Influences on the Design of Exception Handling
ACM SIGSOFT Project on the Impact of Software Engineering Research on Programming Language Design

Barbara G. Ryder
Rutgers University
ryder@cs.rutgers.edu

Mary Lou Soffa
University of Pittsburgh
soffa@cs.pitt.edu
3 May 2003

Abstract

There has long been a close association between research in software engineering and the design of programming languages. Part of the IMPACT project involves an exploration of the interrelations of these two fields and documentation in a report of how fundamental research in software engineering has been a valuable resource for programming language features commonly used today. The resulting report investigates the relationship by considering features in currently used languages, including exceptions, control and data abstractions, types, inheritance, concurrency and visualization mechanisms.

This paper, exerpted from the report, focuses on the influence of software engineering research on the development of exceptions. The paper demonstrates that there is a symbiotic relationship between software engineering research and the design of exception handing in programming languages. Publication of these partial results is aimed at soliciting feedback and comments from both the programming languages and software engineering communities.

1 INTRODUCTION

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 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.

To address the unique demands of application areas, thousands of programming languages have been designed. Some of them have been developed purely for research exploration, while others have been targeted for production use. In practice today, C^{++}, Java, Ada, Perl, Visual Basic, and Cobol 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 IMPACT project was designed to document the influence of software engineering research on current software practice. Part of the IMPACT project involves an exploration of the interrelations of software engineering research and the design of programming languages. The resulting report from this study documents these relationships focussing on exceptions, control and data abstractions, types, inheritance, concurrency and visualization mechanisms [29].

To perform such a study there are a number of challenges that must be addressed. The synergy between software engineering research and programming language design strengthens both fields, but also 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 in this single amorphous field. This field had the same conference publication venues (e.g., IFIPS WG2.3 formed in 1969, NATO Software Engineering conferences 1968, 1969) until the late-1970's. 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 software engineering and programming languages fields. The first in the series of International Conferences on Software Engineering was held in 1975; the first Symposium on Principles of Programming Languages was held in 1973. 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 programming language 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 and software engineering, the 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 [3] in that everyone agreed these were important ideas, but no one really knew where they originated, or who was instrumental in developing them.

Clearly, to study the evolution of the concepts and features of the many 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 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.

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 is to examine results of research as evidenced in published works and to infer influence from these results. We also have obtained short interviews with programming language designers that explore what they were thinking and how they were being influenced at the time of accomplishing their designs.

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.

This paper focuses on the influence of software engineering research on the development of exceptions. The paper demonstrates that there is a symbiotic relationship between software engineering research and the design of exception handing in programming languages.

In Section 2 the research methodology used is described. Section 3 describes the evolution of exception handling in programming languages and documents the synergy between software engineering research and programming language design. Section 4 presents a brief summary of these interactions with respect to other language features to be described in the full report. Conclusions are given in Section 5.

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. 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, 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 an influence on programming language design.

As noted in the challenges in the previous section, there have been thousands of programming languages developed and many are currently in use. A number of languages were considered as possible choices for the focus of the 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.

Visual Basic is a language 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 languages and because of its widespread use. Ada, C^{++}, and Java are widely used for general purpose applications. Thus, the focus of the study centers on the 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 enabled the tracing of historical software engineering influences.

2.1 Software Maturation Models

In 1985, Redwine and Riddle [27] presented a maturation model of software technology, which described 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 subprograms.
- **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 domains, 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 used this model to describe the *coming of age* of software architecture [30]. The

first phase, the basic research phase, involved the exploration of the advantages of specialized software structure for software process. This phase also catalogued 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, 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 use a similar maturation model to describe 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.

Phases 4 and 5 are repeated until software engineers and programming language designers find an acceptable solution.

2.2 Methodology and Resources Used

The sections of the report exploring the evolution of software engineering and programming language concepts, use this model to determine and describe the interactions of concepts in software engineering research and programming languages.

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 [31]. We used these core features to demonstrate the relation between software engineering research and programming languages. We traced in reverse chronological order, the origins of these features and their evolution. We used timelines 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.

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 communications. We also used first person interviews that were published. With the development of later languages came rationales about why language features and their forms were included; these provided another resource. Lastly, because many of the ideas that impacted on software engineering and programming languages were *in the air*, we also conducted interviews with programming language designers to gather some type of oral histories. These interviews were conducted using a fixed set of questions. 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 methodology to exceptions and exception handling as they evolved to their usage in Java, C++, and Ada.

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 which 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.

3.1 Early Error Handling Constructs

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-1950's [3, 23]. 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 an 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 ([23]).

PL/I, a programming language developed at IBM concurrently with System 360 in the mid-1960's ([25]), 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,* etc. The PL/I[5] language designers wanted to give programmers the ability to write reliable and safe programs totally in their language ([25]). 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. There were several problems with this mechanism: (i) an ON unit was associated dynamically with its invocation by an exceptional condition occurrence, rather than being associated lexically with the excepting statement or operation, and (ii) global variables were needed to communicate data to the ON unit code [18]. The construct proved difficult to use, in part because the fixup 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.

3.2 Exceptions Defined as Software Methodology

By the mid-1970's, software reliability was a strong concern in both the software engineering and programming languages communities. In March 1977, SIGPLAN, SIGOPS and SIGSOFT co-sponsored 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.

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 described them "as a means of conveniently interleaving actions belonging to different levels of abstraction" [6]. 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 Horning [4]. 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 paper [6] codified the ideas presented in his earlier paper [7] 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 contribution to exception handling, during this same year Brian Randell described his own construct for error detection and recovery [26] 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 ([11]). Horning described exception handlers as "useful" and "intended to cope with particular anticipated, but unusual, situations". Horning suggested that recovery blocks and exceptions could coexist in the same code as what he termed "an attractive compromise".

3.3 Exceptions in Programming Languages

During the period of these software engineering discussions on how programming languages should be designed to facilitate the handling of faults, Dr. Barbara Liskov and her MIT graduate students were designing a new programming language *CLU* [20, 19, 16], based on the ideas of abstraction and specification. Ten years after their initial design efforts, which were used by MIT students in courses in the late 1970's, the 1986 book entitled **Abstraction and Specification in Program Development** [17] emphasized that its goal was to teach how to construct large programs. Thus, 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 described the need to "program defensively", that is to write each procedure "to defend itself against errors" ([17]). A "robust program" is "one that continues to behave reasonably even in the presence of errors" ([17]). This is a strong emphasis on this aspect of programming language design being motivated by software engineering considerations; this fits well with Goodenough's emphases [6].

Liskov and Snyder [18] discussed the design of the exception handling mechanism in CLU and gave credit to Goodenough specifically for influencing them through his paper [6]. 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 [5]. Their discussion of the *resumption* versus *termination model* of exception handling, explicitly referenced Goodenough's model in his 1975 paper ([18]). 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 com-

[5]PL/I evolved from NPL, a precursor language.

munities [6, 7].

To summarize, CLU's exception handling mechanism is static, in that it associates a handler with the statement which may raise the exception, including a procedure call. CLU uses the termination model, which assumes the procedure that raises the exception is terminated when it passes the exception back to its caller to be handled. CLU provides a default exception handler *others*, which handles all previously unnamed exceptions for that statement; in this usage the approach in CLU to uncaught exceptions is different from that of Goodenough, who checked for these at compile-time [6]. CLU provides a language-defined *failure* exception to cover cases when a fault occurs (almost anywhere), but there is no meaningful "fixup" action.

The Ada programming language was defined in the late 1970's according to specifications set by the US Department of Defense, as a modern programming language that was to be the universal software *linga franca* for their projects. The *Rationale for the Design of the Ada Programming Language* [12] 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 [6] 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* [1] 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 programmer-controlled termination mechanism suggested 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 [32] 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 [22] and to optimize programs with exceptions. These properties led to the rejection of the resumption model which renders both of these difficult.

The Ada exception model is similar to that in CLU; exception handlers are either local or are dynamically associated with callers. However, Ada also offers the user the possibility of suppressing exceptions. The presence of tasks in Ada required some specification of how tasks may propagate exceptions to one another. The language designers wanted to "substantially limit" this possibility because of the unpredictable states of tasks which might result [12]. Most unhandled exceptions are prevented from being passed back from a child task to a parent; task execution is instead terminated. The exception *FAILURE* is allowed to be raised by one task and to be handled by another; this practice is limited to *extreme situations* only [12].

3.4 Recent Research

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 [10, 24]. Perry examined the Larch specification language [8], but preferred a non-algebraic 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 1980's 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 argued for their model of exceptions as objects (which was eventually incorporated in C^{++}) in their 1990 *USENIX C^{++} Conference* paper [14]. They also referred to the termination model of PL/I, CLU and Modula-3 [9] 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 ([2]). 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 appropriate (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 which 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 usually is halted.

More recently, aspect-oriented programming describes how cross-cutting 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 to have aspects [13] express the detection and handling of excep-

tions [15]. 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) which can be instantiated in many different locations, where exception handling requires 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 which ensures program reliability, a software engineering desiderata, using aspects, another programming language mechanism, was presented at the premiere annual software engineering conference.

As another example of the close tie between the disciplines, Robillard and Murphy discussed in their paper [28] 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 at the *TRI-Ada 1990 Conference* [21]. This technique divided software into compartments, defines precise and complete exception interfaces for each compartment and automatically verifies the conformance of the actual program to compartment specification ([28], p 5). 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.

Thus with regard to exceptions and exception handling, there has been a clear symbiotic relationship between research in the programming language design and software engineering communities, to the benefit of software practice.

4 A Taste of the Full Report

In this paper, the question of the influence of software engineering research on programming language design has been considered in the context of exceptions, a powerful feature to enhance software reliability. In the context of the full report, other language features are also examined.

Abstraction is a key notion in early software engineering research. The early work on developing, understanding and modifying programs (e.g., top-down design, structured programming, generators) influenced the design of structured control constructs. The notions of *information hiding* and *modularization* led to the development of modules as language constructs, with their defined interfaces and visibility rules. Data abstraction had an enormous influence on language design through the subsequent refinement of object oriented languages, after their introduction in Simula-67. The success of object oriented languages can be seen as linked to modularity and reliability, both strong themes in software engineering research.

Strong typing in programming languages, desirable in new language designs, was a direct answer to concerns about software reliability and correctness. Type checking satisfied the desire to have more than just syntactic checking of code. Types as data abstractions with code sharing through inheritance appeared in object oriented languages; these can be seen as providing code reuse.

Researchers in programming languages and software engineering greatly influenced the early design of concurrency features in programming languages. More recently, software engineering research has focused on tools for developing and understanding concurrent programs.

Visual programming languages appeared as an outgrowth of research in integrated programming environments by software engineering researchers. The introduction of visual expressions in programming environments was a key step in this development. Incorporation of visual expressions is now common in commercial environments. Visual programming was applied to domain-specific languages with the result being end-user programming, a new genre that allowed users not trained as programmers, nevertheless to write programs.

5 Conclusions

Programming language features including exceptions present evidence of the strong ties between software engineering and programming language research and practice. Although first introduced into programming languages early, the design of exception handling as we know it today was heavily influenced by software engineering emphasis on reliability. This paper documents specific examples of software engineering research influence on exception handling. The influence was determined through the study of published materials, presentations, and co-temporal developments in each field. Also found was evidence of an evolving symbiosis in the research relationship between these fields with regard to exceptions.

6 Acknowledgments

We gratefully acknowledge the feedback from the IMPACT project team and the programming language and software engineering communities, given through constructive comments on earlier presentations of our work.

This work was supported in part by NSF under grants CCR-00-10041 and CCR-01-37766.

References

[1] C. Bron, M. Fokkinga, and A. Haas. A proposal for dealing with abnormal termination of programs. Techni-

cal Report Mem 150, Twente University of Technology, November 1975.

[2] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley Publishing, 1990.

[3] R. Gabriel, October 2001. personal communication.

[4] J. Gannon and J. Horning. Language design for programming reliability. *IEEE Transactions on Software Engineering*, SE-1(2):179–191, June 1975.

[5] C. M. Geschke, J. James H. Morris, and E. H. Satterthwaite. Early experience with mesa. *Communications of the ACM*, 20(8):540–553, 1977.

[6] J. B. Goodenough. Exception handling: Issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, December 1975.

[7] J. B. Goodenough. Structured exception handling. In *Conference Record of the Second Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 204–224, January 1975.

[8] J. V. Guttag, J. J. Horning, and J. M. Wing. The larch family of specification languages. *IEEE Software*, 2(5):24–36, September 1985.

[9] S. P. Harbison. *Modula-3*. Prentice Hall, 1992.

[10] C. Hoare. An axiomatic approach to computer programming. *Communications of the ACM*, 12(10):576–580,583, October 1969.

[11] J. J. Horning. Programming languages. In T. Anderson and B. Randell, editors, *Computing Systems Reliability*, pages 109–152. Cambridge University Press, 1979.

[12] J. Ichbiah, J. Heliard, O. Roubine, J. Barnes, B. Krieg-Brueckner, and B. A. Wichmann. Rationale for the design of the ada programming language. *ACM SIGPLAN Notices*, 14(6), June 1979.

[13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP)*, June 1997. Springer-Verlag LNCS 1241.

[14] A. Koenig and B. Stroustrup. Exception handling for C++. In J. Waldo, editor, *The Evolution of C++: Language Design in the Marketplace of Ideas*. MIT Press, 1993. a USENIX Association book.

[15] M. Lippert and C. V. Lopes. A study on exception detection and handling using aspect-oriented programming. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 418–427, 2000.

[16] B. Liskov. A history of clu. In *Proceedings of History of Programming Languages Conference (ACM SIGPLAN Notices, vol. 28, no. 3)*, pages 133–147, 1993.

[17] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. MIT Press and McGraw-Hill Book Company, 1986.

[18] B. Liskov and A. Snyder. Exception handling in clu. *IEEE Transactions on Software Engineering*, SE-5(6):546–558, November 1979.

[19] B. Liskov, A. Snyder, R. Atkinson, and J. Schaffert. Abstraction mechanisms in clu. *Communications of the ACM*, pages 564–576, August, 1977.

[20] B. Liskov and S. Zilles. Specification techniques for data abstractions. *IEEE Trans. Softwware Eng.*, pages 7–19, 1975.

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

[22] D. C. Luckham and W. Polak. Ada exception handling: An axiomatic approach. *ACM Transactions on Programming Languages and Systems*, 2(2), April 1980.

[23] J. McCarthy, P. W. Abrams, D. J. Edwards, T. P. Hart, and M. Levin. *LISP 1.5 Programmers Manual*. MIT Press, 1965.

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

[25] G. Radin. The early history and characteristics of PL/I. In R. L. Wexelblat, editor, *History of Programming Languages*, pages 551–600. Academic Press, 1981.

[26] B. Randell. System structure for software fault tolerance. In *Proceedings of the International Conference on Reliable Software*, pages 437–439, 1975.

[27] S. T. Redwine and W. E. Riddle. Software technology maturation. In *Proceedings of the 8th International Conference on Software Engineering*, pages 189–200, May 1985.

[28] M. P. Robillard and G. C. Murphy. Designing robust java programs with exceptions. In *Proceedings of the 8th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 2–10, 2000.

[29] B. G. Ryder and M. L. Soffa. The impact of software engineering research on modern programming language features. *ACM SIGSOFT Impact Project, in preparation*, 2003.

[30] M. Shaw. The coming-of-age of software architecture research. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 657–664, May 2001.

[31] M. Shaw, G. T. Almes, J. Newcomer, B. Reid, and W. Wulf. Comparison of programming languages for software engineering. Technical report, Department of Computer Science, CMU, 1978.

[32] W. A. Wulf, D. B. Russell, and A. N. Habermann. Bliss: a language for systems programming. *Communications of the ACM*, 14(12):780–790, 1971.

‘