficulty with it.

Software engineering practices that influenced us included many object-oriented ideas, as expressed in $C++$ and Smalltalk-80. Java designers need to give credit to folks who developed other object-oriented languages, from whom they borrowed ideas.

**3. Can you recollect any of the papers you were reading, the conferences you were attending and the people you were talking to when you were designing Java?** The people I was talking to were pretty much within this group at SUN in 1994. There were 30 in the video-on-demand group. Of these about 12 of us worked on the Java that people would currently recognize. Between July and December of 1994 we wrote WebRunner, a browser that ran Oak applets.... It is true to some degree that we went into a back room and designed the language. We had problems to solve that directed our work. We didn't design this by committee or opinion poll. We took unto ourselves as working programmers, to answer the question: *What is it that we want?* We were designing a language for ourselves to use.

[Ed. note: In a later communication, Lindholm acknowledged that "the Oak team did have a lot of highly experienced language designers and implementors who were very aware of the computer science literature."]

**4. Were there specific applications that drove your design?** Specifically we were targeting applications for the Internet to run on browsers. But we were aware that the properties of the language we were designing would be much more broadly usable.

**5. If you were to design a programming language today, how would it differ from current programming languages?** I am leery of object-orientation as a kind of religion that drags in complexity in the guise of simplicity. It bugs me that sometimes to use an object system requires mind-twisting discussions on what things mean. Real programmers don't have time for religious arguments. Nonetheless I would like to see the world take another run at a Smalltalk-like language, something simpler than current popular languages. Java was striving for elegance and simplicity while retaining familiarity and usability. For example, Gosling went against the grain, and refused to put stuff into Java, such as operator overloading, which makes the language harder to learn and makes it easier to make mistakes in programming. Now that the world is more comfortable with garbage collection, threads, and virtual machines, it would be desirable to try another programming language (like Smalltalk) where more cleanliness and elegance are embodied.

**Tucker Taft (Ada 95):**

Tucker Taft was chief language designer of Ada 95.

**1. What were the problems that you were addressing in your design for Ada 95?** In general, Ada 83 was viewed as a "static" and somewhat "closed" language. Our goal for Ada 95 was to make the language more flexible, more

extensible, and easier to interface with the outside world, be it subsystems written in other languages, external hardware devices, or underlying operating systems. At the same time, we wanted to preserve Ada 83's strong focus on safety, reliability, and readability, and its excellent support for 'programming in the large' thanks to its very strong type checking.

**2. What programming practices or software engineering research do you think influenced your language design, positively or negatively?** The big jump for Ada 95 was to incorporate the thinking on inheritance and polymorphism (i.e., object-oriented programming). One of our challenges was to provide the dynamic extensibility of inheritance and polymorphism, without the negatives of strong coupling that inheritance can create between a base type and a derived type.... The point is that we were very concerned with the software engineering aspects of our design choices. Whether we could point to specific "research" that was critical to our work, that would be harder. But generally the principles that were developed in the software engineering community were very important.

**3. Can you recollect any of the papers you were reading, the conferences you were attending and the people you were talking to when you were designing Ada 95?** At a high level, I would say that the Ada 9X design team was unusually aware of the computer science literature, and formal verification literature, as well as software engineering principles. I would not say we were avid readers of software engineering *research*. In fact, I would say we more often bemoaned the lack of real research into the software engineering advantages or disadvantages of particular language design choices. We had to rely on our own experience and *gedanken* experiments more than we would have liked.

I attended OOPSLA a few times. I also read a number of papers relating to modular thread synchronization mechanisms. Some of this was pretty old stuff, such as Concurrent Pascal. Other articles were more recent, such as work on a language called Orca. All of these had various twists on the notion of a monitor. We ultimately ended up with something called "Protected Types" which combine the guards of Ada 83's rendezvous (which were in turn inspired by CSP), with the passivity of monitors. From a software engineering point of view, I (humbly) think protected types were a great synthesis of old and new ideas.

**4. Were there specific applications that drove your design?** Very large applications; real-time applications; embedded applications. But in addition, Ada 83 had emerged as one of the most widely used languages for teaching programming in college. Because of that, there was a desire to lower the entry barrier to the first time user. So there was also a desire to make it easier to write short programs in Ada 95 that did something useful in a typical desktop environment, so we added more features for dealing with things like command line arguments (not usually relevant in embedded or real-time applications), simplified I/O, etc.

**5. If you were to design a programming language today, how would it differ from current programming languages?** I still feel there is a lot of research worth doing on the relative productivity and error rates for languages with different features or philosophies. But I realize this is difficult research to

do. I think most people agree about the fundamental goodness of abstraction, modularity, information hiding, and encapsulation. But we certainly don't all seem to agree about how that should translate into particular language features. I also think there is an important aspect of language design that has *not* been talked about much, and that is the "human engineering" of languages.... By human engineering I mean whether the language is error prone in various ways.

[Ed. note: Taft continued with a critique of several design decisions in two current languages, C# and Java.]

I was pretty disappointed in Java. It has been a roaring commercial success, but from a language design point of view, it took a lot of steps backward. They dropped enumeration types, which makes absolutely no sense to me from a software engineering point of view. People of course reinvent them, but they are back to the bad old days of defining named constants, with no compiler support, and no type checking.... The other big step backward for Java was eliminating the separation of specification from implementation.

The language C# actually seems better than Java in some of these respects.... I think C# took a step back relative to Java (and a big step back relative to Ada) in relegating thread synchronization support to a class, rather than building the concurrency primitives into the language. Java at least has automatic locking/unlocking as part of making a synchronized method call. Relying on the programmer to balance their lock/unlock calls is very error prone. Ada goes further and provides significant task safety and coordination guarantees through the use of guards in protected types. This sort of thing is very difficult to accomplish using a thread *class* with concurrency primitives available as a set of methods like *lock, unlock,* and *wait.*

## 6. SUMMARY

Programming language features such as exceptions, procedural and data abstraction, and types present evidence of the strong ties between software engineering and programming language research (and practice). Individuals who have worked in both areas tie them together as well as the attention paid by both communities to relevant research in the other community, attested to by their references to this work in their own papers and presentations at each others' conferences. Cotemporality of developments likewise attests to the influences of these fields on one another.

It is perhaps unsurprising that this symbiosis exists between programming languages and software engineering, with regard to software reliability research and exceptions, whose purpose is to provide programmer direction for unusual (but anticipatable) circumstances. Similarly, software engineering research in modularity and reuse dovetails nicely with programming language design emphasis on control and data abstraction, which evolved into inheritance with visibility controls. Likewise, strong typing can be seen as a response to the emphasis on software reliability; somewhat later, user-defined types and generics, are mechanisms to provide for software reuse. In addition, software engineering research on visual programming influenced modern end-user

programming languages, which use visual expressions to allow nonexperts to create programs in specific domains. Finally, the oral histories demonstrate the specific influences of software engineering research (i.e., goals and techniques developed) as well as previous programming languages, on modern programming language design.

## ACKNOWLEDGMENTS

We are most grateful to Carlo Ghezzi for reading several drafts of this manuscript which led to an improved article. We also appreciate the contributions of Daniel Berry, Richard P. Gabriel, Bill Harrison, Andy Koenig, Barbara Liskov, Mike Mahoney, Mary Shaw, and Charles Weisert, their suggestions and comments. We also wish to thank the other members of the IMPACT Project team for their support of our research. Finally, we thank the anonymous reviewers for their helpful comments.

## REFERENCES

ACMCS. 1989. *ACM Comput. Surv.*, volume 21, issue 3.

AMBLER, A. L. AND BURNETT, M. M. 1989. Influence of visual technology on the evolution of language environments. *Computer 22*, 10 (Oct.), 9–22.

ARNSON, R., ROSEN, D., WAITE, M., AND ZUCK, J. 1992. *The Visual Basic How-To*. The Waite Group Press.

BAROTH, E. AND HARTSOUGH, C. 1995. Visual programming in the real world. In *Visual Object-Oriented Programming: Concepts and Environments*, M. Burnett, A. Goldberg, and T. Lewis, Eds. Prentice Hall, Manning Publications, and IEEE.

BASILI, V. R., SELBY, R. W., AND HUTCHENS, D. H. 1986. Experimentation in software engineering. *IEEE Transactions Softw. Eng. 12*, 7 (July), 733–743.

BERGIN, JR., T. AND GIBSON, JR., R. G., EDS. 1996. *History of Programming Languages II*. ACM Press and Addison-Wesley Company.

BIRTWISTLE, G. 1973. *SIMULA BEGIN*. Studentlitterature and Auerback Publishing, Inc., Lund, Sweden and Philadelphia, PA.

BOEHM, B. W. 1976. Software enginering. *IEEE Trans. Comput. C-25*, 12 (Dec.), 1226–1241.

BORNING, A. 1981. The programming language aspects of thinglab, a constraint–oriented simulation laboratory. *ACM Trans. Prog. Lang. Syst. 3*, 4 (Oct.), 353–387.

BOTTONI, P., COSTABILE, M. F., LEVIALDI, S., AND MUSSIO, P. 1997. From visual language specification to legal visual interaction. In *1997 IEEE Symposium on Visual Languages*. 234–241.

BRON, C., FOKKINGA, M., AND HAAS, A. 1975. A proposal for dealing with abnormal termination of programs. Tech. Rep. Mem 150, Twente University of Technology. November.

BURNETT, M., BAKER, M., BOHUS, C., CARLSON, P., YANG, S., AND VAN ZEE, P. 1995. Scaling up visual programming languages. *Computer 28*, 3 (Mar.), 45–54.

BURNETT, M. AND MCINTYRE, D. 1995. Visual programming. *Computer 28*, 3 (March), 14–16.

BURNETT, M., PENDSE, C. C. O., ROTHERMEL, G., SUMMET, J., AND WALLACE, C. 2003. End–user software engineering with assertions in the spreadsheet paradigm. In *International Conference on Software Engineering*. 93–103.

CARDELLI, L. AND WEGNER, P. 1985. On understanding types, data abstraction and polymorphism. *ACM Comput. Surv. 17*, 4 (Dec.), 471–522.

CARGILL, T. A. 1993. The case against multiple inheritance in C++. 101–109.

CHANG, S.-K., TAUBER, M. J., YU, B., AND YU, J.-S. 1989. A visual language compiler. *IEEE Trans. Softw. Eng. 15*, 5 (May), 506–525.

COLLINS, A. M. AND QUILLIAN, M. 1969. Retrieval time for semantic memory. *J. Verb. Learn. Verb. Behav. 8*, 240–247.

COOPER, A. 1996. Why I am called 'the father of Visual Basic'. http://www.cooper.com. Accessed December 26, 2002.

COURTOIS, P. J., HEYMANS, F., AND PARNAS, D. L. 1971. Concurrent control with rreaderss and rwriterss. *Comm. ACM 14*, 10, 667–668.

COX, P. T., GILES, F. R., AND PIETRZYKOWSKI, T. 1989. Prograph: A step towards liberating programming from textual conditioning. In *1989 IEEE Workshop on Visual Languages*. 150–156.

CRIMI, C., GUERCIO, A., PACINI, G., TORTORA, G., AND TUCCI, M. 1990. Automating visual language generation. *IEEE Trans. Softw. Eng. 16*, 10 (Oct.), 1122–1135.

DAHL, O. AND NYGAARD, K. 1967. SIMULA 67 common base definition. Tech. rep., Norwegian Computing Center.

DAHL, O.-J., DIJKSTRA, E. W., AND HOARE, C. 1972. *Structured Programming*. Academic Press.

DIJKSTRA, E. 1969. Structured programming. In *Software Engineering Techniques: Report on a Conference sponsored by the NATO Science Committees*, J. Buxton and B. Randell, Eds. Rome, Italy, 84–88.

DIJKSTRA, E. W. 1965. Solution of a problem in concurrent programming control. *Comm. ACM 8*, 9 (Sept.), 569.

DIJKSTRA, E. W. 1968a. Cooperating sequential processes. In *Programming Languages*, F. Genuys, Ed. Academic Press, 43–112.

DIJKSTRA, E. W. 1968b. Go to statement considered harmful. *Comm. ACM 11*, 3, 147–148.

DOD, U. 1980. Reference manual for the Ada programming language. In *DOD*. New York.

ELLIS, M. A. AND STROUSTRUP, B. 1990. *The Annotated C++ Reference Manual*. Addison-Wesley Publishing.

FRANKEL, M. 1993. Enabling reuse with Ada generics. In *Proceedings of the Tenth Annual Washington Ada Symposium on Ada's Role in Software Enginering, McClean, VA,* 17–30.

GANNON, J. 1977. An experimental evaluation of data type conventions. *Comm. ACM 20*, 8, 584–595.

GANNON, J. AND HORNING, J. 1975. Language design for programming reliability. *IEEE Trans. Softw. Eng. SE-1*, 2 (June), 179–191.

GHEZZI, C. AND JAZAYERI, M. 1998. *Programming Language Concepts*. John Wiley & Sons.

GLINERT, E. P. AND TANIMOTO, S. L. 1984. Pict: An interactive graphical programming environment. *Computer 17*, 11 (Nov.), 7–25.

GOLDBERG, A. 1984. *Smalltalk–80: The Interactive Programming Environment*. Addison-Wesley.

GOLDBERG, A. AND ROBINSON, D. 1983. *Smalltalk-80: the Language and Implementation*. Addison-Wesley, Reading, MA.

GOODENOUGH, J. B. 1975a. Exception handling: Issues and a proposed notation. *Comm. ACM 18*, 12 (Dec.), 683–696.

GOODENOUGH, J. B. 1975b. Structured exception handling. In *Conference Record of the Second Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*. 204–224.

GOSLING, J., JOY, B., AND STEELE, JR., G. L. 1996. *The Java Language Specification*. Addison-Wesley, Reading, MA.

GRAVER, J. O. AND JOHNSON, R. E. 1990. A type system for Smalltalk. In *Conference Record of the 17th Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*. 136–150.

GREEN, T. AND PETRE, M. 1996. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *J. Vis. Lang. Comput. 7*, 2 (June), 131–174.

GUTTAG, J. V., HORNING, J. J., AND WING, J. M. 1985. The Larch family of specification languages. *IEEE Softw. 2*, 5 (Sept.), 24–36.

HABERMANN, A. AND NOTKIN, D. 1986. Gandalf software development environment. *IEEE Trans. Softw. Eng. SE-12*, 12 (Dec.), 1117–1127.

HABERMANN, A. N. 1973. Critical comments on the programming language Pascal. *Acta Informatica 3*, 47–57.

HANSEN, P. B. 1972. Structured multiprogramming. *Comm. ACM 15*, 7 (July), 574–578.

HANSEN, P. B. 1975. The programming language concurrent Pascal. *IEEE Trans. Softw. Eng. 1*, 2, 199–207.

HANSEN, P. B. 1996. Monitors and concurrent Pascal: A personal history. In *History of Programming Languages-II*, T. Bergin and R. Gibson, Eds. Addison-Wesley, 121–172.

HARBISON, S. P. 1992. *Modula-3*. Prentice Hall.

HELM, R., MARRIOTT, K., AND ODERSKY, M. 1991. Building visual language parsers. In *ACM Conference on Human Factors in Computing Systems*. 105–112.

HIRAKAWA, M., TANAKA, M., AND ICHIKAWA, T. 1990. An iconic programming system, hi-visual. *IEEE Trans. Softw. Eng. 16*, 10 (Oct.), 1178–1184.

HOARE, C. 1969. An axiomatic approach to computer programming. *Comm. ACM 12*, 10 (Oct.), 576–580, 583.

HOARE, C. 1974a. Hints on programming language design. In *State of the Art Report 20: Computer Systems Reliabilty*, C. Bunyan, Ed. Pergamon/Infotech. This paper originated in a keynote address at the ACM SIGPLAN POPL conference in Boston in October 1973 and although it was not in the proceedings, it was distributed at the conference. A 1989 collection of Dr. Hoare's essays entitled *Essays in Computing Science* published by Prentice Hall, also contains a reprint of this paper.

HOARE, C. 1974b. Monitors: An operating systems structuring concept. *Comm. ACM 17*, 10 (Oct.), 549–557.

HOARE, C. 1978. Communicating sequential processes. *Comm. ACM 21*, 8 (Aug.), 666–677.

HOPKINS, M. E. 1972. A case for the goto. In *Proceedings of the 25th National ACM Conference*. 787–790.

HORNING, J. J. 1979. Programming languages. In *Computing Systems Reliability*, T. Anderson and B. Randell, Eds. Cambridge University Press, 109–152.

HOROWITZ, E. AND MUNSON, J. G. 1984. An expansive view of reusable software. *Trans. Softw. Eng. SE-10*, 5 (Sept.), 477–487.

ICHBIAH, J., HELIARD, J., ROUBINE, O., BARNES, J., KREIG-BRUECKNER, B., AND WICHMANN, B. A. 1979. Rationale for the design of the ADA programming language. *ACM SIGPLAN Notices 14*, 6 (June), 1–261.

IEEESW 1989. IEEE Software, issue on parallel programming.

IFIP 2004. http://www.mlaa.com.au/IFIPWG2.4/index.htm.

INGALLS, D. 1978. The Smalltalk-76 programming system: Design and implementation. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*. 9–16.

KEFFER, T. 1995. Programming with the standard template library, sage advice for coping with the stl. *Dr. Dobb's Journal 1995*, SI3.

KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C. V., LOINGTIER, J.-M., AND IRWIN, J. 1997. Aspect-oriented programming. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP)*. Springer-Verlag LNCS 1241.

KOENIG, A. AND STROUSTRUP, B. 1993. Exception handling for *C++*. In *The Evolution of C++: Language Design in the Marketplace of Ideas*, J. Waldo, Ed. MIT Press. a USENIX Association book.

LIBRARY, I. S. R. 1970. *IBM System/360 Operating System PL/I (F) Language Reference Manual*. 4th edition.

LIPPERT, M. AND LOPES, C. V. 2000. A study on exception detection and handling using aspect-oriented programming. In *Proceedings of the 22nd International Conference on Software Engineering*. 418–427.

LISKOV, B. 1993. A history of CLU. In *Proceedings of History of Programming Languages Conference (ACM SIGPLAN Notices, vol. 28, no. 3)*. 133–147.

LISKOV, B. AND GUTTAG, J. 1986. *Abstraction and Specification in Program Development*. MIT Press and McGraw-Hill Book Company.

LISKOV, B. AND SNYDER, A. 1979. Exception handling in CLU. *IEEE Trans. Softw. Eng. SE-5*, 6 (Nov.), 546–558.

LISKOV, B., SNYDER, A., ATKINSON, R., AND SCHAFFERT, J. 1977. Abstraction mechanisms in CLU. *Comm. ACM 20*, 8, (Aug.), 564–576.

LISKOV, B. AND ZILLES, S. 1974. Programming with abstract data types. In *ACM SIGPLAN Conference on Very High Level Languages*. 50–59.

LISKOV, B. AND ZILLES, S. 1975. Specification techniques for data abstractions. *IEEE Trans. Softw. Eng. SE-1*, 7–19.

LISKOV, B. H. 1972. A design methodology for reliable software systems. In *Proceedings of the Fall Joint Computer Conference*. 191–198.

LITKE, J. 1990. A systematic approach for implementing fault tolerant software designs in Ada. In *Proceedings of the Conference on TRI-Ada'90*. 403–408.

LUCKHAM, D. C. AND POLAK, W. 1980. Ada exception handling: An axiomatic approach. *ACM Trans. Prog. Lang. Syst. 2*, 2 (April), 225—233.

MACLAREN, M. D. 1977. Exception handling in PL/I. In *Proceedings of the ACM Conference on Language Design for Reliable Software*. 101–104.

MADSEN, O. L., MAGNUSSON, B., AND MOLLER-PEDERSEN, B. 1990. Strong typing of object-oriented languages revisited. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP)*. 140–151.

MCCARTHY, J., ABRAMS, P. W., EDWARDS, D. J., HART, T. P., AND LEVIN, M. 1965. *LISP 1.5 Programmer's Manual*. MIT Press.

MCILROY, D. 1976. Mass-produced software components. In *Software Engineering Concepts and Techniques, 1968 NATO Conference on Software Engineering*, J. Buxton, P. Naur, and B. Randell, Eds. 88–98.

MILLER, R. AND MYERS, B. 2001. Outlier finding: focusing user attention on possible errors. In *ACM User Interface Software and Technology*. 81–90.

MURRAY, R. 1988. Building well-behaved type relationships in *C++*. In *Proceedings of USENIX*. 19–30.

MYERS, B. A. 1990. Creating user interfaces using programming by example, visual programming, and constraints. *ACM Trans. Prog. Lang. Syst. 12*, 2 (April), 143–177.

NYGAARD, K. AND DAHL, O.-J. 1978. The development of the SIMULA languages. In *Proceedings of the History of Programming Languages Conference, ACM SIGPLAN Notices, vol. 13, no. 8*. 245–276.

PANE, J., MYERS, B., AND MILLER, L. 2002. Using hci techniques to design a more usable programming system. In *IEEE Human–Centric Computing Languages and Environments*. 198–206.

PARNAS, D. 1971. On the criteria to be used in decomposing systems into modules. *Tech. Rep. Department of Computer Science, Carnegie-Mellon University*.

PARNAS, D. 1972. On the criteria to be used in decomposing systems into modules. *Comm. ACM 15*, 2, 1053–1058.

PERRY, D. E. 1989. The inscape environment. In *Proceedings of the 11th International Conference on Software Engineering*. 2–12. Selected as best paper from 10 years ago ICSE.

PLAUGER, P. L. 1995. Standard C/*C++* the standard template library. *C/C++ Users Journal 13*, 12, 10–20.

POPEK, G., HORNING, J., LAMPSON, B., MITCHELL, J., AND LONDON, R. 1977. Notes on the design of euclid. *ACM Sigplan Notices 12*, 3, 11–18.

PRIETO-DIAZ, R. 1993. Status report: Software reusability. *IEEE Softw. 10*, 3 (May), 61–66.

RADIN, G. 1981. The early history and characteristics of PL/I. In *History of Programming Languages*, R. L. Wexelblat, Ed. Academic Press, 551–600.

RANDELL, B. 1975. System structure for software fault tolerance. In *Proceedings of the International Conference on Reliable Software*. 437–439.

RAZ, O., KOOPMAN, P., AND SHAW, M. 2002. Semantic anomaly detection in online data sources. In *International Conference on Software Engineering*. 302–312.

REDWINE, S. T. AND RIDDLE, W. E. 1985. Software technology maturation. In *Proceedings of the 8th International Conference on Software Engineering*. 189–200.

REISS, S. 1984. Graphical program development with pecan program development systems. In *ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*.

ROBILLARD, M. P. AND MURPHY, G. C. 2000. Designing robust Java programs with exceptions. In *Proceedings of the 8th ACM SIGSOFT Symposium on Foundations of Software Engineering*. 2–10.

ROSCHELLE, J., KOUTLIS, M., REPENNING, A., JACKIW, N., AND SUTHERS, D. 1999. Developing educational software components. *Computer 32*, 9 (Sept.), 50–58.

ROTHERMEL, G., LI, L., DUPUIS, C., AND BURNETT, M. 1998. What you see is what you test: A methodology for testing form-based visual programs. In *International Conference on Software Engineering*. 198–207.

ROWE, L. A. 1980. Data abstraction from a programming language viewpoint. In *ACM Workshop on Data Abstraction, Databases, and Conceptual Modeling*.

ROWE, L. A., DEUTSCH, L. P., SHAW, M., THATCHER, J. W., MAYR, H. C., ZILLES, S. N., AND HAYES, P. J. 1980. Types (discussion). *SIGMOD Record 11*, 2, 43–52.

RUBIN, F. 1987. 'goto considered harmful considered harmful. *Comm. ACM 30*, 3, 195–196.

RUBIN, R. V., GOLIN, E. J., AND REISS, S. P. 1985. Thinkpad: A graphical system for programming by demonstration. *IEEE Softw. 2*, 2 (Mar.), 73–79.

SCOTT, M. L. 2000. *Programming Language Pragmatics*. Morgan Kaufmann Publishers, San Francisco, Ca.

SHAW, M. 1981. *ALPHARD: Form and Content*. Springer-Verlag, New York.

SHAW, M. 1984. Abstraction techniques in modern programming languages. *IEEE Softw. 1*, 4, 10–26.

SHAW, M. 2001. The coming-of-age of software architecture research. In *Proceedings of the 23rd International Conference on Software Engineering*. 657–664.

SHAW, M., ALMES, G. T., NEWCOMER, J., REID, B., AND WULF, W. 1978. Comparison of programming languages for software engineering. Tech. rep., Department of Computer Science, CMU.

SHAW, M., WULF, W. A., AND LONDON, R. L. 1977. Abstraction and verification is alphard: Defining and specifying iteration and generators. *Comm. ACM 20*, 3, 553–564.

SMITH, D. C. 1977. *Pygmalion: A Computer Program to Model and Stimulate Creative Thought*. Birkhauser.

STROUSTRUP, B. 1987. Possible directions for *C++*. In *Proceedings of USENIX C++ Workshop*.

STROUSTRUP, B. 1993. A history of *C++*: 1979–1991. In *Proceedings of the History of Programming Languages Conference (ACM SIGPLAN Notices, vol. 28, no. 3)*. 699–755.

SUTHERLAND, I. E. 1963. Sketchpad: A man–machine graphical communication system. In *AFIPS Spring Joint Computer Conference*.

SUTHERLAND, W. 1966. On-line graphical specification of computer procedures. Tech. rep., MIT. MIT Ph.D. Thesis.

TANENBAUM, A. S. 1976. A tutorial on Algol68. *ACM Comput. Surv. 8*, 2 (June), 155–190.

TEITELBAUM, T. AND REPS, T. 1981. The Cornell Program Synthesizer: A syntax-directed programming environment. *Comm. ACM 24*, 9 (Sept.), 563–573.

TEITELMAN, W. 1984. A tour through cedar. In *ICSE '84: Proceedings of the 7th International Conference on Software Engineering*. IEEE Press, Piscataway, NJ, USA, 181–195.

TGS SYSTEMS. 1990. *Prograph Tutorial Manual*. The Gunakara Sun Systems, Ltd. 2nd Printing, July.

TICHY, W. F. 1998. Should computer scientists experiment more? 16 reasons to avoid experimentation. *IEEE Comput. 31*, 5 (May), 32–40.

VAN WIJNGAARDEN, A., MAILLOUX, B., PECK, J., KOSTER, C., SINTZOFF, M., LINDSEY, C., MEERTENS, L., AND FISKER, R. G. E. 1968. *Revised Report on the Algorithmic Langauge ALGOL68*. Also appeared in ACM SIGPLAN Notices, Volume 12, Number 5, 1–70, May 1977; available online at http://members.dokom.net/2.kloke/RR/rrTOC.html.

WALDO, J. 1991. The case for multiple inheritance in *C++*. *Comput. Syst. 4*, 1, 111–120.

WEGNER, P. 1984. Capital-intensive software technology. *IEEE Softw. 1*, 3 (July), 43–97.

WIRTH, N. 1971a. Program development by stepwise refinement. *Comm. ACM 14*, 2, 221–227.

WIRTH, N. 1971b. The programming language Pascal. *Acta Informatica 1*, 35–63.

WIRTH, N. 1977. Modula: A language for modular multiprogramming. *Software Practice and Experience 7*, 3–35.

WULF, W., LONDON, R., AND SHAW, M. 1976. An introduction to the construction and verification of alphard programs. *IEEE Trans. Softw. Eng. SE-2*, 4, 390.

WULF, W. AND SHAW, M. 1973. Global variable considered harmful. *SIGPLAN Notices 8*, 28–34.

WULF, W. A., RUSSELL, D. B., AND HABERMANN, A. N. 1971. Bliss: A language for systems programming. *Comm. ACM 14*, 12, 780–790.

Received September 2004; revised June and August 2005; accepted September 2005