Hindawi Publishing Corporation
Advances in Human-Computer Interaction
Volume 2010, Article ID 602570, 26 pages
doi:10.1155/2010/602570

Research Article

On Compiler Error Messages: What They *Say* and What They *Mean*

V. Javier Traver

Computer Languages and Systems Department, Campus Riu Sec, Jaume-I University, 12071 Castellón, Spain

Correspondence should be addressed to V. Javier Traver, vtraver@uji.es

Received 20 July 2009; Revised 16 March 2010; Accepted 1 June 2010

Academic Editor: Matt Jones

Programmers often encounter cryptic compiler error messages that are difficult to understand and thus difficult to resolve. Unfortunately, most related disciplines, including compiler technology, have not paid much attention to this important aspect that affects programmers significantly, apparently because it is felt that programmers should adapt to compilers. In this article, however, this problem is studied from the perspective of the discipline of human-computer interaction to gain insight into why compiler errors messages make the work of programmers more difficult, and how this situation can be alleviated. Additionally, because poorly designed error messages affect novice programmers more adversely, the problems faced by computer science students while learning to program are analyzed, and the obstacles originated by compilers are identified. Examples of actual compiler error messages are provided and carefully commented. Finally, some possible measures that can be taken are outlined, and some principles for compiler error message design are included.

1. Introduction

One reason why high-quality software development is difficult lies in the nature of software itself [1]. To tackle the challenges of the demanding intellectual activity of software design and construction, a whole discipline, software engineering [2, 3], exists. Software engineering is devoted to principles, techniques, methods, strategies, and technologies for modeling, conceiving, managing, developing, and maintaining software systems. Object orientation [4, 5], the team software process (TSP) and the personal software process (PSP) [6], extreme programming [7], and so forth are only a few of the proposals in this line.

Focusing on the coding task, high-level programming languages have been promoted as a means of closing the huge gap in the abstraction level that exists between machine language idiosyncrasies and human thinking and language. In addition, integrated environments have been conceived to ease the editing, compilation, running, and debugging of computer programs. Visual programming techniques have also proven beneficial because they offer the programmer an easy and intuitive way to building attractive user-friendly graphical interfaces.

However, in spite of all this effort, not much has been done with compiler messages to make the life of programmers much easier. Error messages shown by compilers are, more often than not, difficult to interpret, resolve, and prevent in the future. The lack of computer support in this sense is somehow paradoxical. For instance, tools exist to help an analyst draw class diagrams; in some cases, these analysis tools even generate a basic code skeleton automatically. But, curiously, the difficulties faced by programmers, particularly those concerning compiler error messages, have not yet been addressed by mainstream compiler writers and remain a topic within the academic context. It seems to us that programmers are seen as "jack-of-all-trades", as if they could—and should—struggle against the compiler. After all, many people may think this is their job and what they are paid for. Is this a desirable situation from a software engineering point of view? Is this what programmers want? How does this affect the software development process, the product, and the programmers themselves? And what is the impact on the process of learning to program or a programming language? Is such a situation so problematic? Can it be improved? Why has this problem been almost completely neglected in the past? Questions like these and

many others may easily come to mind, and much research is still required to fully answer them. This paper focuses on analyzing the problem, how it may affect the programmers (mostly the novice ones), and how the situation could be improved.

If we look at today's state-of-the-art compilers, research and advances in the field focus on implementing new features of a programming language, or developing compilers for new programming languages. There are also efforts to improve optimization techniques, so that compiled code uses less space or runs faster. Other projects aim to develop compilers that run faster. All these are commendable and interesting topics of study, but it is striking that there is little concern on devising techniques to help the user of the system (the programmer) to do their job properly. This issue is ignored, not only in compiler textbooks (no matter how recent [8, 9] or advanced [10] they may be), but also, and most importantly, by current research directions. This trend in compiler research can easily be appreciated by looking at the table of contents of proceedings in conferences such as the *International Conference on Compiler Construction, CC*. As an example, readers can have a look at the program of the recent editions of CC, one of the conferences within the *European Joint Conferences on Theory and Practice of Software, ETAPS* (http://etaps08.mit.bme.hu/Program/progCC.html, http://www.brics.dk/~mis/CC2009) and see how many papers related to error messages there are.

While there seems not to be any formal study on why commercial compilers have neglected the area of error diagnostics, suggested ideas include the fact that higher priority has been paid to other product features such as compilation speed or the speed of the resulting executable program [11, 12], or that the developer team usually has tight schedules or does not have enough experience or the required skills to work on it [13]. In 1999, Alexandrescu, a C++ expert and developer, wrote an open letter addressed to C++ compiler vendors and the C++ community with a short proposal "to make diagnostic messages generated by C++ compilers easier to read and understand in the presence of templates" [14]. Three compiler experts from three different companies were offered the opportunity to respond to Alexandrescu's letter. They admit the problem and one of the responders, Jonathan Caves, from Microsoft Corporation, acknowledged that "[i]t is a sad but true fact that diagnostics are one of the most overlooked aspects of compiler development" and gave three main reasons for this. First, it is the historic concern about memory requirements; compilers' performance would degrade if information required for better error messages were kept. Second, compiler developers are obviously familiar with the compiler they build, and they are the ones who write the error messages themselves by using the language specification to provide a succinct description for an error situation. Unfortunately, their messages are unintelligible for the average user. In this sense, this situation in compiler development is not much different to the general case of designers who end up knowing their product so well that "[t]hey cannot think the same way as someone who does not know what they know" [15, page 220]. And the third reason has to do with how new compiler releases are planned and developed, where the list of new features is prioritized, but "better error messages" is always in the lowest priority group and then rarely addressed.

Human-computer interaction (HCI) is a discipline that aims to provide user interfaces that make working with a computer a more productive, effective, and enjoyable task. However, it seems that we forget that a compiler is also a program, and that programmers are also human beings that happen to use such programs. Why is this not taken into account? It is obvious that the efficiency and efficacy of programmers' work has an impact not only on programmers' satisfaction in the workplace and their self-esteem, but also on the software development project itself—and on the quality of the final product.

In this paper, we focus on compiler error messages. First, related literature is reviewed and the problem placed in the context of the interaction framework (Section 2). Next, we analyze why it can be hard to deal with such messages (Section 3). Examples of specific messages from four different compilers are given in Appendices A and B, where it is discussed why they are difficult, how to interpret them, and how they can be fixed or prevented. A set of general principles are suggested (Section 4) that might be used by compiler builders when designing error messages, and two broad groups of approaches that can be adopted to alleviate or solve the problem to some extent are also provided (Section 5). Finally, concluding remarks are given (Section 6).

It is worth emphasizing two positions we adopt in this paper.

(1) As was mentioned above, poorly designed error messages can affect the work of programmers and the software being built. However, we are particularly motivated to help a special group of programmers: *novice programmers* or students of computer science/engineering. In this case, the problem is not so much one of productivity and quality in software, as it is of *learning*—and teaching—programming languages. While all programmers experience similar difficulties when writing working code, obstacles faced by novices and experienced programmers are quite different.

(2) In the analysis of the problem, of the compiler errors, and of possible solutions, we keep a clear user-centered point of view. The rationale behind our arguments builds on findings in the disciplines of psychology and *human-computer interaction*. We maintain that compilers, and their messages in particular, are the interface between the machine and the programmers. We believe this perspective can shed interesting light on many aspects of the pitfalls in current compilers, and on design issues for compilers, hopefully in the near future.

The contributions of this paper are as follows. It provides an overview of the past related work which allows the identification of several still open practical and research questions about a number of interesting issues related to

compiler messages and teaching/learning computer programming. It also offers a careful analysis of the problem at the light of the HCI concepts and principles, which helps understand why error messages are poor and how this affects the programmers' coding task. This analysis is based on and complemented with a case study derived from the author's teaching experience and examples from their computer science students. From all this insight, principles of compiler error design are discussed. The paper also proposes and compares two general approaches that could be followed in the future to address the problems considered throughout the paper. The reflective nature of the paper aims also at providing additional awareness and motivation of the problem of compiler error messages and at encouraging further actions towards smarter compilers.

A simplified version of this paper was published as a conference paper in Spanish [16]. It was only 4 pages long and did not include as much detail as the current paper does. In particular, very little background and motivation material was present, the part on the related state-of-the-art work was very limited, the interaction framework was not considered, fewer error messages were included, and the analysis of the problem and the possible solutions was less exhaustive.

2. Related Work

This section reviews related work, much of it comes from the fields of Psychology of Programming, and Computer Science Education, possibly because these areas are most aware of the cognitive complexity of computer programming [17, 18], and of how novice programmers suffer particularly from it. The reviewed literature is grouped into four main categories depending on how they approach the problem of poorly designed compiler error messages. While the distinction made is not always sharp, it can help the reader in comprehending how the problem has generally been approached.

2.1. Ameliorating and Analyzing the Problem. Programming environments can be built so that the editing, execution and debugging of programs are guided by the syntax [19]. For instance, predefined templates (i.e., skeletons of language elements such as "if-then-else") can be inserted by commands, which prevents the introduction of errors. To alleviate the problems faced by novice programmers, several approaches have been explored, mostly from an educational point of view. One common idea is that of developing pedagogic environments that simplify some tasks, usually with visual interactive interfaces on top of existing languages, such as in BlueJ [20–22] or Alice [23, 24], based on Java. The visual component of these tools aims at facilitating object-oriented programming by making students interact more directly with "objects". By constraining the way users build their programs, the introduction of syntactic errors is prevented. The negative side of this is that users may not develop a good "sense of syntax" [23]. Therefore, while simplifying tools for novice use might sound a good idea, they may also have their risks. Thus, a reasonable choice is to expose them to real error messages of those tools they will later use in their professional lives [25]. Another sensible idea seems to be making the student progressively require less assistance [26], for example by turning off some assistive features, either on instructors' or students' initiative.

A typical problem students have when learning to program is that compiler error messages do not always match the student's level at a given point. Subsets of a full programming language can be used to address this situation. One example is ProfessorJ [27], designed with three levels that are gradually introduced so that students are likely to come across only messages they can understand. This is an advantage over other alternatives such as BlueJ, but some students were not happy with the constrains imposed by the subset of the language being used at a given time [27]. Furthermore, while some error messages are filtered out by disabling some features of the language, there is no guarantee that the remaining error messages will be clear and informative.

As important as trying to solve the problem is finding why the problem actually exists. In this sense, some authors explored why novices have difficulties in programming [18, 28] or make errors [29, 30]. The underlying themes in these studies have to do with which language constructs are better, or how to better teach programming, in order to cope with the knowledge gap students have, and to make them acquire good mental models [31]. These results, while interesting, tend to be too general to be directly applied in practice.

An alternative idea is to use specialized tools to log students' actions in order to easily explore their behavior while programming and compiling [32], which can provide good insight into which students are facing most problems when, and guide the instructors consequently. Recently, a tool has been proposed that not only collects actual compilation errors but also prepares reports both for instructors and students with suggestions and recommendations [33]. While all these approaches somehow analyse the problem of compiler error messages from different perspectives, in none of them the problem of poorly designed compiler error messages is explicitly addressed.

An interesting recent study [34] compares different messages styles, error types, and programming experience of the subject in terms of the time invested in correcting an error or the number of correct answers given. One of the conclusions is that longer messages do not necessarily help subjects identify the error better or faster. However, the validity of the study is questioned by its authors, and they recognize the need to look more deeply into compiler messages to really find out which aspects would help novice programmers more.

2.2. Preventing the Problem. Another sensible direction to addressing the compiler error messages consists of designing better programming languages or systems, by studying how programming languages could be made more natural so that expressing solutions to problems become easier [35, 36]. Considering usability issues when designing a new programming language is certainly a good idea, but it has

been little explored, and some existing prototype has been designed basically for children [37], and not much for professional use. Furthermore, issues on error messages or their usability are not considered.

Interestingly, some language elements of COBOL were found to be error-prone with high frequency while not being of key importance for the language (e.g., the use of commas as word delimiters), a finding that suggests their redesign in COBOL and their avoidance in future languages [38].

In the mid 1970s, Horning [39] was concerned about the adequate compiler-programmer communication. Although some of his considerations are outdated today, Horning's general characteristics that good error messages should have can still be valid. For instance, the apparently simple advice that messages should be restrained and polite is unfortunately not always followed in current compilers.

2.3. Evidence of the Problem. The general problem of poor messages in human-computer interaction has long been acknowledged. Back in the 1980s, some authors showed their concern about error message design [40] and the scarce interest paid by the HCI community to error messages [41]. Shneiderman provided guidelines of how system messages should and should not be. His recommendations were based on empirical evidence on how better messages can lead to improved user performance and satisfaction. For instance, in one of his experiments, COBOL compiler syntactic error messages were modified, and novice users were asked to fix COBOL statements. Repairing scores were found to be 28% higher with those messages with increased specificity [40]. Similarly, Brown complained about how poor error messages were in general. His tests included the analysis of the response that 15 different Pascal compilers gave to a wrong input program. He found these responses to be generally disappointing and poor and offered a few simple and somehow vague suggestions for solutions [41]. In an older study, the error diagnosis accuracy of a Cobol compiler was compared with human judgements of the true cause of the error and it was found that up to 80% of the errors were inaccurately diagnosed [38].

Nowadays, compile-time messages in Java are widely given that can be undecipherable for novice programmers who get frustrated and may waste hours on a simple error [42] or require the assistance of instructors to make any progress in their assignments [26]. To alleviate this situation, a precompiler was developed [42] to preprocess student's source code and produce more specific and (sometimes) fun messages. While no user tests were done to formally evaluate the benefits of the system, informal results were positive. For instance, qualitative assessment revealed that students not only found more informative and helpful the modified messages, but also produced better code, and instructor's workload was reduced by not wasting their time explaining the same messages over and over. Furthermore, a system was developed to collect all students' errors so that the most common errors were identified. While the general statistics that can be derived from this centralized repository can lead to helpful lines of actuation, such as rewriting the messages for the 10–20 top Java errors [43], the most commonly ocurring errors do *not* necessarily represent the most complicated to fix. Additionally, while rewriting messages is generally a good idea, there is still a risk that the new messages are not made clear enough for each individual student or programmer.

Similarly, an automated tool providing better feedback on errors of programs using a subset of Pascal language was developed and tested [11]. A survey of more than 500 students who used this tool revealed that many students become more aware of simple mistakes they made. In another survey, instructors found that their grading time decreased, and the quality of students' programs increased. However, these surveys also uncovered undesired effects such as the possibility that students become dependent on the tool rather than making them more autonomous. Again, this is in fact a risk that pedagogic tools may have if solutions are not generalized and brought to the professional domain or if particular care is not taken in the design or use of the learning tools. Reducing the support that students receive over time is a possible measure but rarely considered [26].

As part of a more general learning support tool, a database with common compiler errors with an explanation of the likely way to resolve them was developed so that students were presented with both the original messages for the compiling errors in their programs and the enhanced messages [26]. The general tool was observed to have a positive impact on the learning experience of the students participating in the study. In particular, the group of students that used the support tool were more likely to resolve syntax errors which are more complex and less common than the group of students that did not use the tool; the students using the tool also corrected the errors faster.

Therefore, evidence on the poor quality of compiler error messages and that they affect the learning process is ample (e.g., [26, 32, 34, 42–45]). While less literature exists on how the problem affects advanced or expert programmers, it is recognized the need to provide good error messages and how little research there exists on this [46, page 522]. During their practice, programmers develop skills to better deal with error messages, but this does not mean these messages are good for them either. Likely, programmers just learn how to solve the errors or just end up accepting the error messages. Actually, compiler vendors do receive user complaints on error diagnostics [14]. Low usability in the programming environments may severely affect programmers' productivity [37] and their overall user experience [13]. On the other hand, moving from novice to expert is only a slow gradual process, which can be hindered by the difficulties of error messages (among other reasons). Furthermore, experienced programmers in some language may use another compilers at some point or also be novice learners of other programming language and still have problems with the new compilers or languages in spite of their expertise [47]. Therefore, improved messages can benefit these programmers too [48], and the topic of error messages is definitely of interest both in educational and professional contexts.

2.4. Addressing the Problem. Some tools try to better inform the programmer about simple syntax errors committed due to not remembering or mistyping the name of some variable or function [45]. Basically, they find the closest matches of the wrong text to entries in the symbol table. Other authors [48] aim at producing more useful error messages and better locations for them. In their prototype, they decouple error-message generation from type-checking, which simplifies compiler construction. This kind of solution can be more general than others such as STLFilt [49], which uses regular expressions to just replace the infamous cryptic and overly long compiler error messages (those related with templates and STL usage in C++) with much shorter and more comprehensible messages. Error messages in Java when using generic programming leave also room for improvement [50].

An error recovery method which can be generated automatically given the grammar of the language is given in [13]. Despite being automatic, their experiments on 600 Pascal programs written by students suggest the method performs satisfactorily in comparison with the best system the authors knew. The errors were analyzed statistically, but no evaluation with users was performed. Only syntactic errors, not semantic ones, were addressed. Another disadvantage is that, since the method is automatic, messages are all of the type "XXX expected", "XXX expected before YYY", or "XXX expected instead of YYY", which are not as specific and informative as handcrafted ones can be.

The most representative example of how compiler design should consider compiler error messages is probably `Merr` [51], a recent prototype where the compiler writer provides code fragments as examples of errors together with their corresponding diagnostic messages. Then, the parser is run on these code fragments and the parser state is kept. Later on, when parsing an input program, the parser state will be used to identify which error message to associate with the encountered error. One of the limitations of this system is that not all possible states will necessarily have errors associated with them.

Very recently, a social recommender system [44] has been proposed with which programmers can look for how other peers have fixed similar errors. Evaluation in constrained situations with limited data (13 students working on the same set of programming problems for a total of 39 person-hours) shows that half of the queries submitted for help returned useful suggestions. A larger database of examples is required to assess the effectiveness of the tool. Additionally, its cooperative setting offers advantages and disadvantages, and opens up many other interesting research issues. However, solutions like this are implicitly accepting the error messages the way they are and try to help programmers overcome this difficulty, but do not address the problem of whether and how better messages can be produced.

To summarize, it is widely recognized that compiler error messages are generally poor and that the quality of the messages has learning and productivity effects on students, instructors and programmers. Some efforts have been made on trying to *prevent* the problem, *simplify* the programming conditions, or *analyze* programming and compiling behaviors. While interesting, many of these studies offer inconclusive findings or raise new open questions. Furthermore, they do not challenge the existing error messages, and only a few works do actually *address* the problem of producing more informative messages or diagnostics. This paper provides further awareness of the importance of compiler error messages and propose to apply the body of knowledge of the Human-Computer Interaction (HCI) as a convenient and natural way of studying the underlying problem and proposing some design guidelines for compiler error messages. This HCI perspective on analysing the compilation task has rarely (if at all) been considered, despite the insights that can be derived from it. Within the HCI discipline, the interaction framework (Section 2.5) allows us to contextualize the problem and understand what elements of the interaction existing approaches have addressed and what others have been disregarded or overlooked.

2.5. Interaction Framework. The general interaction framework (Figure 1(a)) by Abowd and Beale (as can be found in the textbook [52]) can be instantiated for the programming task (Figure 1(b)) and reveals how past efforts proposing, for example, new programming paradigms (procedural versus object-oriented) or languages (visual or textual languages) view these as articulation problems. However, much less consideration has been given to the *observation* stage, where the programmer must interpret the messages offered by the compiler within the programming environment. Consequently, the next articulation iteration (source code modification to fix the errors) becomes extremely difficult if the nature of the error is not made clear through the error message.

Therefore, much concern exists on reducing the *gulf of execution* (i.e., how well the system supports the user's goals) [53] by carefully studying and trying to improve the articulation and performance steps, while the aim of minimizing the *gulf of evaluation* (i.e., how easy the system response can be perceived by the user in terms of his/her intentions) has been generally neglected, by ignoring or paying a minor emphasis to the presentation and observation steps. However, it is known that the interaction is seriously affected if some of the four steps is not properly addressed. We think this reflects the current situation in programming language/compiler design, and this is clearly calling for an adequate treatment of these disregarded areas.

3. Problem Analysis

Anyone who has done any computer programming, however little, has faced the tedious task of, firstly, understanding what the compiler says through the error/warning messages; secondly, guessing what these messages really mean; thirdly, figuring out what to do to fix such an error/warning; fourthly, learning how to act to avoid them thereafter; and fifthly, recognizing recurring messages and remembering how they were fixed in the past.

This routine, cumbersome as it is for every programmer, is especially tedious and inconvenient for novice programmers, such as the students in their first years in our

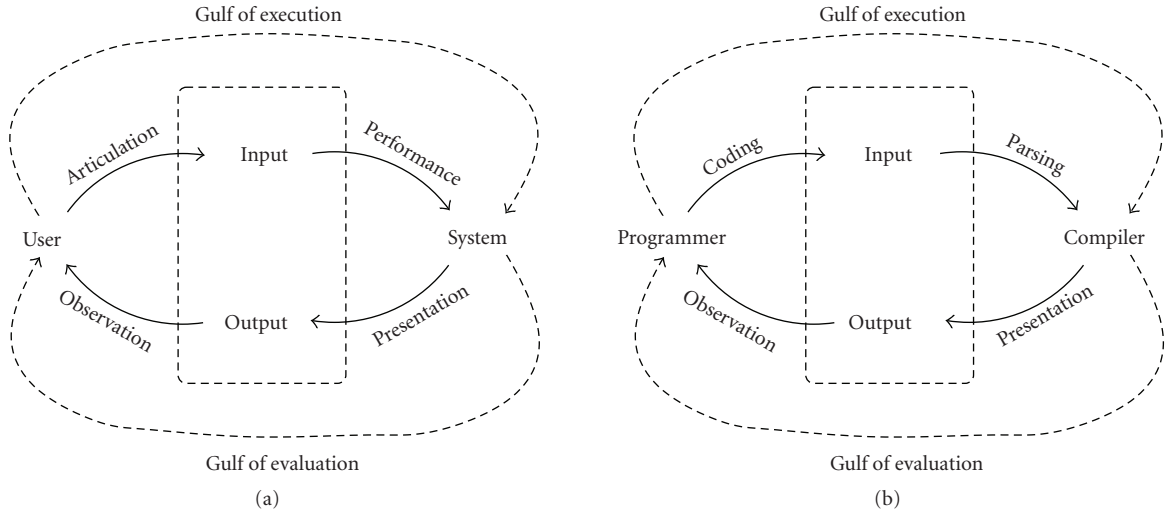

Figure 1: The interaction framework: (a) general proposal [52]; (b) our instantiation to a compiler environment. The gulfs of execution and evaluation [53] have been added.

universities. There are a number of factors contributing to this difficulty. In the following, we categorize them as originating from limitations in either

(i) training,

(ii) experience and habits,

(iii) environment,

(iv) human cognitive system,

(v) compiler.

3.1. Limitations in Training. Students are still learning the basic concepts of a programming language or programming in general, so they are still getting acquainted with the syntax, which they have not yet mastered. Compiler messages are usually in English, while many of the students (whose mother tongue is not English) have a very rudimentary knowledge of English. This, though it might seem unimportant, is a basic issue, as it does not only prevent the students from understanding some messages, but may also lead to some *misunderstanding*, which, in turn, may give rise to even worse complications. In addition, in our case, messages translated into Spanish, when such translations exist, tend to be of very poor quality and, consequently, they may harm more than help.

Another point concerns the lack of knowledge of other subjects. In particular, some knowledge on compiler construction theory is advisable. However, not all programmers can/should be assumed to have such a background. This is especially true in the case of undergraduate students, who will take advanced subjects, such as those having to do with compilers, at higher levels, that is, long after they learn to program. It is obvious that after learning how a compiler works, one better understands their limitations and, therefore, their messages. From the point of view of HCI, not having a proper mental model of how compilers work, may hinder a successful interaction with them, specially when the error messages reveal implementation details [50].

It is easy to imagine additional complications. For example, a student might understand those error messages that are within his/her current knowledge. However, it is very likely that some compiler error message is beyond their current knowledge (either programming- or compiler-related), even when the programmer's action is limited to the syntax he/she is still learning. Partly, this is so because error messages tend to be uncorrelated with the actual mistake in source code. This can be seen as an example of limitations in training (the programmer is still exploring), compiler technology (gap between actual error and message yielded), and educational tools (one can envision a compiling environment where the knowledge of the programmer is taken into account). One example of such concern is the gradual levels in Java introduced in ProfessorJ [27].

3.2. Limitations in Experience and Habits. Skills in programming are acquired with experience and practice. In addition, with time and lots of practice, some messages become familiar and you can recognize them and already know how to deal with them (provided you can remember how you fixed them in the past!). None of this happens to a novice who may encounter most error messages for the first time.

Many programmers, both novices and experienced, have the ingrained habit of not using pen and pencil when they are programming. They do not even write personal notes in a text file. However, our view is that keeping a file of messages encountered, their symptoms, their causes, and so on would be of great help. Some students do not even try to understand error messages [54] or read them [11]. Many messages are certainly difficult to comprehend, but they sometimes provide valuable clues for fixing the errors. In contrast, many programmers rapidly jump to the offending line in the source code and try to guess what is wrong. Other bad habits consist of not following good naming conventions

for data types, variables, or functions; the lack of a proper, consistent programming "style"; not paying attention to code indentation, not using good habits such as writing the closing brace '}' each time an opening brace '{' is typed in, so that it is not forgotten later, and so forth. Certainly, students tend to perceive programming style as something of little importance [11].

3.3. Limitations in the Environment/Infrastructure. In some cases, the hardware and software are not the best for the task at hand. The reasons for this may be out of the hands of the programmer, economic factors might prevent students from having faster computers, physical laboratory conditions cannot not easily be changed, installed software do not meet the desired standards, and so forth. However, on other occasions it is the programmer who does not use proper tools, available otherwise, such as language-sensitive editors with syntax highlighting options, or those editors featuring *click-and-go* (i.e., the user clicks on the compiler error message and the editor brings the cursor to the offending line of the code). As an anecdotal, real example, some of our students were reluctant to use these kinds of editors and seemed to prefer to work directly using the command shell, where compiler messages mix with user input and with output from other command-line interactions, making it difficult to visually perceive where the compiler output starts, where the first compiler error is, and so on. It would be of interest to find out whether this preference still holds and why.

3.4. Limitations in the Human Cognitive System. There are a number of issues in human cognition that have a significant impact on how the interaction between people and computers takes place. In Appendix A, we will look at the problems involved in a programming task with a particular attention to error messages, but more generally, we can mention a few here. One key aspect is that of attention: we cannot attend to every stimulus in the world at all times. On the contrary, our attention can be *focused* on a very specific stimulus, or *divided* into several streams of information [55].

While programming, there are several things that can affect our attention. Firstly, we can consider programming itself as a *primary task*, that is, the main reason why the programmer is interacting with the computer, whereas dealing with compiler error messages can be considered as a *secondary task*, which tends to distract the programmers from their primary task. The problem with these interruptions is that they break the mental concentration or line of thought, making it difficult to resume the programming task effectively after each interruption. Because it is the programmer who decides when to compile, the frequency of these interruptions can be made small so as to diminish their negative effect on attention. Unfortunately, this approach is rarely advisable: it is usually easier to fix errors with frequent compilations because of the correlation of new error messages with recent code editing. Secondly, programming sessions are usually long, which makes it difficult to maintain attention all the time.

The multistore model of human memory consisting of sensory, short-term and long-term memory stores is widely accepted. Limitations in memory, and in particular in short-term memory (the working memory), are important to consider; when looking at the list of error messages, we should keep in memory a certain image of what the code looks like. Conversely, when looking at the code, we should remember what the error message was trying to tell us. However, our capacity to hold information is limited in both amount and time, as the classic Miller's magical number 7 ± 2 suggests [56]. This is a good reason for keeping both the source code and the compiler messages visible at the same time. Finally, mental models of computers, the programming language, and the compiler should be good enough for a smooth programmer-compiler interaction. The interface itself should be responsible for supporting good mental models, letting the user rectify incorrect models and building new appropriate ones [57].

3.5. Limitations in Compilers. Compilers are built around the sound concept of formal grammars [58], and some understanding of how parsers operate reveals, in part, why error messages are limited. Technically speaking, some of these limitations might be difficult to overcome with existing compiler technology. Probably, these are precisely the barriers that compiler research should aim at breaking down. Meanwhile, however, we can only keep this in mind and live with it.

Fortunately, there are many other limitations that are not so much technical as psychological. For example, how messages are phrased is basically a matter of being aware of their importance for an effective programmer-compiler interaction. In Section 4 we suggest some error messages features that we think would be desirable for compilers to have.

It is this kind of limitation that originates from compilers, which we are now mostly interested in.

3.6. Are Good Error Messages Important? Besides the difficulty in understanding the message, poorly designed error messages have the side effect that they lead the programmers to take (sometimes many) "random" actions in the form of source code editing to try to eliminate the error situation, possibly without making an actual effort to understand the problem. This kind of behavior has also been observed by other instructors in the context of addressing compiling errors [54] and debugging programs [59]. Sometimes, when programmers run out of ideas, it may happen that they insist on compiling wrong code even without modification. Though it may appear absurd, this behavior may have an explanation from psychology: programmers, as human beings, are used to dealing with other human beings and because human-human interaction is (fortunately) not as rigid as human-machine interaction, the programmers tend to repeat their request as if, unconsciously, they were thinking that the computer might change its "mind" and give in. This could be related to the Reeves and Nass' *media equation* [60]; that is, people tend to deal with media,

computers included, as if they were real people and real places. It was evidenced throughout Section 2 how strongly the quality of error messages affect the programmer. In contrast, well-designed error messages help the programmer take (few) directed actions to correct the wrong code while also helping them to understand why the code was wrong. This can even act as a learning aid in particular to novice programmers by supporting or correcting their user models of either the programming language or the compiler itself.

3.7. Can Error Messages Be Helpful? Despite all their negative points, compiler error messages (both their contents and the logging of their appearances) can also be seen as a valuable resource.

> *For students* to clarify concepts, remove misconceptions/misunderstandings, or improve their mental models.
>
> *For educators* to discover the problems/obstacles faced by their students and to observe general/individual progression/regression.
>
> *For working professionals* to perfect their understanding, to increase their productivity, to identify areas where their training is lacking and to take appropriate measures.
>
> *For compiler writers and programming language designers* to identify which are the most common errors that programmers make, and why, and how they could be prevented.

Curiously, some poorly designed error messages may allow instructors to discover learning problems that their students have that would otherwise have gone unnoticed with a good error message that helped the student to fix their problem. An example of this is given in Appendix A. This concern of knowing the students better and discovering which problems they may have when programming is underlying many analysis and logging efforts (e.g., [32, 33, 54]). Notwithstanding this possible "advantage", the drawbacks of bad error messages go possibly beyond their benefits. A necessary caveat here would be the risk of students becoming dependent on the simplification or pedagogic tool they use [11, 61], since the students may face problems later on. This raises important research issues regarding the design of teaching/learning tools or the most suitable way of using them. These issues have not received proper attention in the past.

4. Principles for Compiler Error Message Design

Why do people commit errors? Three reasons are given by Shneiderman [62]: lack of knowledge, incorrect understanding, or inadvertent slips. It can be readily observed that all of these can be true for programmers interacting with a machine. Well-designed compiler error messages should provide help, not obstacles. In contrast, poorly-designed messages affect both novices and experts, as is well illustrated in the following quote:

> "These concerns are especially important with respect to novices, whose lack of knowledge and confidence amplify the stress-related feedback [...] Although these effects are most prominent with novice computer users, experienced users also suffer. Experts in one system or part of a system are still novices in many situations." [62]

Just as there are principles for programming language design [63], or software can be characterized by its inherent nature [1], it seems natural to think of a set of principles to guide compiler error messages design. We propose a set of desirable characteristics of messages in compilers. The proposed principles are inspired by the body of knowledge in HCI and were derived from examples of actual compiler errors and the author's experience as a programmer and as an educator. In particular, the guidelines of heuristic evaluation [64] provide good insight to define how compiler errors should be. One of these general principles behind the heuristic evaluation has just to do with providing good error messages, so that the user can easily diagnose and recover from errors. The rest of the heuristics provide rich information that can be applied to consider how the compiler-programmer interaction should be and, since most of this interaction is in the form of the compiler error messages, these other heuristics can in turn also guide the design of the error messages themselves. The following is the set of proposed principles and next to each are some heuristics they are related to the following

(i) Clarity and brevity (aesthetic and minimalist design, recognition rather than recall).

(ii) Specificity (recognition rather than recall; help user recognize, diagnose and recover from errors).

(iii) Context-insensitivity (consistency and standards).

(iv) Locality (flexibility and efficiency of use).

(v) Proper phrasing (match between system and the real world).

(vi) Consistency (consistency and standards).

(vii) Suitable visual design (aesthetic and minimalist design; error prevention).

(viii) Extensible help (help and documentation).

Notice that the relationship between our principles and the usability heuristics are richer than shown above for the sake of simplicity. For instance, "clarity and brevity" not only obeys "minimalist design" and "recognition better than recall", but also "efficiency of use": the faster the programmer reads and understand a message, the faster he/she will be able to discover the correct diagnostic and fix the problem. Other heuristics such as "user control and freedom" and "error prevention" have less immediate applicability. For instance, errors can be prevented by the nature of the programmer language or by some features

of the development environment, but not by the error messages themselves. However, it can also be argued that good error messages can help in preventing the programmer from introducing new errors since unclear, uninformative messages may lead the programmer to perform random corrections in the source code that do not really solve the problem but introduce new errors.

These principles are briefly discussed in the following paragraphs. In order to illustrate this discussion, examples from a case study are used. This study is based on a selection made by the author from compiler errors found by his students during their lab assignments in a course on advanced programming. Although all those messages came from the same C++ compiler, the messages from three other compilers for the same faulty source code snippets are also examined. See Appendices A and B for the full details.

4.1. Clarity and Brevity. Too often compiler messages are very cryptic or long and hard to decipher even for experienced programmers. This is one of the worst problems, to our mind, but happily also one of the technically easiest to solve. As pointed out by Shneiderman, "phrasing of error messages or diagnostic warning is critical" and "can significantly affect user performance and satisfaction" [62, page 305]. And if this is generally true for computer user interfaces, it is not less true for compiler error messages. Brevity is very important since it is a relatively common student behavior not to fully read error messages, which may lead the students to misinterpret or not to follow the error messages, even if they are informative and helpful [11]. In addition, more information does not necessarily mean better guidance in fixing the error [34].

4.2. Specificity. An error which is not specific may be related to a number of different diagnostics. This is the case of our example in Section A.5:

```
parse error before,
```

These too-general errors ("parse error", "illegal character", etc.) make it difficult to know what has gone wrong, and as a result, to take the proper corrective action. It is plainly frustrating to come across these simple (and simplistic) messages.

As further motivation on the importance of effective and specific messages, the reader is reminded the findings by Shneiderman (whose work was mentioned before in Section 2) of how messages with increased specificity were easier to repair. Furthermore, almost forty years ago, Weinberg [63] had already pointed out that compiler diagnostics should be more explicit. And it seems that not much progress has been made in this area in all these years.

4.3. Context Insensitivity. When errors are context-sensitive, the same problem (the diagnostic) can give rise to different error messages, depending on the context. For instance, these parts of three messages from our examples in Section A.3),

(1) `parse error before '{'`

(2) `parse error before ','`

(3) `declaration of "float SavingAccount::getInterestRate()" outside of class is not definition`

while different, all try to explain the same problem in the source code (a missing '}' to close the body a function).

Context-insensitivity can be understood as a form of robustness in the sense that the same logical error originates the same message, regardless of the particular situation the error is. Many messages are currently very sensitive to the context; in other words, for a given error, a slight change in the code nearby provokes a radical change in the diagnostic. An example of this is given in Error message 5 in Section A.3.

4.4. Locality. Locality is said not to hold in software [1]: "the symptoms of a bug can be manifested arbitrarily far away from the cause" (no spatial locality) and "arbitrarily long after the execution of the fault code" (no temporal locality). While this may be true and unavoidable for software, it should not be the case for compiler error messages. Note that our meaning for locality here has nothing to do with run-time errors. In addition, the compiler can only explain some errors by referring to any part of the code, as in the case of a function definition and a call to that function, which can obviously be arbitrarily far apart from each other. This is perfectly correct. The lack of locality we refer to here is that of the compiler *wrongly* suggesting the error is in one place when it is not actually there. It is highly desirable for programmers to count on spatial locality, that is, that the true origin of the error be where the message indicates (or as close as possible). Several examples in Appendix A (Error messages 3, 4 and 5) fail to have locality. They suggest the problem to be in one line at the beginning of a function definiton whereas the actual conceptual problem is before, since the body of the previous function definition fails to be properly closed.

One issue with the locality criterion is the *precision* in error location, which can be high or low. Many compiler error messages have very poor locality, like those in Section A.3. Some compiler error messages have locality at code-line level, but often this may not suffice, as in the example in Section A.1, where some ambiguity exists. Others (like Error message 5) have much smaller locality precision—or none at all. One typical example where the locality principles fails and usually creates confusion, to novices at least, is when the error is located in one source file, but the message points to a distinct file (usually including the former). Since the actual error and the message location are even in different files, locality is, conceptually, even lower. The problem also happens in type errors, and methods for better explaining them have been studied [65]. In general, the lower the locality of the compiler messages, the more confusing they can be for the programmer.

4.5. Proper Phrasing. There are a number of issues related to how the messages are phrased. In this respect, the general guidelines for messages in computer interfaces (e.g., [40]) can be helpful here too. Here are some of them:

(i) Positive Tone. It is crucial not to blame or condemn programmers. Current compilers do not seem to be too poor in this respect, but it is still worthwhile to strive to eliminate negative-sounding words such as illegal, invalid, bad, and so forth. As Shneiderman expresses it, "there is no excuse for hostile messages". Because programmers are so used to the language used in compiler messages, it might be difficult to draw where the boundary between positive and negative tone really is, unless the tone is extremely negative. Error message 1 in Section A.1, for instance,

```
ANSI C++ forbids declaration
‘ostream’ with no type ‘ostream’

is neither function nor method;
cannot be declared friend

parse error before ‘&’
```

speaks about something that is forbidden, which is not genuinely a positive message, and tends to blame the programmer for the error. To guarantee quality error messages, the participation of usability experts would therefore be called for, unlike the mainstream of making compiler developers write the diagnostics themselves [14]. Findings from the media equation can also be considered: a more delightful and productive interaction can be expected if criticism or blaming from the computer is replaced with praise [60, page 62].

(ii) Constructive Guidance. Compiler error messages should help programmers understand what is wrong and why. Moreover, messages should provide programmers with guidance in what they could do to fix the error. Technically speaking, it might be hard or impossible to determine with precision what the programmer's intention was [13], but alternatives, possibly based on user profile, could be suggested. As an example, all g++ error messages analyzed in Appendix A fail to provide useful suggestions or even make hard to notice where the problem actually is. For instance, the first line of Error message 2:

```
can’t initialize friend function ‘<<’

friend declaration not in class
definition
```

provides confusing information to those trying to write a definition for `operator <<`, since "initialization" is hard to understand in this context. In contrast, the second line can be more helpful, but only for programmers prepared enough to understand what it refers to, which is not usually the case of novice programmers who can hardly tell the difference between declaration and definition.

(iii) Programmer Language. Compiler messages should use simple language. Of course, because their users are technical people (programmers), these messages can make use of programming jargon, but this should be kept to the bare minimum. By no means should internal details of the compiler be given, something which is clearly violated in the example in Section A.4 (page 21):

```
/tmp/ccf2R75s.o: In function
‘Figure::Figure(int)’:

/tmp/ccf2R75s.o(.text+0x9):
undefined reference to ‘Figure
virtual table’
```

since it is referring to temporal files with meaningless names, and to a "virtual table" which has to do with how dynamic binding is implemented, an information quite unhelpful for the novice programmer.

It is often the case that compiler diagnostic messages are written from the language or compiler point of view rather than from the programmer's [11]. It is also frequent that some details of the implementation of some process is revealed in the error message, a situation which does not help programmers interpret the message, since they do not usually know such implementation [50].

(iv) Nonanthropomorphic Messages. Anthropomorphic messages are those of the kind "I can't find the prototype for this function" (a made-up compiler message). Shneiderman gives some arguments against the use of anthropomorphic messages. For instance, he argues that these messages can suggest to the user that computers can think, thus providing an incorrect model of how computers work or what their abilities are. In the context of compilers, however, this is probably less of a concern, given that the users (programmers) are computer literate and know how computers operate. However, even computer experts may be unconsciously deceived by simple computer features [60]. In particular, the language used in error messages is one opportunity for providing personality hints to users [60, page 97]. Even though we do not take this into account in the proposed alternative messages (Appendix A), we believe that this issue, the relationship of phrasing and emotions [66], and so forth, are topics that deserve further investigation.

4.6. Consistency. Consistency throughout all messages is also a desirable feature. For instance, if nonanthropomorphic messages are chosen, they should always follow this rule; if alternate actions are suggested, this should always be done in the same way; messages should always be of approximately the same length, and so forth.

4.7. Visual Design. The physical format of messages and the use of colors or different fonts, and so on, are further important considerations. These are higher-level issues, which probably go beyond compiler research and are a matter of programming environments. They are outside of the scope of this work, but their importance should not be underestimated, because "the format of the information presented is very important for the process of learning this information" [67].

4.8. Extensible Help. Above, we said that messages should be constructive and provide guidance. But we also suggested

Table 1: Characteristics that error messages should have (Horning's [39] and proposed here).

Our principle	Horning's
Clarity and brevity	concise yet distinctive
Specificity	specific
Context-insensitivity	—
Locality	localize the problem
Positive tone	restrain and polite
Constructive guidance	suggest corrections, restrain and polite
Programmer language	user-directed, source-oriented, readable
Nonanthropomorphism	—
Consistency	—
Visual design	visible pointer, standard format
Extensible help	—
??	complete

that messages should be short. To make these two aspects compatible, help provided by messages could be organized into levels: a first short message would probably be enough most of the time; if not, some brief explanation or examples can give extra assistance to the programmer. A further level could consist of a list of potential corrective actions. This would also be an example of design that accommodates users with different skills and needs (e.g., novices *versus* experts), an interesting topic within the HCI discipline. The tentative finding that longer messages do not necessarily help more [34] is relevant here too.

At this point, one can think of the possibility of automatically correcting errors. Nowadays, this would be possible only under very simple error diagnostics. But even if this were a mature technique, it is not clear whether this would benefit the programmer, because it could lead to the situation where the programmer did not learn the programming language syntax and conventions and became dependent on these compiler actions, which would be out of the programmer's control.

Some characteristics that well-designed error messages should have are discussed in [39]. These characteristics have some resemblance to the principles we suggest above, which were developed before we knew of Horning's manuscript. This similarity adds evidence of the suitability and desirability of our ideas on compiler error messages. To be more specific, Table 1 compares the principles proposed here with those Horning suggested. It can be seen how several of Horning's ideas and ours are related, although the explanation of the characteristics in [39] varies in depth and clarity. For instance, he mentions error diagnostics should be "complete", but it is unclear what he meant with it.

Horning does not provide specific examples from real compilers to illustrate the considered characteristics. Unlike ours, which are based on the output from actual compilers, his examples are general; for instance, to illustrate that symptoms should be described in a "positive fashion", he indicates that "I expect this or this, but found that" would be preferable to "missing right parenthesis". This example also indicates that he found anthropomorphic messages to be acceptable or appropriate (or he was not aware that this issue might be relevant).

Several of his suggestions regarding phrasing are all considered within our principle of "programmer language". He uses "user-directed" messages to refer to reports of problems being made in terms of user's actions, rather than the way a compiler functions. Similarly, "source-oriented" messages are those not referring to "mysterious internal representations". Finally, he uses "readable" to denote the use of user's natural language. All these three characteristics are included in our more general principle.

Regarding the localization of the error, Horning states that the line and symbol where the problem is found should be indicated. Although this is somehow related to our "locality" principle, our emphasis on the meaning and implications of the locality is higher.

For constructive guidance, Horning advice that suggestions for correcting error can be in the form "Have you checked…?", and that the suggested correction may use the repairs done by the compiler in the source code when parsing it. Horning considered visual issues in terms of the technology of that time; he mentions the use of visual information such as a pointer into the offending line as being superior to simply describing the location using text. Lastly, no reference to context-insensitivity, consistency and extensible help is found in [39].

In general, although Horning's motivation was in man-machine communication, our inspiration is more clearly in human-computer interaction, a discipline which is more developed, recognized and mature now than it was in the mid 70s.

Table 2 summarizes which of the principles considered above are violated by the error messages in our case study (Appendix A). Although this categorization is subjective to some extent, this table highlights several issues. A given error message usually infringes more than one principle. More principles violated means more difficulty for the programmer in dealing with the errors. Improper phrasing is the most common problem that compiler error messages suffer from. The particular reason that the phrasing is not adequate may be a negative tone, lack of constructive guidance, or failing to use programmer's language. The lack of suggestions for correction is frequent. At the end of Appendix A, the messages provided by four other different compilers for the same seven error conditions are given. The general poor quality of all these compilers illustrates how generalized the problem is.

5. Possible Solutions

Among the possible solutions to the problem this paper is concerned about, two broad categories, programmer-driven and compiler-driven approaches, are discussed in this section. There are other possible actions, such as having students and programmers develop better editing and

Table 2: Which principles are violated by which error messages (see Appendix A). ● indicates violation of the principle, whereas ◐ means the violation is less clear or to a lesser extent.

	Error Message No.						
Principle	1	2	3	4	5	6	7
Clarity & Brevity	●	●		●		●	
Specificity							●
Context-insensitivity			●	●	●		
Locality	●		●		●		
Proper phrasing:							
Positive tone	●						
Constructive guidance	●	◐	●	●	●	●	●
Programmer language	◐		◐	◐	◐	●	◐

programming skills and habits, or using better editors or integrated development environments. Nevertheless, however helpful these actions can be, they generally simply try to *prevent* or better live with the problem, rather than *solve* it, while the suggestions made here are aimed at actually having better error diagnostics and messages.

5.1. Programmer-Driven Approaches. Keeping a record of error messages as the programmer encounters them is a fairly simple idea, but it can be quite effective. Furthermore, it is in consonance with the ideas of the Personal Software Process (PSP) [6], that every programmer is different to all others, that it is important to do a quality job, that it is helpful to keep a number of records about your work, and so forth. Therefore, under this same philosophy, it is suggested keeping a log with messages found, their meaning, likely successful actions (updated on a historical basis) to fix the error, the number of times the message appeared or the corrective action that worked, and so forth.

It happens that some errors tend to be recurring, and some others might be infrequent, but are particularly difficult to fix. For some compilation errors, it may simply not be remembered how they were solved in the past [33]. Because of this, it is highly desirable to avoid learning lessons over and over. The suggested approach, although simple, would lead to more systematic, scientific, satisfying work, and higher productivity. In addition, some kind of error logging can be useful not only to programmers themselves, but also to instructors, managers, language designers, compiler writers [39] and other programmers [44].

To implement error logging, one can employ the conventional pen-and-paper technique, but we recommend storing the data electronically. However, even in electronic form, manual data collection is tedious and error-prone. Thus, it is not difficult to imagine software that eases the task of recording messages, searching through them, visualizing solutions, browsing them, and navigating to and back from source code, and so forth. Similar tools exist to assist in other software-related tasks; for instance, some packages to help students use PSP are mentioned in [68]. In closer relation with compiler messages, a recent proof-of-concept tool is `HelpMeOut` [44], which features some of the ideas discussed here but in a social context where programmers help each other, rather than as an individual tool.

5.2. Compiler-Driven Approaches. An alternative idea has to do with relieving the programmer of the chores of collecting new messages, browsing recorded ones, applying old solutions to the newly encountered errors, and updating the database. One possibility would be to have a high-level software layer working on current compilers. This module would be in charge of suggesting possible causes of the given messages, or even detecting the precise origin and location of the problem. The implementation of this approach could range from quite simple programs, with *ad hoc* solutions, to complete expert systems, to systems that are user-adaptable. Rather than as independent modules, or as pre- or postcompilers, these systems should, with time, be part of the compilers themselves. Of course, for these things to be possible, awareness of their importance is needed in the compiler construction community, which entails research efforts being made in this line. One limited, but encouraging system is `Merr` [51], which associates diagnostic messages with syntax errors present in examples of faulty code fragments provided by the compiler writer.

As computer systems improve and become more and more intelligent, it does not seem preposterous to imagine the application of affective computing [66] to compiler construction. For example, when the programmer is insisting on trying to compile defective code without success, current compilers just invariably show the same message again and again. On the contrary, a compiler featuring emotional aspects, could help the desperate programmer by stating the message in different ways at different moments, and by signaling alternative ways of where the problem can be and what can be done to solve it, and, in general, by trying to reassure and relieve the stressed programmer.

5.3. A Comparison. Both programmer-driven and compiler-driven approaches have advantages and disadvantages, which are discussed in this section.

Compiler-driven approaches might be considered the ideal solution, but they have drawbacks too. These tools imply a significant effort for the compiler writer. For

instance, in system like [51], the compiler developer must write both syntactically wrong code snippets and their associated diagnostic messages. Furthermore, the developer must collect a large number of code fragments containing some syntactic or semantic error. Since the number of potential errors is high, it is possible to miss important ones that will later be found (made) by the programmer. Another difficult job for the compiler writer is to invent good diagnostic messages. The problem here is that these messages should be readable and meaningful for every programmer using the compiler (or the associated tool); however, while general guidelines exist, what is considered a good message may be person-dependent. An additional disadvantage is that the system is built at *construction time*, without the possibility of growing or changing at *usage time*, unless some easy updates are possible via Internet, for example. Ideally, these updates might be based on user feedback or even exploit user-allowed logging of the programmer's activity. The positive side is that the programmer need not be concerned with the tedious task of diagnostic maintenance. Another good point is that the effort is made *only once* by one person (or a few of them, the developer team), while the resulting tool can be enjoyed *many* times by *many* people; it can therefore be very cost-effective.

In contrast, programmer-driven approaches place the burden on the programmer, who is charged with the task of creating and maintaining diagnostics. However, this is only done as needed (when the programmer encounters new error messages). Possibly, the biggest advantage is that each programmer can use their own words to phrase the diagnostics. Even though these diagnostics might lack technical correctness, they will probably be highly meaningful and explanatory for the programmer, and technical correctness would be of little concern as long as the diagnostics are informative and helpful. This is an important issue since people working on a similar code are more likely to make and fix similar errors [44], and different programmers do not agree on the utility of error messages [34]. Furthermore, unless the intentions of the programmer are known, some errors can hardly be diagnosed [13]. Therefore, all these findings call for the advisability of personalized diagnostics. As an additional feature of programmer-driven solution, new diagnostics can always be added for the same error message, and they can be edited through time, growing and evolving with the programmer's experience and knowledge. One problem would be that, at least for the most common errors, thousands of programmers would be "reinventing the wheel" time after time.

Regarding the principles for compiler error messages discussed above, both approaches can theoretically produce messages fulfilling these principles. However, specificity, context-insensitivity and locality can be harder to achieve by the more automatic methods within the compiler-driven paradigm, whereas the programmers can use their insight and experience to counteract the limitations of the automatic messages. Proper phrasing is not difficult to be achieved by compiler-driven solutions, and the phrasing of messages written by a programmer for his/her own use are expected to raise no problem regarding the usage of positive tone or programmer-centered language, and the degree of constructive guidance of these messages will depend on the ability of the programmer to do so.

From this discussion, it is quite clear that the ideal tool would have the best of both approaches. With such a tool, the writer would provide, besides error messages as well-designed as possible, a set of basic diagnostics at construction time. Additionally, at usage time, users (programmers) would be able not only to add new ones but also to edit existing ones to suit their needs and knowledge, so that the set of diagnostics evolve in parallel to their own experience.

6. Conclusions

Research in compiler design and construction seems to have neglected the very important area of compiler error messages. An overview of the literature reveals that some approaches exist that address the problem to some extent or from some point of view. However, the problem is far from being properly solved, and many open and interesting issues have been identified that deserve further investigation. The problems faced by programmers, and particularly novice programmers (e.g., computer science students), have been analyzed. Examples of actual error messages have been described, which are illustrative of how poorly designed some of these messages can be and how this affects the programmer either in their productivity or learning processes.

Another contribution of the paper is viewing the compiler error messages as the interface between the computer (the compiler in particular) and the human user (the programmer). This view leads naturally to the consideration of the human-computer interaction discipline as a source of knowledge and inspiration for both understanding the existing problems and proposing better approaches and some principles for compiler error design.

Finally, short- and long-term solutions have been suggested to tackle the problem considered in this paper, namely, that there is often a big gulf between what compiler error messages say and what they actually mean. It is our hope that state-of-the-art compilers will advance towards smart, very programmer-friendly compilers. However, because this journey may be long, more realistic approaches have been considered to help us survive in the meantime.

From our own work and from the literature review, it has become evident that there are still many open questions regarding what is best for students and for programmers, not only in terms of the compiler messages, but also concerning education strategies, learning aids, support tools, and so forth. In this respect, the tradition in HCI to validate design decisions through empirical studies involving people remains a necessary work to be done to see a systematic progress in many important related areas within the fields of software development and education.

This paper certainly has a theoretical flavor, and future work should obviously address important practical issues concerning the design of better compiler error messages, the development of useful software features mentioned in the paper, as well as testing and evaluating them with programmers, in particular with novice programmers in an

educational setting, but also with expert programmers in professional contexts.

Appendices

A. Case Study

In this section, we give examples of actual compiler error messages, which enable us to illustrate many points concerning the problems that programmers of all levels may experience. These messages are from the C++ GNU compiler (g++) [69] and were mostly collected from the interaction with our Computer Science students in laboratory sessions of the course "Advanced Programming", at Jaume I University (Castellón, Spain), when it was taught in the first semester of the 2002-2003 academic year. The motivation behind the work described in this paper is two-fold. On the one hand, we are motivated by our own personal problems encountered when programming in C++ to solve real-world problems; and on the other hand, by the appreciation that these same problems and frustrations, but amplified, can be suffered by novice programmers (such as our students). It is important to stress that the examples provided in this section are only a small sample of the many messages that make a programmer's life a little harder.

The messages discussed here correspond to the version of the g++ compiler used in that academic year. We have been using subsequent versions of the same compiler in the following years. Since the error messages for these other versions have not changed significantly, current students in this course are facing basically the same problem.

For each message, we use the following structure for our analysis:

(1) The *error message*, as given by the compiler. These are numbered so we can easily refer to them (e.g., in Table 2, page 14).

(2) A small piece of the *source code*, around the (supposedly) offending code. Lines of this code are preceded with the symbol "?" denoting "suspicious" code. The particular line of code that the compiler points to as the location of the error is underlined.

(3) The *diagnostic*, that is, a simple explanation of where the problem in the source code is and why the compiler complains. Notice that more often than not, this diagnostic cannot be derived easily and directly from the error message, but only after a lot of thought or thanks to previous experience. For those readers with some background in C++, it could be an interesting exercise to think about a possible diagnosis *after* seeing the error message and the code snippet, but *before* reading further.

(4) An *alternative error message*, which could be more appropriate, because it leads more directly to the true diagnosis of the problem. Please, notice that different people may disagree on how helpful and informative different messages are. Then, the spirit of these alternative messages is to suggest that better ones are possible, without claiming the ones provided are absolutely good.

(5) A *comment* about why the message error is difficult, or confusing or problematic, which principles of human-computer interaction seem to be violated or ignored, and what could be done about it.

The code snippets shown here are deliberately simple, because we want to focus on the error messages and the context giving rise to them. For a clearer presentation, the example error messages are introduced under headings summarizing what the main problem with those messages is. These headings are:

(i) Unclear, not-to-the-point messages

(ii) Misleading messages

(iii) Same logical error, different error messages

(iv) Internal-detail messages

(v) Same error messages, many possible logical errors.

A.1. Unclear, Not-to-the-Point Messages

Error Message 1*:*

```
ANSI C++ forbids declaration
'ostream' with no type 'ostream'

is neither function nor method;
cannot be declared friend

parse error before '&'
```

Offending Code:

```
?class SavingAccount {
?friend ostream & operator<< (ostream
    &os, const SavingAccount &sA);
? };
```

Diagnostic: The problem is just that the programmer forgot to include the header file `iostream.h`, thus the compiler does not know what `ostream` is.

Alternative Message:

```
I do not know what 'ostream' is.
Perhaps you forgot to include a header
file (maybe 'ostream.h')
```

Comments: The reader may think that the original error message is not that difficult. And it is not, but only once you have found it and solved it several times. Another issue is the last part of the message: `parse error before '&'`. But, which `'&'` does this refer to? There are three `'&'` symbols in the same line of code and the programmer might wrongly focus on one of them without realizing there are others.

Regarding the suggested alternative, it is not difficult nor unreasonable for the compiler to suggest what the missing header is, given the operator (<<) that is being declared, and

the fact that the well-known `ostream` keyword is present in the source code.

Notice that the alternative message is anthropomorphic in that the compiler seems to be alive and is able to speak and think ("I do not know..."). We briefly discuss this issue in Section 4. Now, our concern is not about the suitability or not of anthropomorphic messages or the way the messages address to users ("Perhaps *you* forgot..."). More important than this is to convey a clear message that the programmer can quickly understand and that is useful for fixing the error.

The proposed alternative message clearly states what specifically the compiler does not understand or has problem with (the word `ostream`) and does not provide confusing or ambiguous explanations. Additionally, a suggestion of how to fix the error is given in terms of a possible diagnostic (a header file not being included). No compiler-oriented terms are used; the only programmer-oriented, "header file", is simple, and it is reasonable to expect the programmer to understand. The name of the likely header file which is missing (`'ostream.h'`) provides further help even if a novice programmer did not know what a "header file" is. It can be argued that the message is blaming the programmer by saying "you forgot", but the full message is polite and says "Perhaps you forgot". In any case, the alternative messages suggested in this section are intended to be illustrative, rather than definitive solutions.

A.2. Misleading Messages

Error Message 2:

```
can not initialize friend function '<<'
friend declaration not in class
definition
```

Offending Code:

```
? friend ostream & operator<<
  (ostream &os,const SavingAccount & sA){
?  os << ''this is a saving account'';
?   return os;
? }
```

Diagnostic: The keyword `friend` is used inside the declaration of a class *giving* the friendship to a particular function, not when defining that function. It is easy to make this mistake because one usually takes the header of a function from its declaration, possibly by copy and paste. The error message is easier to understand once one understands not only the rule stating "friendship is given, not taken", but also who gives the friendship to whom.

Alternative Message:

```
The keyword friend should not be
here (just remove it)

Remember: friendship is granted, not
taken
```

Comments: The first part of the original message is certainly misleading: what is the compiler trying to "initialize"? The second part, though, has at least something to do with the true diagnostic. But would programmers notice or read this second part or would they just strive to understand the first sentence first?

The proposed alternative message states very directly that the keyword `friend` should not be where it appears, and it clearly indicates the required action to fix the error (remove this keyword). The second line provides a short, gentle reminder of why `friend` is not correct here. If a programmer needed more explanations, the message would be extended by additional help levels. For instance, the next level could briefly explain the implication that declaring a function `friend` within the definition of a class has and why it does not make sense to do it outside the class.

A.3. Same Logical Error, Different Error Messages.

Error Message 3:

```
In method 'float
SavingAccount::getInterestRate()':
parse error before '{'
```

Offending Code:

```
? float SavingAccount::getInterestRate()
?  {
?  return rate;
?
? SavingAccount::SavingAccount() {}
```

Diagnostic: In this and the following two examples, the cause of the error is the same: a missing `'}'` for closing the body of a preceding function (here, `getInterestRate()`). However, the error messages differ depending on the particular place where this character is missing.

Alternative Message:

```
A function declaration inside a
function body is not possible.

Did you forget '}' to close the body
of the previous function definition?
```

Comments: Because these sample code fragments are simple, the source of the problem can be quite easily identified. However, this is not necessarily always the case, and when code in the body of a function is longer, and it has several pairs of braces, `'{'-'}'`, and indenting is not done properly, it is not rare to miss some `'}'` that matches a previous `'{'`, without realizing it is missing.

In such circumstances, the current error message may not be very helpful. Here the problem lies in the lack of precision locating the source of error. Programmers tend to look for

potential mistakes locally, probably in the same line or the line just before the offending line. Fortunately, in this error message there is a good clue to help us find out where the problem is: it seems that the error is found while parsing `float SavingAccount::getInterestRate()`, which is suspicious considering where the offending line is.

However, people tend not to read carefully (particularly on screens), and programmers will often pay special attention to the location of the error rather than to details of the message text [11]. One additional reason that programmers do not pay more attention to the error message is probably because most messages are not meaningful or helpful. Therefore, programmers develop the automatic habit of ignoring them and trying to make sense of the error just by looking at (or near) the offending line.

The first part of the proposed error message indicates what could be happening if the compiler tries to make sense of the source code. Since this may not be fully helpful for the programmer, this speculation is followed by a plausible interpretation of what may have happened. The message uses "function declaration", "function body" and "function definition". Some user tests would be required to find out whether these terms are understood or require some redesign, and whether further help levels could be available to clarify them.

Error Message 4:

```
In method 'float
SavingAccount::getInterestRate()':
parse error before ','
```

Offending Code:

```
?float SavingAccount::getInterestRate()
  {
?return rate;
?
?SavingAccount::SavingAccount(string
owner_, float initialBalance)
?       : owner(owner_),
balance(initialBalance)
?{}
```

Diagnostic: The same as in Error 3.

Alternative Message:

```
A function declaration inside a function
body is not possible.
Did you forget '}' to close the body of
the previous function definition?
```

Comments: The puzzling point here is that one could try to see what is wrong just before the comma (,). Typically this message is issued when a type name has been misspelled, or a necessary header has not been included. For instance, in this line of code, one may wonder whether the compiler recognizes `string`, whereas the actual problem is logically far from this (again, a missing '}' in the body of the previous function).

Error Message 5:

```
In method 'SavingAccount::SavingAccount
()':
  declaration of 'float
SavingAccount::getInterestRate()'
  outside of class is not definition
```

Offending Code:

```
? SavingAccount::SavingAccount() {
?
? float SavingAccount::getInterestRate()
  {
?       return rate;
? }
```

Diagnostic: The same as in Error 3.

Alternative Message:

```
A function declaration inside a function
body is not possible. Did you forget
'}' to close the body of the previous
function definition?
```

Comments: The error message here is quite different to (and longer than) the other two. The sentence combines the concepts of *declaration* and *definition* that not every programmer (especially novices ones) may clearly distinguish, or at least not without an extra, conscious, mental effort. Good human-computer interfaces are designed to minimize the required mental effort. In particular, programmers have enough work with their demanding *primary task* (programming itself), without also having to deal with an annoying *secondary task* (understanding compiler error messages and fixing the underlying problems). (The topic of primary and secondary tasks is covered in [55, page 105])

A missing '}' in a definition of a member function inside its class ("inline" definition) yields another error message:

```
parse error at end of input
```

with the offending line being a line beyond the last line in the file. The spatial locality principle does not hold here: the source of the problem in the code may be arbitrarily far from the offending line (the end of file!). Of course, spatial locality can be forced (facilitated) by a well-known trick: divide & conquer. We mean that one can, selectively, get rid of pieces of code so that the wrong code is eventually found. But here it is probably faster to use the temporal locality: where has the programmer been editing recently (i.e., since the last time the source file was compiled)? But most novice programmers do not know either of these tricks (spatial and temporal locality).

A.4. Internal-Detail Messages

Error Message 6:

```
/tmp/ccf2R75s.o: In function
‘Figure::Figure(int)’:
/tmp/ccf2R75s.o(.text+0x9): undefined
reference to ‘Figure virtual table’
```

Offending Code:

```
?class Figure {
?      private:
?            int color;
?      public:
?            Figure(int c = 0);
?            virtual float Perimeter();
?};
```

Diagnostic: The problem is that the virtual member function `Perimeter()` is not made *pure* virtual, like this:

```
virtual float Perimeter() = 0;
```

and no definition for it was provided.

Alternative Message:

```
If class ‘Figure’ is to be abstract,
some member function should be declared
pure virtual (with ‘= 0’)
```

Comments: The message considered here is actually a *linking*, not compiling, error message. These errors are generally more difficult to resolve than compiling errors, partly because they do not refer to specific lines in the source code. In fact, in the code fragment shown above, we marked as the offending line the constructor, guided (or better, misguided) by the error message, which refers to this constructor. We later may learn that the problem is not in the constructor, but in a member function that was declared virtual, but not *pure* virtual.

We think that another reason why linking error messages are harder to decipher is because they arise as a result of putting together pieces of code that, separately, are correct. This is because, unlike in other disciplines, in software, errors are synergistic [1].

Obviously, one may use `virtual` functions before (or without) learning how they are internally implemented. This is, in fact, the common situation of undergraduates learning programming in their first years. Here, abstract classes and run-time polymorphism are difficult concepts by themselves for a novice programmer to understand, without also having to struggle with how they are implemented. Hiding the complexities of underlying internals is a very useful means of minimizing the mental load associated with using a system [57]. Speaking the users' language is also a well-known usability heuristic [70].

This is a clear example where having a *functional* mental model should be enough for the task at hand. Of course, having a *structural* mental model would help in understanding the error message and its source in the code. But why assume the programmer has this structural mental model? For convenience, the distinction of functional and structural models is summarized here. Functional mental models of a system are basic knowledge users have that allows them to operate that system by relying on simple facts and possibly on analogies with known systems. Structural mental models are more elaborate since they involve knowledge of how the system actually works, so these models are harder to build but they are more powerful than functional ones, because they allow the users to make predictions (e.g., what might be wrong when the systems does not work) [55, pages 134–137].

It is possible that this message reminds the reader, as it reminds the author, of some infamous error message in some window-based systems stating something along the lines of "Fatal error occurred. Stack addresses 0x076AC-0x0FFD8". This kind of message and ours have two things in common: reference to strange-looking, useless internal data (temporary files like `/tmp/ccf2R75s.o`), and implementation details (something about a virtual table is mentioned, but what does it have to do with our code?) Both of these things are almost unhelpful in understanding and solving the problem, and add unnecessary complexity and confusion for the programmer.

This error message can easily be turned into one in Section A.3, that is, different messages being produced for the same logical error. Surprisingly, in the case we are now considering, this error message does *not* appear if the constructor is defined within the class definition, rather than outside.

The proposed alternative predicts one possible programmer's intention (having class `Figure` abstract) and suggest its implication (declaring at least on function pure virtual). The message reminds the programmer the syntax to use (`= 0`) to declare a function *pure* virtual. Subsequent levels of the message would suggest alternative diagnostics and corrective actions; for instance, the programmer could have just forgotten to write the body function for `Perimeter()`.

A.5. Same Error Message, Many Possible Logical Errors

Error Message 7:

```
parse error before,
```

Offending Code:

```
?SavingsAccount::SavingsAccount(string
  owner, string IdNum,
?  Date  openDate, float rate)
?     : BankAccount(string owner, string
  IdNum, Date openDate);
?{
?  //...
?}
```

Diagnostic:

> The programmer repeated the declaration of the data types for the arguments of the base-class (`BankAccount`) constructor in the initialization list of the derived-class (`SavingsAccount`) constructor. The right initialization list is:
>
> ```
> : BankAccount(owner, IdNum, openDate);
> ```

Alternative Message:

> ```
> Do not declare types in the
> initialization list (i.e., what follows
> ':')
>
> Use calls instead, such as
> 'BankAccount(owner, IdNum, openDate)'
> rather than 'BankAccount(string owner,
> string IdNum, Date openDate)'
> ```

Comments: `Parse error before...` is probably the most general error message we can think of. It certainly says little about the nature of the error. Therefore, the programmer can work hard and long trying to find out where the error actually is. This kind of messages can be very confusing and frustrating [11].

At first glance, one may argue that this is a silly, unlikely mistake to commit and even once committed, it is not that difficult to discover where the problem is. This may certainly be true for experienced programmers, but not for a novice programmer such as the one who actually wrote this code. Indeed, the writing of this incorrect code reveals a lot about its author. Probably, he did not fully understand how to declare a derived-class constructor. For this same reason, he would find nonsensical that the compiler complains about this line of code. For him, it is perfectly error-free C++ syntax.

But there is still more that we can learn from this example. A more expert programmer whom the (novice) author asked for help might also find it difficult to find out what is wrong in this line. In fact, in one of our lab classes, one of our students asked me to help him on this error message. Even though I knew that such a line of code was incorrect (because the compiler said so), I made several unsuccessful attempts, and I did not realize what the problem was until I tried to write the derived-class constructor myself. Only when comparing what he wrote with what I wrote did what was wrong in the student's code become evident. This resembles an observed debugging strategy where parts of the code are simply rewritten to solve the problem instead of trying to find and fix the bugs in the original code [71].

This phenomenon can be related to the psychological known fact that our perception is not fully objective, but it is sometimes biased by our expectation: both my student and I were looking at the code with the confidence that it was right. The following words of Weinberg give further support to the fact of "the perfectly normal human tendency to believe that ones "own" program is correct in the face of hard physical evidence to the contrary" [63, page 56].

Of course, because the compiler was complaining, it was for certain that there was some error somewhere in the code. But where it could be? Our attention was divided among several lines (we already know about the unfortunate fact of the lack of locality in the *actual* error and the *signaled* error). With so many potential logical errors behind this textual message, we could not focus on the appropriate point. "When a programmer has a difficult time finding a bug, it is because he is looking in the wrong place" [63, page 251].

Needless to say that a more focused message error would definitely have helped. In this case the student could probably have fixed the problem without my intervention. From a more positive point of view, a poorly designed error message such as this allowed us to realize one of the learning problems our student was having, as was suggested in Section 3.7.

Another lesson here is that it might be a good idea to ask for the help of another person, and to rewrite a piece of code guided by *what you want* to program, not by *how you are trying* to do it (i.e., without looking to the original code).

With regard to the feasibility of a compiler to generate the alternative message proposed here, we think a general solution might be more complicated, but an *ad hoc* solution is not very difficult in terms of checking whether the list of arguments is as it should be expected or not. In this sense, the proposed message direct the programmer at the problem (the use of declarations within the initialization list) and at the solution (the use of calls). The programmer is also reminded the initialization list is what follows the " : ". The third line of the proposed alternative makes the message longer, but it gains in clarity since it makes more explicit the difference between what the programmer wrote and what is expected in the initialization list.

B. Messages with Other Compilers

To illustrate that the problem is not limited to the particular compiler and platform (`g++` in Linux) considered here, three other C++ compilers were tested for the same error conditions analyzed above. The compilers were another free-software compiler, MinGW within Dev-C++ 4.9.9.2 [72], and two commercial ones, Microsoft Visual C++ 6.0 [73] and Borland C++ Builder 5.0 [74] which were accessible to the authors. The error messages as provided by the four compilers are collected in Tables 3 and 4, and referred to as `g++`, `MinGW`, `MVC++` and `BC++`, respectively. For convenience, minor parts of the messages related to the names of the source files and line numbers have been removed. When the messages took up many lines, some of its lines have been omitted (this is indicated in the tables). Finally, in other examples, the line as shown by the compiler was too long to fit one row in the table, and it has been split into two lines.

In the following subsections the messages given by each of the four analyzed compilers for each of the seven error situations are commented. Before going into detail, a glance at these tables reveal the violation of basic usability principles, as discussed in this paper. Notice for instance how the messages use words such as "illegal" or "improper", which are implicitly blaming the programmer, whereas it is advised that system messages should prevent such offensive

Table 3: Error messages of four different C++ compilers for Errors 1–4.

Compiler	**Message for Error 1**
g++	ANSI C++ forbids declaration ‘ostream’ with no type ‘ostream’ is neither function nor method; cannot be declared friend parse error before ‘&’
MinGW	ISO C++ forbids declaration of ‘ostream’ with no type ‘ostream’ is neither function nor member function; cannot be declared friend expected ‘;’ before ‘&’ token
MVC++	error C2143: syntax error: missing ‘;’ before ‘&’ error C2433: ‘ostream’: ‘friend’ not permitted on data declarations error C2501: ‘ostream’: missing storage-class or type specifiers (plus 4 more lines)
BC++	E2061 Friends must be functions or classes E2139 Declaration missing; E2321 Declaration does not specify a tag or an identifier
Compiler	**Message for Error 2**
g++	can’t initialize friend function ‘<<’ friend declaration not in class definition
MinGW	can’t initialize friend function ‘operator<<’ friend declaration not in class definition
MVC++	error C2255: ‘<<’: a friend function can only be declared in a class
BC++	E2092 Storage class ‘friend’ is not allowed here
Compiler	**Message for Error 3**
g++	In method ‘float SavingAccount::getInterestRate()’: parse error before ‘{’
MinGW	In member function ‘float SavingAccount::getInterestRate()’: expected ‘;’ before ‘{’ token
MVC++	error C2143: syntax error: missing ‘;’ before ‘{’
BC++	W8066 Unreachable code E2379 Statement missing; (plus 2 more lines)
Compiler	**Message for Error 4**
g++	In method ‘float SavingAccount::getInterestRate()’: parse error before ‘,’
MinGW	In member function ‘float SavingAccount::getInterestRate()’: expected primary-expression before ‘(’ token (plus 2 more lines)
MVC++	error C2275: ‘string’: illegal use of this type as an expression c:\program files\[...]\include\xstring(612): see declaration of ‘string’ error C2146: syntax error: missing ‘)’ before identifier ‘owner_’ error C2059: syntax error: ‘)’
BC++	W8066 Unreachable code E2108 Improper use of typedef ‘string’ (plus 4 more lines)

Table 4: Error messages of four different C++ compilers for Errors 5–7.

Compiler	**Message for Error 5**
g++	In method ‘SavingAccount::SavingAccount()’: declaration of ‘float SavingAccount::getInterestRate()’ outside of class is not definition
MinGW	In constructor ‘SavingAccount::SavingAccount()’: expected primary-expression before ‘‘float’’ expected ‘;’ before ‘‘float’’
MVC++	error C2601: ‘getInterestRate’: local function definitions are illegal fatal error C1004: unexpected end of file found
BC++	E2089 Identifier ‘getInterestRate’ cannot have a type qualifier (plus 3 more lines)
Compiler	**Message for Error 6**
g++	/tmp/ccf2R75s.o: In function ‘Figure::Figure(int)’: /tmp/ccf2R75s.o(.text+0x9): undefined reference to ‘Figure virtual table’
MinGW	C:\...\Temp\ccqwbaaa.o(.text$_ZN6FigureC2Ei[Figure::Figure(int)]+0x8) In function ‘ZSt17__verify_groupingPKcjRKSs’: [Linker error] undefined reference to ‘vtable for Figure’
MVC++	error6.obj: error LNK2001: unresolved external symbol ‘‘public: virtual float__thiscall Figure::Perimeter(void)’’ (?Perimeter@Figure@@UAEMXZ) Debug/error6.exe: fatal error LNK1120: 1 unresolved externals
BC++	[Linker Error] Unresolved external ‘Figure::Perimeter()’ referenced from C:\DOCUMENTS AND SETTINGS\[...]\ERROR6.OBJ
Compiler	**Message for Error 7**
g++	parse error before,
MinGW	In constructor ‘SavingsAccount::SavingsAccount(std::string, std::string, Date, float)’: expected primary-expression before ‘‘owner’’ expected primary-expression before ‘‘idNum’’ expected primary-expression before ‘‘openDate’’
MVC++	error C2144: syntax error: missing ‘)’ before type ‘string’ error C2612: trailing ‘#’ illegal in base/member initializer list error C2512: ‘BankAccount’: no appropriate default constructor available error C2082: redefinition of formal parameter ‘owner’ error C2146: syntax error: missing ‘;’ before identifier ‘idNum’ (plus 4 more lines)
BC++	E2108 Improper use of typedef ‘string’ E2312 ‘string’ is not an unambiguous base class of ‘SavingsAccount’ E2312 ‘Date’ is not an unambiguous base class of ‘SavingsAccount’ E2251 Cannot find default constructor to initialize base class ‘BankAccount’

Table 5: Which principles are violated by which compilers for error message 1.

	Compiler			
Principle	g++	MinGW	MVC++	BC++
Clarity & Brevity	●	●	●	●
Specificity				
Locality	●	●	●	●
Proper phrasing:				
Positive tone	●	●	●	
Constructive guidance	●	●	●	●
Programmer language	◐		◐	◐

Table 6: Which principles are violated by which compilers for Error message 2.

	Compiler			
Principle	g++	MinGW	MVC++	BC++
Clarity & Brevity	●	●	◐	◐
Specificity				
Locality				
Proper phrasing				
Positive tone				◐
Constructive guidance	◐	◐	◐	◐
Programmer language				◐

language. The use of cryptic terms is particularly evident in messages for Error 6. It can be seen that different compilers provide (quite) different messages for the same error situation and, while some messages might be argued to be better than others at helping to diagnose and fix the problem, no compiler can be regarded as providing generally good messages.

Therefore, the problem of poor error messages is very general, with minor differences between different compilers of a same programming languages. Furthermore, as it has been seen at the beginning of this paper, the problem is present in other programming languages, even in popular ones such as Java.

Error 1. Many messages form MinGW are similar to those by g++. For instance, in this error message, MinGW uses "member function" instead of "method". Given their similarity, the principles violated by MinGW and g++ are basically the same. Notice, however, that unlike g++, MinGW does not use the term "parse" which is more a compiler term than a programmer term. Therefore, this aspect of the phrasing is slightly better in MinGW.

Both MVC++ and BC++ use error codes (e.g., C2143, E2061) which are not obviously or immediately useful for the programmer. This additional information is against minimalist design, and reduces the visibility and clarity of the remaining information.

MVC++ tries to interpret the unknown symbol "ostream" as if it was a data member the programmer was trying to declare. Under this interpretation, it can be understood its message stating that friend is not permitted on data declarations, and that some type specifier is missing. Since the compiler does not know what the programmer's intention is, it is misinterpreting the correct diagnostic. As a result, this wrong compiler's interpretation is likely to confuse the programmer, who expects the compiler to help rather than distract.

The message given by BC++ is similarly unwise. Additionally, "not permitted" fails to be a positive way to address the compiler user. Both MVC++ and BC++ use terms which novice programmers and many advanced ones can easily not know, such as "storage-class" and "tag".

Which principle is violated by each compiler is given in Table 5. By definition, context-insensitivity can only be analyzed by changing the context of a given error and observing whether the resulting message is the same. Since this is done in Error messages 3–5, this principle will only be considered for those errors. Locality is marked to be violated in all cases in the sense that the problem is not actually in the line where ostream is used. Since the problem is a missing header file, ideally the compiler should refer to this problem as a general one and use this line only as a place where the need of such header file is found, not where the problem is. Otherwise, the programmer is wrongly induced to make any corrections on this line. We judged specificity not to be violated by these error messages in the sense that they make specific (not general or vague) claims, even in a wrong direction.

Error 2. Again the message form the MinGW compiler is very similar to g++, with the only difference of referring to the function operator<<, instead of referring to function <<.

Table 7: Which principles are violated by which compilers for Error message 3.

	Compiler			
Principle	g++	MinGW	MVC++	BC++
Clarity & Brevity				
Specificity				
Context-insensitivity	●	●	●	●
Locality	●	●	●	●
Proper phrasing:				
Positive tone				
Constructive guidance	●	●	●	◐
Programmer language	◐			

Table 8: Which principles are violated by which compilers for Error message 4.

	Compiler			
Principle	g++	MinGW	MVC++	BC++
Clarity & Brevity	●	●	●	●
Specificity				
Context-insensitivity	●	●	●	●
Locality				
Proper phrasing:				
Positive tone			●	●
Constructive guidance	●	●	●	●
Programmer language	◐	●		

Messages from MVC++ and BC++ seem less confusing since they do not refer to any strange "initialization" (as g++ and MinGW do), and focus on saying where the keyword `friend` may go ("can only be in", in MVC++) or where it cannot be ("is not allowed here", in BC++). The tone is more positive in MVC++ than in BC++, but both messages could provide, possibly as extended help on user demand, some brief rationale of why `friend` makes no sense in that line. The advanced programmer can easily fix the problem (i.e., just remove `friend`) given these messages, but the novice would face some problem in understanding what is wrong, and the terms "declared" (in MVC++) and "storage class" (in BC++) would possibly harm more than help. A more direct guidance, just suggesting the deletion of this keyword, would probably be beneficial. Table 6 summarizes which message violates which principle.

Error 3. Since the same error condition (a body function with a missing } to indicate the body end) has been tested on two other different situations (Errors 4 and 5), and the resulting messages are different in each case, all the tested compilers fail to have context-insensitivity. Locality is weak since the compilers suggest the problem to be at the beginning of the next function. MinGW and MVC++ indicate ; is missing before { probably because the compilers would expect a function call at that position, rather than the declaration of a new function. This is also the case of BC++ but this compiler issues an additional warning, "unreachable code". In all cases, the compiler is making one possible interpretation of the faulty code which is different to the actual user intention. In other words, the programmer forgot to close one function definition whereas the compiler makes the assumption that the programmer's idea was to complete the body of the function. This different interpretation is a clear example of a compiler-oriented message, and it may confuse the novel programmer. With practice, experienced programmers learn to "translate" these compiler interpretations into the correct diagnostics. The warning of unreachable code provided by BC++ could probably be more helpful for the programmer to find where the error is, if the programmer is able to notice that the unreachability is caused by the `return` statement, something which could readily be made more explicit by the compiler. The violation of principles by each compiler is given in Table 7.

Error 4. For the same fault in the source code as before (an unclosed body function), the messages are very different and considerably less helpful, except for the warning issued by BC++ regarding the unreachable code which may provide some hint on the actual problem. It can be noticed how a negative tone in the phrasing of error messages is used in both MVC++ ("illegal use") and BC++ ("improper use"). A difficult term for an average programmer to understand is "primary-expression" (used by MinGW). Table 8 summarizes the principles being violated.

Error 5. The diagnostics from the compilers are again in the wrong direction, and no appropriate guidance is provided

Table 9: Which principles are violated by which compilers for Error message 5.

	Compiler			
Principle	g++	MinGW	MVC++	BC++
Clarity & Brevity				
Specificity				
Context-insensitivity	●	●	●	●
Locality	●			
Proper phrasing:				
Positive tone			●	
Constructive guidance	●	●	◐	●
Programmer language	◐	◐	◐	◐

Table 10: Which principles are violated by which compilers for Error message 6.

	Compiler			
Principle	g++	MinGW	MVC++	BC++
Clarity & Brevity	●	●	●	
Specificity				
Locality				
Proper phrasing:				
Positive tone			●	
Constructive guidance	●	●	●	●
Programmer language	●	●	●	◐

Table 11: Which principles are violated by which compilers for Error message 7.

	Compiler			
Principle	g++	MinGW	MVC++	BC++
Clarity & Brevity		●	●	●
Specificity	●			
Locality				
Proper phrasing:				
Positive tone			●	●
Constructive guidance	●	●	●	●
Programmer language	◐	◐	●	●

for the programmer to fix the problem. Possibly, the message from MVC++ can be the most helpful, but not immediately: if the programmer understand what "*local* function" means, he/she would realize why the compiler mentions this and eventually understand what is wrong. Only careful user may notice that in g++ and MinGW, the messages begin by indicating where the compiler finds the problem (in method `SavingAccount::SavingAccount()`), which may indirectly provide the programmer with a hint. The "unexpected end of file" message from MVC++, even though pointing to the last line of the file, might suggest an attentive experienced programmer that the problem is related with unbalanced braces. In all these cases, the actual problem is not made salient by the compiler, and finding the true diagnostic relies on the cognitive attention, insight and experience of the user. Regarding the phrasing, language with negative tone is used in MVC++ ("illegal", "fatal error"), and not-obvious terms for the novice programmer appear in BC++ ("type qualifier"). See Table 9 for a summary of the principles violated.

Error 6. This linking error generates messages with strange-looking names and codes (e.g., function `'ZSt17__verify_groupingPKcjRKSs'` in MinGW, or `(?Perimeter@Figure @@UAEMXZ)` in MVC++) which provide noise and no useful information. This happens in all but the message from BC++ which does not have this problem. Additionally, g++ and MinGW include implementation details ("virtual table" in g++, "vtable" in MinGW) which speaks the compiler's language, not the programmer's. The fact of mentioning the virtual table may provide some cue to those programmers in a position to make the association between this and dynamic binding. But most novice programmers, if not all, will not appreciate this subtle help, and a more direct guidance is definitely required to be helpful to all programmers. The

principle of locality is not marked as being violated since it is an error from the linker and refers to object files, not to source files. However, for this very reason, it might be argued that locality is not fulfilled. `MVC++` uses the unnecessarily worrying term "fatal error". The evaluation of which principles are violated is given in Table 10.

Error 7. The message provided by `g++` is certainly too general to point to the true source of the problem. The messages from the other compilers are more specific, but with varying degrees of usefulness. `MinGW` complains about the expressions before each of the three parameters, but saying "expected primary-expression" is unlikely to be understood: what exactly does the compiler expect before "`owner`"?

Among the lines of the message issued by `MVC++`, the third and fourth ones, referring to "no appropriate default constructor available" and to "redefinition of formal parameter", respectively, might, by being optimistic, guide the programmer. However, the programmer may easily wonder why a "default constructor" is mentioned at all, or why the parameter "`owner`" is redefined. Moreover, programmers would unlikely understand or find useful the adjective "formal" qualifying the "parameter"? The other lines add probably more confusion than help: for example, what is the `trailing '#'` if there are not any `#`? Lastly, `MVC++` also uses "illegal", which is an inadequate language blaming the programmer.

The message from `BC++` also refers to a default constructor to initialize the base class `BankAccount`. Programmers may find this puzzling since their intention is to use a specific constructor, not the default one. It is also difficult to understand what the compiler means with "unambiguous base class" and how it relates with the actual problem in the source code. Finally, the term "improper use" is not a correct way to address the the programmer. See Table 11 for an assessment of the messages of the four compilers for this error.

Acknowledgments

The authors are grateful to the anonymous reviewers for their comments and the "Servei de Llengües i Terminologia" at Universitat Jaume I for their professional English revision service.

References

[1] B. Beizer, "Software is different," *Annals of Software Engineering*, vol. 10, no. 1–4, pp. 293–310, 2000.

[2] R. Pressman, *Software Engineering: A Practitioner's Approach*, McGraw-Hill, New York, NY, USA, 2000.

[3] I. Sommerville, *Software Engineering*, Addison-Wesley, Reading, Mass, USA, 2001.

[4] J. Rumbaugh, *OMT Insights: Perspectives on Modelling from the Journal of Object-Oriented Programming*, SIGS Books, New York, NY, USA, 1996.

[5] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, Reading, Mass, USA, 1999.

[6] W. S. Humphrey, *Introduction to the Personal Software Process*, Addison-Wesley, New York, NY, USA, 1997.

[7] K. Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, Reading, Mass, USA, 2000.

[8] D. Grune, H. E. Bal, C. J. H. Jacobs, and K. G. Langendoen, *Modern Compiler Design*, John Wiley & Sons, New York, NY, USA, 2000.

[9] K. D. Cooper and L. Torczon, *Engineering a Compiler*, Morgan Kaufmann, San Francisco, Calif, USA, 2004.

[10] S. S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, San Francisco, Calif, USA, 1997.

[11] T. Schorsch, "CAP: an automated self-assessment tool to check pascal programs for syntax, logic and style errors," in *Proceedings of the 26th SIGCSE Technical Symposium on Computer Science Education*, pp. 168–172, ACM, 1995.

[12] P. G. Moulton and M. E. Muller, "DITRAN—a compiler emphasizing diagnostics," *Communications of the ACM*, vol. 10, no. 1, pp. 45–52, 1967.

[13] E. Kantorowitz and H. Laor, "Automatic generation of useful syntax error messages," *Software: Practice and Experience*, vol. 16, no. 7, pp. 627–640, 1986.

[14] A. Alexandrescu, "Better template error messages," C/C++ Users Journal. March 1999. http://erdani.com/publications/better_template_error_messages.html.

[15] T. K. Landauer, *The Trouble with Computers: Usefulness, Usability, and Productivity*, Person Educac, Person Educación, 2001.

[16] V. J. Traver, "Sobre los mensajes de error de los compiladores," in *Proceedings of the Actas del VII Congreso Internacional de Interacción Persona-Ordenador (Interacción '06)*, M. A. Redondo Duque, C. Bravo Santos, and M. Ortega Cantero, Eds., pp. 345–348, Puertollano, Spain, November 2006.

[17] R. Brooks, "Towards a theory of the cognitive processes in computer programming," *International Journal of Human Computer Studies*, vol. 51, no. 2, pp. 197–211, 1999.

[18] A. Ebrahimi, "Novice programmer errors: language constructs and plan composition," *International Journal of Human-Computer Studies*, vol. 41, no. 4, pp. 457–480, 1994.

[19] T. Teitelbaum and T. Reps, "The Cornell program synthesizer: a syntax-directed programming environment," *Communications of the ACM*, vol. 24, no. 9, pp. 563–573, 1981.

[20] M. Kölling and J. Rosenberg, "Blue—a language for teaching object-oriented programming," in *Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education*, pp. 190–194, ACM, Philadelphia, Pa, USA, March 1996.

[21] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg, "The BlueJ system and its pedagogy," *Journal of Computer Science Education*, vol. 13, no. 4, 2003.

[22] BlueJ—the interactive Java environment. http://www.bluej.org/.

[23] S. Cooper, W. Dann, and R. Pausch, "Teaching objects-first in introductory computer science," in *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, pp. 191–195, Reno,Nev, USA, 2003.

[24] "Alice: an educational software that teaches students computer programming in a 3D environment," http://www.alice.org.

[25] A. Savidis, "Rapidly implementing languages to compile as C++ without crafting a compiler," *Software—Practice and Experience*, vol. 37, no. 15, pp. 1577–1620, 2007.

[26] N. J. Coull, *SNOOPIE: development of a learning support tool for novice programmers within a conceptual framework*, Ph.D. thesis, School of Computer Science, University of St Andrews, St Andrews, Scotland, UK, 2008, http://research-repository.st-andrews.ac.uk/handle/10023/522.

[27] K. E. Gray and M. Flatt, "ProfessorJ: a gradual introduction to Java through language levels," in *Proceedings of the 18th Annual*

ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 170–177, ACM, New York, NY, USA, 2003.

[28] R. L. Shackelford and A. N. Badre, "Why can't smart students solve simple programming problems?" *International Journal of Man-Machine Studies*, vol. 38, no. 6, pp. 985–997, 1993.

[29] R. Jeffries and J. R. Anderson, "Novice lisp errors: undetecte losses of information from working memory," *Human-Computer Interaction*, vol. 1, no. 2, pp. 107–131, 1985.

[30] J. Bonar and E. Soloway, "Preprogramming knowledge: a major source of misconception in novice programmers," *Human-Computer Interaction*, vol. 1, no. 2, pp. 133–161, 1985.

[31] J. J. Cañas, M. T. Bajo, and P. Gonzalvo, "Mental models and computer programming," *The International Journal of Human-Computer Studies*, vol. 40, no. 5, pp. 795–811, 1994.

[32] M. C. Jadud, "Methods and tools for exploring novice compilation behaviour," in *Proceedings of the 2nd International Computing Education Research Workshop (ICER '06)*, vol. 2006, pp. 73–84, ACM, New York, NY, USA, 2006.

[33] C. Murphy, G. Kaiser, K. Loveland, and S. Hasan, "Retina: helping students and instructors based on observed programming activities," in *Proceedings of the 40th ACM Technical Symposium on Computer Science Education (SIGCSE '09)*, pp. 178–182, Chattanooga, Tenn, USA, 2009.

[34] M.-H. Nienaltowski, M. Pedroni, and B. Meyer, "Compiler error messages: what can help novices?" *SIGCSE Bulletin*, vol. 40, no. 1, pp. 168–172, 2008.

[35] W. M. McKeeman, "Programming language design," in *Compiler Construction: An Advanced Course*, F. L. Bauer and J. Eickel, Eds., vol. 21 of *Lecture Notes in Computer Science*, pp. 515–519, Springer, Berlin, Germany, 1976.

[36] J. F. Pane, C. Ratanamahatana, and B. A. Myers, "Studying the language and structure in non-programmers' solutions to programming problems," *The International Journal of Human-Computer Studies*, vol. 54, pp. 237–264, 2001.

[37] J. F. Pane, B. A. Myers, and L. B. Miller, "Using HCI techniques to design a more usable programming system," in *Proceedings of the IEEE Symposia on Human Centric Computing Languages and Environments (HCC '02)*, p. 198, IEEE Computer Society, Washington, DC, USA, 2002.

[38] C. R. Litecky and G. B. Davis, "A study of errors, error-proneness, and error diagnosis in Cobol," *Communications of the ACM*, vol. 19, no. 1, pp. 33–37, 1976.

[39] J. J. Horning, "What the compiler should tell the user," in *Compiler Construction: an Advanced Course*, F. L. Bauer and J. Eickel, Eds., vol. 21 of *Lecture Notes in Computer Science*, pp. 525–548, Springer, Berlin, Germany, 1976.

[40] B. Shneiderman, "Designing computer system messages," *Communications of the ACM*, vol. 25, no. 9, pp. 610–611, 1982.

[41] P. J. Brown, "Error messages: the neglected area of the man/machine interface," *Communications of the ACM*, vol. 26, no. 4, pp. 246–249, 1983.

[42] T. Flowers, C. A. Carver, and J. Jackson, "Empowering students and building confidence in novice programmers through gauntlet," in *Proceedings of the 34th ASEE/IEEE Frontiers in Education Conference (FIE '04)*, vol. 1, pp. T3H-10–T3H-13, October 2004.

[43] J. Jackson, M. Cobb, and C. Carver, "Identifying top Java errors for novice programmers," in *Proceedings of the 35th Annual Conference Frontiers in Education (FIE '05)*, pp. T4C-24–T4C-27, October 2005.

[44] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer, "What would other programmers do? Suggesting solutions to error messages," in *Proceedings of ACM Conference on Human Factors in Computing Systems (CHI '10)*, Atlanta, Ga, USA, April 2010.

[45] C. Burrell and M. Melchert, "Augmenting compiler error reporting in the Karel++ Microworld," in *Proceedings of the 20th Annual Conference of the National Advisory Committee on Computing Qualifications (NACCQ '07)*, pp. 41–46, New Zealand, 2007.

[46] D. Grune and C. J. H. Jacobs, *Parsing Techniques: a Practical Guide*, Springer, Berlin, Germany, 2nd edition, 2008.

[47] J. Scholtz and S. Wiedenbeck, "Using unfamiliar programming languages: the effects on expertise," *Interacting with Computers*, vol. 5, no. 1, pp. 13–30, 1993.

[48] B. S. Lerner, M. Flower, D. Grossman, and C. Chambers, "Searching for type-error messages," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*, pp. 425–434, San Diego, Calif, USA, 2007.

[49] L. Zolman, STLFilt: an STL error message decryptor for C++. 2005. http://www.bdsoft.com/tools/stlfilt.html.

[50] N. E. Boustani and J. Hage, "Improving type error messages for generic Java," in *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Program Manipulation (PEPM '09)*, pp. 131–140, Savannah, Ga, USA, 2009.

[51] C. L. Jeffery, "Generating LR syntax error messages from examples," *ACM Transactions on Programming Languages and Systems*, vol. 25, no. 5, pp. 631–640, 2003.

[52] A. Dix, J. Finlay, A. Gregory, and R. Beale, *Human-Computer Interaction*, Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 1998.

[53] D. A. Norman, *The Psychology of Everyday Things*, Basic Books, New York, NY, USA, 1988.

[54] M. H. Ng Cheong Vee, B. Meyer, and K. L. Mannock, "Empirical study of novice errors and error paths," Tech. Rep., ETH Zurick, Zurick, Switzerland, 2005.

[55] J. Preece, Y. Rogers, H. Sharp, D. Benyon, S. Holland, and T. Carey, *Human-Computer Interaction*, Addison Wesley, New York, NY, USA, 1994.

[56] G. A. Miller, "The magical number seven, plus or minus two: some limits on our capacity for processing information," *Psychological Review*, vol. 63, no. 2, pp. 81–97, 1956.

[57] L. Barfield, *The User Interface: Concepts and Design*, Addison-Wesley, Reading, Mass, USA, 1994.

[58] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers Principles, Techniques, and Tools*, Addison-Wesley, Reading, Mass, USA, 1986.

[59] L. Murphy, G. Lewandowski, R. McCauley, B. Simon, L. Thomas, and C. Zander, "Debugging: the good, the bad, and the quirky—a qualitative analysis of novices' strategies," in *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '08)*, pp. 163–167, ACM, 2008.

[60] B. Reeves and C. Nass, *The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places*, Cambridge University Press, New York, NY, USA, 1996.

[61] M. Hristova, A. Misra, M. Rutter, and R. Mercuri, "Identifying and correcting Java programming errors for introductory computer science students," in *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, pp. 153–156, Reno, Nev, USA, February 2003.

[62] B. Shneiderman, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, Addison-Wesley, Reading, Mass, USA, 1992.

[63] G. M. Weinberg, *The Psychology of Computer Programming*, New York, NY, USA, Dorset House Publishing, 1998, Silver Anniversary Edition.

[64] J. Nielsen, Ten usability heuristics. http://www.useit.com/papers/heuristic/heuristic_list.html.

[65] J. Yang, "Explaining type errors by finding the source of a type conict," in *Proceedings of the Scottish Functional Programming Workshop*, P. W. Trinder, G. Michaelson, and H.-W. Loidl, Eds., vol. 1 of *Trends in Functional Programming*, pp. 59–67, 2000.

[66] R. W. Picard, *Affective Computing*, The MIT Press, Cambridge, Mass, USA, 1997.

[67] R. Navarro-Prieto and J. J. Cañas, "Are visual programming languages better? The role of imagery in program comprehension," *The International Journal of Human-Computer Studies*, vol. 54, no. 6, pp. 799–829, 1999.

[68] D. A. Carrington, B. McEniery, and D. B. Johnston, "PSP in the large class," in *Proceedings of the 14th Conference on Software Engineering Education and Training (CSEET '01)*, IEEE Computer Society, pp. 81–88, Charlotte, NC, USA, February 2001.

[69] GCC, the GNU Compiler Collection. http://gcc.gnu.org/.

[70] J. Nielsen, "Heuristic evaluaiton," in *Usability Inspection Methods*, J. Nielsen and R. L. Mack, Eds., John Wiley & Sons, New York, NY, USA, 1994.

[71] S. Fitzgerald, G. Lewandowski, R. McCauley et al., "Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers," *Computer Science Education*, vol. 18, no. 2, pp. 93–116, 2008.

[72] Bloodshed Software. MinGW C++ compiler. http://www.bloodshed.net/devcpp.html.

[73] Microsoft Developer Network. Microsoft Visual C++. http://msdn.microsoft.com/en-us/visualc.

[74] Borland. C++ Builder. http://www.borland.com/.